CHASING THE “TAIL AT SCALE”: TOWARD CLOUD-NATIVE ARCHITECTURES

BY

JOVAN STOJKOVIC

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2025

Urbana, Illinois

Abstract

Cloud computing is undergoing a radical transformation with the emergence of lightweight cloud-native computing paradigms, such as microservices and serverless computing. Users build their applications by combining services, benefiting from a simplified programming model and fine-grained billing. At the same time, providers consolidate many services into a smaller number of servers, improving the utilization of their infrastructure. However, the detailed characterization of cloud-native environments presented in this thesis shows that these workloads differ significantly from traditional monolithic applications. They execute services that run for short times, exhibit bursty invocation patterns, and have frequent I/O operations that cause context switches. In addition to their core logic, services also execute many auxiliary operations known as datacenter tax, such as data serialization and encryption. Finally, services have stringent *tail latency* bounds, requiring the slowest requests to complete within a strict deadline. These characteristics result in significant inefficiencies in *performance*, *energy*, and *resource utilization* when cloud-native workloads run on conventional servers with conventional software stacks, negating the paradigm's potential benefits.

The goal of this thesis is to design hardware platforms and software stacks that enable the execution of cloud-native workloads with orders of magnitude better efficiency. The first part of the thesis designs a new hardware stack for cloud-native services. It introduces *μManycore*, a CPU architecture that minimizes the tail latency of cloud-native services. The thesis then extends the architecture with *HardHarvest* to boost utilization via hardware-based core harvesting, and refines the microarchitecture with *Mosaic* for better performance under frequent context switches. Finally, this thesis integrates on-package accelerators into the architecture and proposes *AccelFlow*, a framework that enables fine-grained, low-overhead orchestration of accelerators to reduce the datacenter tax in cloud-native environments.

To maximize the efficiency of the proposed hardware architecture, the second part of the thesis builds a full software stack that is tightly co-designed with the hardware. It begins with *MXFaaS*, a mechanism that improves resource utilization by efficiently multiplexing resources during bursts of same-function invocations. Then, it integrates the novel *Concord* distributed caching system for FaaS environments, and uses *SpecFaaS* to accelerate end-to-end application workflows through speculative service execution. Finally, this thesis improves the energy efficiency of cloud-native environments with two frameworks: *EcoFaaS*, which uses fine-grained scheduling and dynamic frequency scaling, and *SmartOClock*, which under-provisions resources and selectively overclocks cores during load spikes.

To my family, for their unconditional love and unwavering support.

Acknowledgments

First, I would like to express my deepest gratitude to my PhD. advisor, Professor Josep Torrellas, for his invaluable guidance, support, and the many opportunities he provided throughout my PhD journey. His mentorship taught me how to conduct high-quality research while encouraging me to pursue the areas that genuinely excited me. He showed me the importance of persistence and resilience, always reminding me never to give up. I vividly remember the many times I felt lost, questioning whether a PhD in Computer Science was the right path for me. During those moments, Josep was always ready to meet, offering encouragement and perspective. Our conversations helped me lift my head above the challenges and continue growing, developing what he fondly called "elephant skin." I will always remember the countless Zoom meetings preparing camera-ready papers, the practice talks before conferences, and his exceptional support during my academic job search. His dedication to mentoring and belief in my potential have left a lasting impact.

I am also grateful to Professor Tianyin Xu for his encouragement and support throughout my PhD journey. He consistently cheered for me, believed in my abilities, and helped me achieve my goals. I remember our meetings in his office during the early years of my PhD, where he patiently taught me how to write papers—how to tell a compelling story while maintaining both technical depth and breadth. Tianyin also gave me the opportunity to mentor younger students, an experience through which I grew tremendously.

I would also like to thank Dr. Hubertus Franke for his collaboration and advice throughout my PhD. From the early stages of my research, I had the privilege of working with Hubertus on several projects, including a memorable summer at IBM Research under his guidance. His energy, enthusiasm, and readiness to take on new challenges were inspiring. Through these experiences, I learned how to effectively pitch my research and present my ideas to a wide range of audiences—from domain experts to patent lawyers.

I am also deeply thankful to Professor Dimitrios Skarlatos for his mentorship and support throughout my PhD. When I was first accepted to UIUC, Dimitris was finishing his own PhD in the same lab and played an important role in helping me get started with research. We worked together on my first project, and his guidance during those early days was invaluable. Over the years, he continued to be a trusted mentor—always ready to offer advice, share his experiences, and help me navigate the many challenges of graduate school. His feedback on my academic job market materials was incredibly helpful, and I am especially grateful for the many hours he spent patiently helping me weigh my options and make decisions.

Towards the end of my PhD, I had the opportunity to collaborate with Dr. Christopher Hughes, from whom I learned a great deal about computer architecture and how to frame research ideas in ways that are both exciting and practical within the production constraints of large companies such as Intel. Chris was always ready to review paper drafts, offer thoughtful feedback, and engage in in-depth discussions about next steps and how to strengthen our work. His intelligence, energy, and kindness made working with him an incredibly rewarding experience, and I am grateful for everything I learned through our collaboration.

Next, I would like to thank the rest of my committee members—Professor Christina Delimitrou, Professor Darko Marinov, and Professor Jian Huang—for their support and valuable feedback throughout my PhD. Professor Delimitrou's work has always been a source of inspiration for me. Her research on benchmarking and analyzing cloud workloads sparked many ideas and directions in my own work, and I have long admired her contributions to the field. Professor Marinov was one of my earliest contacts at UIUC, as we share the same home country and undergraduate institution. His cheerful spirit, energy, and willingness to help made a lasting impression on me. Professor Huang served on the committees for all of my major milestones—qualifying exam, preliminary exam, and final defense—and his insightful comments and feedback consistently helped me strengthen my research and broaden my perspectives. I am grateful for their time, mentorship, and encouragement.

I would also like to thank the Microsoft Azure Research Systems group, and in particular the colleagues I closely collaborated with: Dr. Chaojie Zhang, Dr. Esha Choukse, Dr. Iñigo Goiri, Dr. Haoran Qiu, Dr. Enrique Saurez, Dr. Pulkit Misra, Dr. Sam Whitlock, Dr. Rodrigo Fonseca, and Dr. Ricardo Bianchini. I had the opportunity to spend two summers at Azure, working on exciting projects that shaped my research interests and skills. I am grateful for all the help, advice, and mentorship I received, and I appreciate how our collaborations continued even beyond the internship periods.

I would also like to thank the current and past members of the i-acoma research lab: Apostolos, Dimitris Skarlatos, Serif, Neil, Nam, Antonis, Makis, Dingyuan, Nikoleta, Chase, Burak, Jacob, Chloe, Abe, Shashwat, Filippos, and Dimitris Merkouriadis. We shared many memorable moments—whether in the office, at group seminars, during conference trips, or simply outside of research, where I made real and lasting friendships.

Whenever I needed help with anything related to graduate school, housing, traveling, or anything else, I could always rely on Apostolos, whose kindness and willingness to help meant a great deal to me. As I mentioned earlier, Dimitris was a constant and important figure throughout my PhD journey. Neil was immensely helpful when it came to organizing my job application materials, providing feedback, and always offering support; I am excited that we will continue our careers at the same institution. Antonis and I started our Ph.D.s together—he is a great friend and a roommate, both in Champaign and during our internships, where we shared unforgettable experiences. Makis is a close friend with whom I shared many wonderful memories, from going out in Champaign to traveling for conferences; his constructive feedback and support in my research were always greatly appreciated. Chloe joined our lab later as a postdoc; I learned a lot from her, and I will always be grateful for her warm heart, thoughtful advice, and readiness to listen and help. I also want to thank the new students of i-acoma I had the chance to collaborate with and become friends with during my later years—Dimitris, Filippos, and Abe.

I would also like to thank my friends—those from my hometown, childhood, high school, and undergraduate years: Ana, Dusan, Neda, Nemanja, Jovana, Anja, and Aleksandra. Thank you for always being there for me, for supporting me, and for welcoming me back with open arms no matter how much time had passed. Thank you for celebrating the good moments with me and helping me through the challenges. I am deeply grateful; this journey would never have been the same without you. In addition, I want to thank the new friends I made during my PhD, including the i-acoma group, Anna, and Ben, whose support and friendship made the experience even more meaningful.

A special thank you to Nikoleta for being there through it all—whether it was tackling research challenges, sharing daily ups and downs, traveling together to explore new places, or making each other's tough days feel lighter and each other's successes feel even sweeter. Having you alongside made this whole experience easier, a lot more fun, and truly memorable. I'm grateful we got to go through this together.

Finally, I want to thank my family. My parents, Dragana and Ivan, and my sister, Nevena, have always been my biggest supporters and the foundation of my journey. They have always believed in me, even from afar. No matter how difficult it was in the early years to go to the airport and travel to the other side of the ocean, I always knew that I had a home to return to and unwavering support behind me. There is absolutely no way I would be where I am today without them. Their love, sacrifices, and belief in me have made everything possible, and I am forever grateful.

This work was supported by NSF under grants CNS 1956007, CCF 2107470, and CCF 2316233; by Intel Corporation under the Transformative Server Architectures (TSA) Center; by the IBM-Illinois Discovery Accelerator Institute; and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

TABLE OF CONTENTS

CHAPTER 1: Thesis Overview

Cloud computing is undergoing a paradigm shift, as large monolithic applications are being replaced by compositions of many lightweight, loosely-coupled *microservices* [1]. Each microservice is implemented and deployed as a separate program, and executes a portion of the application's logic, such as key-value serving [2], protocol routing [3], or ad serving [4]. This composable application design simplifies development and enables programming language and framework heterogeneity. Moreover, each microservice can be shared among multiple applications, while being scaled and updated independently. This new paradigm is being embraced by major IT companies, such as Amazon, Netflix, Alibaba, Twitter, Uber, Facebook, and Google [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. In addition, there are many open-source systems that manage microservices, such as Kubernetes [17] and Docker Compose [18].

Building on the microservices model, *serverless* or *Function-as-a-Service (FaaS)* computing represents the next evolution in cloud-native design. It retains the modular, functionally decomposed structure of microservices, while significantly simplifying their deployment and management. Instead of requiring developers to provision and manage the infrastructure for each microservice, serverless platforms allow users to upload individual functions, with the cloud provider handling the runtime environment, system services, and scaling. Each function runs in an ephemeral, stateless container or micro virtual machine (VM) that is created and scheduled on demand in an event-driven manner. In this environment, applications can achieve high resource utilization, scale seamlessly, and benefit from fine-grained billing. Today, serverless cloud services are offered by all major cloud providers [19, 20, 21, 22] and are widely used in domains such as e-commerce [23, 24], image and video processing [25, 26], and machine learning inference and training [27, 28, 29], among many others [30].

However, after characterizing cloud-native environments, I observed that these workloads differ significantly from traditional monolithic applications. They execute services that run for short times, exhibit bursty invocation patterns, and have frequent I/O operations that cause context switches. Moreover, in addition to their core logic, services execute many auxiliary operations known as datacenter tax, such as data serialization and encryption. Finally, services have stringent tail latency bounds, requiring the slowest requests to complete within a strict deadline. These characteristics result in significant inefficiencies in performance, energy, and resource utilization when cloud-native workloads run on conventional servers with conventional software stacks, negating much of the paradigm's potential benefit.

Therefore, the goal of this thesis is to design hardware platforms and software stacks that enable the execution of cloud-native workloads with orders of magnitude better efficiency.

1.1 HARDWARE FOR CLOUD-NATIVE SERVICES

Cloud-native services demand hardware that prioritizes tail latency, fine-grained execution, and dynamic resource management. To address these needs, I designed a new architectural stack that, while remaining general-purpose, it is optimized for the unique properties of cloud-native environments. I first designed *μManycore*, a processor architecture that minimizes the tail latency of cloud-native services. Then, I extended it with *HardHarvest* to boost utilization through hardware-based core harvesting, and refined the microarchitecture with *Mosaic* for better performance under frequent context switches. Finally, I integrated on-package accelerators into this architecture and proposed *AccelFlow*, a framework that enables fine-grained, low-overhead orchestration of accelerators to reduce the datacenter tax in cloud-native environments.

μManycore [31](Chapter 3). Microservice environments execute short service requests that interact with one another via remote procedure calls (often across machines), and are subject to stringent tail-latency constraints. In contrast, current processors are designed for traditional monolithic applications. They support global hardware cache coherence, provide large caches, incorporate microarchitecture for long-running, predictable applications (such as advanced prefetching), and are optimized to minimize average rather than tail latency.

To address this imbalance, I designed μManycore, an architecture optimized for microservice environments. Based on a characterization of microservice applications, μManycore is designed to specifically target overheads that hurt tail latency. First, as services are relatively small but users invoke many of them concurrently, μManycore does not rely on having only a few large cores. Instead, it incorporates many small cores for the same total power and area budget. As a result, the system reduces queuing without substantially hurting single-request performance.

Providing manycore-wide cache coherence is expensive. In fact, it is hardly needed, as microservices hardly use shared memory to communicate. Hence, rather than supporting manycore-wide hardware cache coherence, μManycore has multiple small hardware cache-coherent domains, called *Villages.* In a village, services can communicate using shared memory, while across villages, they communicate using network messages.

Clusters of villages are interconnected with an on-package leaf-spine network, which has many redundant, low-hop-count paths between clusters. To minimize latency overheads, μManycore schedules and queues service requests in hardware, and includes hardware support to save and restore process state when doing a context-switch.

HardHarvest [32](Chapter 4). While μManycore improves tail latency for microservices, it does not tackle another critical inefficiency of these environments—chronically low core utilization due to overprovisioning for peak demand. In microservice environments, users size their virtual machines (VMs) for peak loads, leaving cores idle most of the time. To improve core utilization and overall throughput, it is instructive to consider a recently-introduced software technique for environments with relatively long running monolithic applications: Core Harvesting. With this technique, Harvest VMs running batch applications temporarily steal idle cores allocated by Primary VMs running latency-critical applications, and return them on demand. Unfortunately, re-assigning cores across VMs has substantial overhead, resulting from hypervisor calls, context switching, and flushing TLBs/caches. While such overhead is acceptable in monolithic application environments, it would be prohibitive in environments with sub-millisecond microservices.

To address this problem, I designed the first architecture for core harvesting *in hardware*. The architecture, called *HardHarvest*, targets cloud-native services. It aims to: 1) maximize core utilization, 2) minimize any impact on Primary VM tail latency, and 3) boost Harvest VM throughput. Building on the μManycore architecture, HardHarvest eliminates the software overheads of core harvesting by using in-hardware request scheduling, and partitioning TLBs/caches with a smart replacement algorithm.

Mosaic [33](Chapter 5). As μManycore provides high core counts and HardHarvest improves utilization through dynamic core sharing, the rate of context switches rises sharply, leading to persistent disruption of stateful microarchitectural structures such as caches, TLBs, and branch predictors. These structures, traditionally sized and managed for long-running applications, are poorly matched to the ephemeral and fragmented execution patterns of cloud-native services. The result is diminished performance due to lost locality, excessive misses, and increased energy usage.

Based on these insights, I proposed *Mosaic*, a microarchitecture optimized for cloud-native environments that maintains generality to efficiently support other workloads. Mosaic has two components: (1) *MosaicCPU*, a core microarchitecture that efficiently runs both cloud-native workloads and traditional monolithic applications, and (2) *MosaicScheduler*, a software stack for cloud-native systems that maximizes the benefits of MosaicCPU. MosaicCPU slices micro-architectural structures into small chunks and assigns tiles of such chunks to individual services. The processor retains the state of services in their tiles across context switches, improving performance. Furthermore, currently-inactive tiles are set to a low power mode, reducing energy consumption. MosaicScheduler maximizes efficiency via predictive right-sizing of the per-service tiles, along with smart scheduling based on the state of the tiles.

AccelFlow [34](Chapter 6). In addition to their main application logic, cloud-native services suffer from the execution of auxiliary operations known as datacenter tax, such as remote procedure call (RPC) processing, transmission control protocol (TCP) processing, data (de)serialization, data (de)encryption, and data (de)compression. To minimize this tax, multiple hardware accelerators have been proposed. However, it is unclear how these accelerators should be orchestrated. Past work has focused only on orchestrating accelerators in coarse-grained environments with monolithic applications.

By characterizing the needs of orchestrating an ensemble of on-package accelerators in cloud-native environments, I observed that such orchestration frameworks need to be highly dynamic and nimble. The sequences of accelerators vary across invocations of the same service, and the basic operations to be accelerated are fine grained, potentially taking only tens of μs. Moreover, the sequence of accelerators is often affected by "branch conditions" whose real-time resolution determines the set of subsequent accelerators needed. To address these challenges, I designed *AccelFlow*, the first orchestration framework for on-package accelerators in cloud-native environments. In AccelFlow, CPU cores build software structures called *Traces* that contain sequences of accelerators to call. A core enqueues a trace in an accelerator in user mode and, from then on, the accelerators in the trace execute in sequence without CPU involvement. A trace can include branch conditions whose outcomes determine the control flow inside the trace. Overall, the requests move from one accelerator to another with minimal overheads, allowing the CPU cores to focus on the main application logic.

1.2 SOFTWARE FOR CLOUD-NATIVE SERVICES

To maximize the efficiency of the proposed hardware architecture for cloud-native environments, I built a full software stack that is tightly co-designed with the hardware. I began with *MXFaaS*, a mechanism that improves resource utilization by efficiently multiplexing resources during bursts of same-function invocations. I also integrated a distributed caching layer called *Concord* to reduce overheads from frequent remote storage accesses. I then accelerated end-to-end application workflows using speculative function execution with *SpecFaaS*. Finally, I improved energy efficiency with *EcoFaaS* through fine-grained scheduling and dynamic frequency scaling, and with *SmartOClock* through under-provisioning of resources and selectively overclocking cores during load spikes.

MXFaaS [35](Chapter 7). Cloud-native workloads frequently exhibit bursts of invocations of the same service. For example, different end-users can create bursts of invocations of popular services, triggered by certain events. Alternatively, a single end-user may issue thousands of invocations of the same service to parallelize data processing. Such pattern is not handled well by current serverless platforms, which end up spawning many containers. Supporting this invocation pattern efficiently can speed-up serverless execution substantially and improve the resource utilization of serverless environments.

I target this dominant pattern with a new serverless platform design named *MXFaaS*. MXFaaS improves function performance by efficiently *multiplexing* (i.e., sharing) processor cycles, I/O bandwidth, and memory/processor state between concurrently executing invocations of the same function. MXFaaS introduces a new container abstraction called *MXContainer*. To enable efficient use of processor cycles, an MXContainer carefully helps schedule same-function invocations for minimal response time. To enable efficient use of I/O bandwidth, an MXContainer coalesces remote storage accesses and remote function calls from same-function invocations. Finally, to enable efficient use of memory/processor state, an MXContainer first initializes the state of its container and only later, on demand, spawns a process per function invocation, so that all invocations can share unmodified memory state and hence minimize memory footprint.

Concord [36](Chapter 8). With MXFaaS, functions reduce their startup times and improve resource utilization through efficient multiplexing, but their individual performance remains bound by frequent accesses to remote global storage. Costly accesses to global storage substantially limit the performance of serverless functions. To mitigate this overhead, data can be cached in the memory of the nodes where functions are executed. However, existing caching schemes either (1) restrict a data item to be cached in a single node, causing frequent remote reads or (2) allow a data item to be cached in multiple nodes concurrently, adding substantial overhead to maintain cache coherence. These current approaches are suboptimal for the access patterns present in serverless workloads, which are characterized by frequent reads to small data items, strong temporal locality, and a small number of nodes concurrently executing functions of the same application.

Driven by these insights, I proposed *Concord*, a distributed software caching system tailored to serverless environments. Concord uses MXFaaS within a node, and scales to a distributed setup. It allows multiple copies of the same data item to be cached in different nodes concurrently, allowing each cache to satisfy local reads. To maintain coherence across software caches, Concord proposes a directory-based distributed coherence protocol. The protocol is inspired by hardware cache coherence, and is enhanced to minimize coherence traffic, reduce contention points, and be robust to node failures and frequent coherence domain changes. Further, Concord unlocked two new capabilities in serverless environments: transactional storage accesses and transparent data-aware function placement.

SpecFaaS [37](Chapter 9). While improving the performance of a single function is important, serverless applications are typically composed of multiple functions chained together. To accelerate serverless application workflows, I extended MXFaaS and Concord systems with a novel execution model based on software-supported *speculative execution* of functions. The proposal is termed *Speculative Function-as-a-Service* (*SpecFaaS*). It is inspired by out-of-order execution in modern processors, and is grounded in a characterization analysis of FaaS applications. In SpecFaaS, functions in an application are executed early, speculatively, before their control and data dependences are resolved. Control dependences are predicted like in pipeline branch prediction, and data dependences are speculatively satisfied with memoization. With this support, the execution of downstream functions is overlapped with that of upstream functions, substantially reducing the end-to-end execution time of applications.

EcoFaaS [38](Chapter 10). Serverless platforms should maintain high performance and good resource utilization while using minimal energy. Unfortunately, the energy and power consumption behavior of serverless systems has hardly been explored. Hence, I performed a thorough characterization and observed that serverless environments pose a set of challenges not effectively handled by the existing energy-management schemes. Short serverless functions execute in opaque virtualized sandboxes, are idle for a large fraction of their invocation time, context switch frequently, and are co-located in a highly dynamic manner with many other functions of diverse properties. These features are a radical shift from more traditional application environments and require a new approach to manage energy and power.

Using these insights, I extended my MXFaaS serverless platform design with *EcoFaaS*, the first energy management framework for serverless environments. EcoFaaS takes a user-provided end-to-end application Service Level Objective (SLO). It then splits the SLO into per-function deadlines that minimize the total energy consumption. Based on the computed deadlines, EcoFaaS sets the optimal per-invocation core frequency using a prediction algorithm. The algorithm performs a fine-grained analysis of the execution time of each invocation, while taking into account the specific invocation inputs. To maximize efficiency, EcoFaaS splits the cores in a server into multiple Core Pools, where all the cores in a pool run at the same frequency and are controlled by a single scheduler. EcoFaaS dynamically changes the sizes and frequencies of the pools based on the current system state.

SmartOClock [39](Chapter 11). I further enhanced the efficiency of EcoFaaS with processor overclocking during load spikes of cloud-native services. Operating server components beyond their voltage and power design limit (i.e., overclocking them) enables improved performance and lower cost for cloud-native workloads. However, overclocking can significantly degrade component lifetime, increase power draw, and cause power capping events, eventually diminishing the performance benefits.

I characterized the impact of overclocking on cloud workloads by studying their profiles from Microsoft Azure production deployments. Based on the characterization, I proposed SmartOClock, the first distributed overclocking management platform specifically designed for cloud environments. The idea is to allocate the resources to cloud-native services based on their average load, and then, during load spikes, overclock the cores to compensate for resource under-provisioning. SmartOClock is a workload-aware scheme that relies on power predictions to heterogeneously distribute the power budgets across its servers based on their needs and then enforces budget compliance locally, per-server, in a decentralized manner.

1.3 IMPACT OF THIS THESIS

My research has helped advance cloud computing. My *μManycore* processor architecture for microservices was recognized with an IEEE Top Picks Honorable Mention and received great attention from academia and industry partners such as Intel, IBM, and Microsoft. Further, four patents have been filed with IBM and Microsoft for my work on serverless software stack design (*MXFaaS*, *SpecFaaS*, and *Concord*) and processor overclocking in the cloud (*SmartOClock*). Building on insights from my research on cloud-native workloads, I also designed the first power and energy management systems for LLM inference in the cloud—*DynamoLLM* and *TAPAS*. Both works led to patents filed with Microsoft. Additionally, *DynamoLLM* was recognized with *the Best Paper Award* at HPCA. Notably, versions of my *SmartOClock* and *TAPAS* systems are now *being implemented* in Microsoft Azure production environments.

CHAPTER 2: General Background on Cloud-Native Services

2.1 MICROSERVICE ENVIRONMENTS

In microservice environments (e.g., managed by Kubernetes [17] or Docker Compose [18]), large complex applications are organized as workflows of multiple interdependent services. Figure 2.1 shows an example of a microservice-based application (*ComposePost*) from the DeathStarBench suite [40]. Each service, e.g., *Text* or *SGraph*, performs its dedicated functionality, communicates with other services, and scales independently of other services.

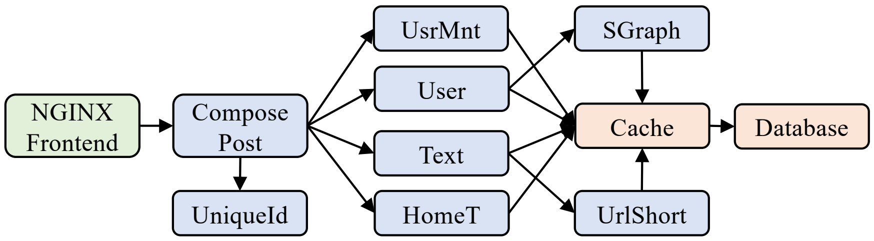


Figure 2.1: *ComposePost* microservice-based application. Blue boxes represent microservices. Green and orange boxes represent frontend and backend helper applications.

Requests for a given microservice are served by one or more *Instances* present as separate VMs or containers. When created, a VM is assigned a certain number of cores and amount of memory that it can use. Cloud providers use these resource limits to pack the VMs on servers. To accommodate the peak load, users typically overprovision VMs, leaving resources such as cores underutilized throughout the majority of the VM lifetime. Even when there is substantial load for the microservice, requests do not fully utilize cores, as cores often stall on synchronous RPCs to read/write to/from remote storage, or to invoke other microservices.

Often, a service request invokes one or more other services that perform simple operations and then aggregates the obtained data. Studies by Alibaba [5] and Facebook [14] show that such a multi-tier paradigm is popular in production-level microservice architectures. Services communicate with each other via RPC/HTTP protocols, such as gRPC [41] and eRPC [42]. When a service request calls another service synchronously, it waits on the results before continuing the execution. This operation also introduces potentially significant stall times.

Individual microservices are significantly simpler than their monolithic counterparts. They have a smaller memory footprint and working set, less pressure on instruction fetching, and orders of magnitude shorter execution time. However, in reality, these environments have substantial performance challenges. Short execution times and frequent, costly remote storage accesses and communication between services induce overheads that cannot be overlooked [2, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52].

2.2 SERVERLESS PLATFORMS

With serverless computing, developers still follow the microservice-based software architecture. Additionally, developers focus only on the main application logic, while cloud providers are in charge of provisioning all the resources and infrastructure needed to run such serverless applications. Figure 2.2 illustrates a typical architecture of a serverless platform, such as OpenWhisk, KNative, OpenFaaS, or OpenLambda [53, 54, 55, 56, 57, 58, 59, 60]. A platform consists of centralized control modules (e.g., the *frontend* and the *load balancer*) that accept function invocations and distribute them to the nodes. In each node, there is an *invoker* module that is responsible for the execution of the function.

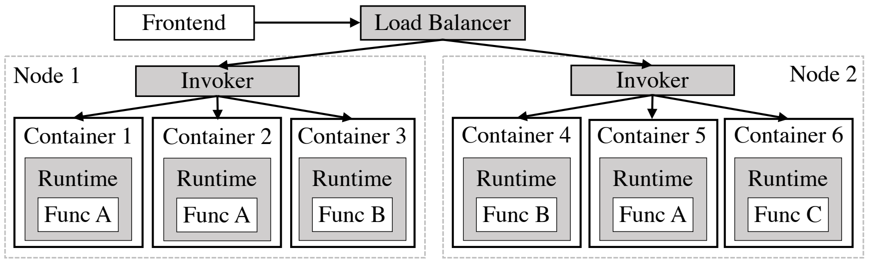


Figure 2.2: Overview of conventional serverless platforms.

To execute a user-provided function, an invoker encapsulates the function code together with an execution runtime in a container, and spawns the runtime process [61]. The user function is executed inside the address space of the runtime process [62]. The runtime process first executes initialization code to set up network connections and initialize global variables. Then, it invokes the requested function. The runtime only services one request (i.e., one function invocation) at a time. To execute multiple invocations of the same function concurrently, production-level platforms need to spawn multiple containers.

When a function invocation completes, the serverless platform keeps the container in memory in warm state for a certain period of time [63, 64, 65]. If, during this period, another request for the same function is received, it is executed by the runtime process in the warm container. However, with this approach, the global state of the runtime process is sequentially shared across function invocations. Such sharing may lead to security and correctness issues [62].

CHAPTER 3: Processor Architecture for Tail Latency of Cloud-Native Services

3.1 INTRODUCTION

Microservice environments have new characteristics that impact the system and hardware architecture of the platforms on which they run. Specifically, requests for microservices in an application are typically short-running and may execute on different machines. Requests for different microservices share no memory state and interact with one another via remote procedure calls (RPCs) [41, 42]. Further, requests have small working sets and are often invoked in bursts, frequently waiting in queues before being executed. Finally, the decomposition of an application places tight sub-ms latency Service Level Objectives (SLOs) on individual services [2, 50]. As a result, while reducing average latency and improving throughput are important, the key performance target in these environments is now *minimizing tail latency* [66] (e.g., improving the 99th-percentile responses). Many of these characteristics are also found in emerging deployment methods based on microservices, such as Function-as-a-Service (FaaS) environments [19, 20, 21, 22].

Current processors are not expressly designed for these environments. Indeed, multicores invest significant hardware and design complexity to support global hardware cache coherence. They have large caches to capture the working sets of long-running applications. They are relatively unconcerned with supporting short-running, RPC-communicating programs. Instead, they incorporate microarchitectural optimizations for long-running, predictable applications, such as advanced prefetchers and branch predictors. These optimizations add significant hardware complexity and are, at best, marginally effective for microservices. Perhaps most importantly, current processors are highly optimized to minimize the average latency of programs or transactions, and ignore tail-latency considerations.

How should one change the design of processors so that they match microservice requirements? First, some of the hardware optimizations that introduce design complexity and are hardly needed by microservices, such as global hardware cache coherence, should be reconsidered. Second, there should be a comprehensive effort to optimize for tail-latency reduction. Optimizations should target both inefficiencies affecting all requests, and contention-based overheads that may affect a subset of requests. The resulting processor might not be competitive for general-purpose loads, it can be the CPU of choice for microservice datacenters.

In this chapter, we propose a processor architecture highly optimized for cloud-native microservices. We call it *μManycore*. *μManycore* is not an accelerator; it retains general-purpose processor capabilities, while it may not be as competitive for monolithic applications.

To design *μManycore*, we first characterize production-level microservice traces from Alibaba [5] and microservices from DeathStarBench [40]. Our analysis shows that bursty service requests create periods of high demand where long waiting queues are likely to appear. In addition, requests spend most of their time blocked, waiting for the completion of their accesses to storage or their calls to other services. In the meantime, CPUs context switch frequently, introducing overhead. Moreover, service-initiated messages between cores experience the latency of interconnection networks (ICNs), often suffering contention delays that further increase tail latency. Finally, while requests have small working sets, microservices benefit from a large nearby pool of memory that stores per-service read-mostly state.

Based on these findings, we design a chiplet-based *μManycore*. Rather than supporting package-wide hardware cache-coherence, *μManycore* is built with multiple small hardware cache-coherent domains called *Villages*. Microservices are assigned to individual villages. A few villages, together with a memory chiplet (storing read-mostly state), are grouped in a cluster. Clusters are interconnected with a leaf-spine ICN [67, 68]. This topology has many redundant, low-hop-count paths between any two clusters—hence, minimizing contention between multiple messages with the same source and destination clusters and reducing tail latency. To minimize scheduling overheads, *μManycore* enqueues, dequeues, and schedules service requests in hardware. Finally, to minimize the overhead of frequent context switching, cores include hardware support to save and restore process state.

Our simulation results show that *μManycore* delivers high performance for microservices. We compare a 1024-core *μManycore* to two conventional server-class multicores: one with the same power and one with the same area as *μManycore*. A cluster of 10 *μManycore* servers delivers 3.7× lower average latency, 15.5× higher throughput, and, importantly, 10.4× lower tail latency than a cluster with the iso-power conventional multicores. Similar results are attained compared to a cluster with the power-hungry iso-area conventional multicores.

This chapter makes the following contributions:

- A characterization of microservice workload behavior in conventional processors.
- *μManycore*, a processor architecture highly optimized for microservice workloads.
- An evaluation of *μManycore*, comparing it to two conventional server-class multicores: one with the same power and one with the same area.

3.2 THE NEED FOR A CLOUD-NATIVE CPU

The ever-increasing complexity of software systems has kept pushing forward processor design. For example, researchers have proposed numerous prefetching, branch prediction, and cache replacement schemes. These proposals introduce custom microarchitectural structures that increase processor area, power consumption, and design complexity in order to improve application performance.

However, many of these optimizations hardly benefit cloud-native microservices. To validate this hypothesis, we consider four published microarchitectural optimizations for which the simulator and applications used in the publications are open sourced. For each of the optimizations, we first run the original applications [69, 70, 71, 72, 73] on the original simulator and record the performance with and without the optimizations. The results are depicted as bars *Baseline* and *Optimized* in *Mono* (for Monolithic) in Figure 3.1, normalized to *Baseline*. We then run a set of microservice applications—SocialNetwork from DeathStarBench [40], and Router and SetAlgebra from μSuite [50]—on the original simulator and record the performance with and without the optimizations. The results are depicted as bars *Baseline* and *Optimized* in *Micro* (for Microservice) in Figure 3.1, normalized to *Baseline*.

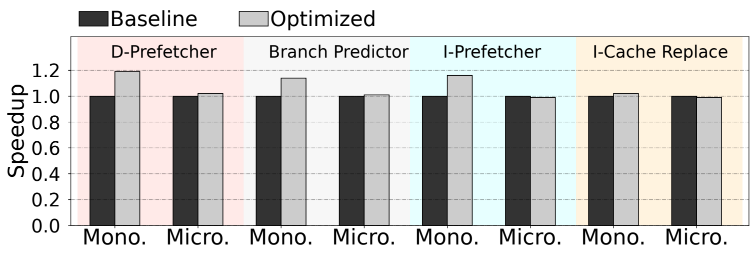


Figure 3.1: Performance improvements of four recently-proposed microarchitectural optimizations using monolithic (*Mono*) and microservice (*Micro*) applications. For each optimization and application set, the bars are normalized to *Baseline*.

The optimizations are as follows:

D-Prefetcher shows the impact of the Pythia reinforcement-learning data prefetcher [74]. Pythia speeds-up monolithic applications by 19% on average over a system without a prefetcher. However, it brings only marginal benefits of 2% to microservices.

Branch Predictor shows the impact of a perceptron-based branch predictor [75]. The predictor speeds-up monolithic applications by 14% on average over a system with a simple g-share predictor. On the other hand, the predictor speeds-up microservice applications by only 1% on average over the g-share predictor.

I-Prefetcher shows the impact of the I-SPY context-driven instruction prefetcher [76]. The prefetcher speeds-up monolithic applications by 16% on average over a system without instruction prefetcher. On the other hand, it does not speed-up microservice applications.

I-Cache Replace shows the impact of the Ripple profile-guided instruction cache replacement algorithm [77]. The algorithm speeds-up monolithic applications by 2% on average over a system with LRU replacement. However, it does not bring any benefits to microservices.

The reason for the discrepancy in the effectiveness of the proposed optimizations is the reduced data and instruction memory footprint of the microservice workloads compared to the monoliths, as well as their increased cache hit rates and different branch behavior. This data shows that a different type of processor microarchitecture is needed to speed-up microservice applications.

3.3 CHARACTERIZING MICROSERVICES ON CURRENT PROCESSORS

To guide the design of *μManycore*, we first characterize the behavior of microservice applications on current processors. We execute the DeathStarBench [40], TrainTicket [78], and μSuite [50] open-source microservice application suites, as well as real-world production-level microservice execution traces from Alibaba [5]. Our main conclusions are described next.

3.3.1 Monolithic Cache Coherence Provides Limited Advantage

To enable high availability, fast scalability, and fault tolerance, microservice applications are implemented as sets of services. Each service is built as a standalone RPC/HTTP server—in our workloads, a TThreadedServer [79], RestController [80], HTTPServer [81], or gRPCServer [41]. Upon service instance initialization, network connections are set up, libraries are loaded, and preparation code is executed. Upon service request arrival, the service instance spawns a new worker (a process, thread, or co-routine) or reuses an existing one to serve the request.

Different services or different instances of the same service do not share any modifiable memory. A worker can update its private state, the local state of its service instance and, with RPC calls, the state in global storage. This is in contrast to traditional multi-threaded applications, where concurrently-running threads are often free to share memory.

Given this environment, conventional monolithic hardware cache coherence, as it is used in current large multicores, is hard to justify. Cache coherence is only needed inside a service instance, which typically uses only a few cores. One could argue that global coherence would still be needed if we allowed service instances to migrate across any cores. However, unimpeded migration of service instances across a large 1K-core multicore is unlikely to deliver performance improvements and, in fact, is likely to increase tail latency. Hence, given the well-known hardware complexity and scalability challenges of large-scale hardware cache-coherence, it is more reasonable to support only small-scale cache-coherence domains among the cores used by individual service instances. Service requests for a given instance can still migrate between the cores used by the service instance if needed for load balance—resulting in a more efficient environment.

3.3.2 Bursty Requests Increase Tail Latency

We use Alibaba's production-level traces [5] to characterize the arrival rate of service requests. The traces include requests directed to 10,000 servers. In each server, service requests arrive in bursts, creating periods of high and low request demands. Figure 3.2 shows the CDF of the number of Requests per Second (RPS) arriving at a server [82, 83]. We can see that a server that gets and processes a median of ≈500 RPS, sometimes gets multiple times these many RPS—i.e., 20% of the time, it receives 1,000 RPS or more, and in 5% of the time, it receives 1,500 RPS or more. When these large numbers of requests are received, they have to wait in queues.

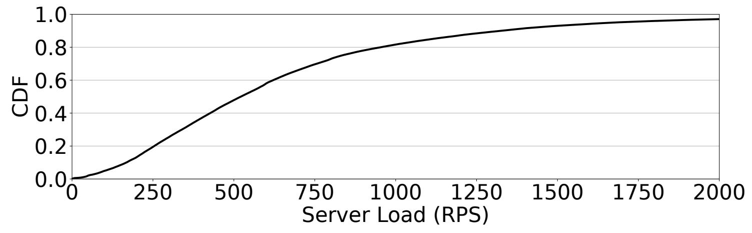


Figure 3.2: CDF of Requests per Second (RPS) received by a server.

Given this environment, it is important to design queuing systems that have minimal overhead. Previous proposals have considered a fully-centralized First-Come-First-Serve (FCFS) queue [47, 48]. However, under high concurrency, such an approach induces high synchronization overheads. At the other extreme, one can have fully-decentralized FCFS queuing, with a per-core queue. This approach is equally undesirable, as it leads to load imbalance and head-of-line blocking.

Any suboptimal queuing structure will lead to increased average response time for the service requests. Most importantly, it will have a major impact on the *tail* response time.

To see this effect, we take the DeathStarBench applications [40] and run them on a simulated 1024-core ScaleOut manycore (described in Section 3.5). We issue requests using a Poisson distribution with 50K RPS, on average. Figure 3.3 shows the average and tail response time of the requests as we vary the number of queues in the manycore. The leftmost point (1024) means that each core has a dedicated queue, and the next one (512) that every two cores share one queue, and so on. In the rightmost point, all cores share a single queue.

Requests are assigned to queues randomly. In addition, we evaluate a system that allows a core to steal requests from other queues when its assigned queue is empty.

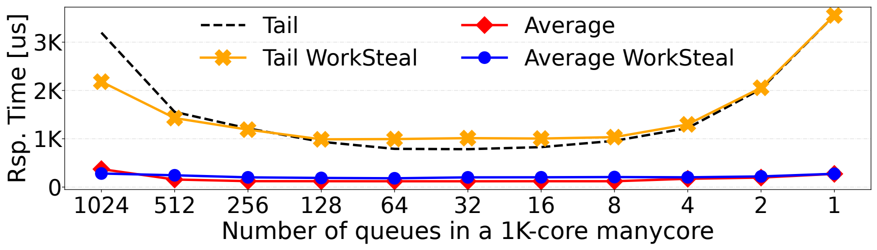


Figure 3.3: Average and tail response time of requests for different numbers of queues in a 1024-core manycore.

The figure shows that the average response time increases modestly as we go from the best scenario (32 queues with 32 cores per queue) to the worst ones (1024 queues or one queue). However, the tail response time changes dramatically. With 1024 queues and with one queue, the tail is 4.1× and 4.5× higher, respectively, than the tail with 32 queues.

Work stealing significantly reduces the tail when the system has one queue per core. This is because it mitigates load imbalance. However, as we increase the number of cores per queue, and thus reduce the load imbalance, work-stealing becomes less useful and even increases the tail due to the added overheads. Work stealing does not change the average latency.

3.3.3 Context Switching Hurts Tail Latency

We now use Alibaba's traces to characterize the execution of service requests. We find that requests are typically very short: 36.7% of the dynamic invocations take less than 1ms; the geometric mean duration of the remaining dynamic invocations is 2.8ms. In addition, service requests spend most of that time waiting (i.e., blocked) on I/O. Specifically, the median CPU utilization is only ≈14%. Further, 99% of the requests utilize the CPU less than 60%. The reason for the low utilization is the frequent execution stalls due to RPC invocations: the request execution is blocked waiting for the completion of storage requests or calls to other services. A request performs a median of ≈4.2 RPC invocations. Moreover, about 5% of the requests invoke 16 or more RPCs.

As another data point, in the DeathStarBench applications [40], the average execution time of a service request is 120μs, and the average request performs 3.1 RPC invocations.

To use resources efficiently in microservice environments, CPUs need to context switch every time a request blocks on an RPC invocation. The cost of a context switch is ≈5K cycles in Linux-based systems and ≈2K cycles in state-of-the-art software schedulers [48].

This may be negligible for monolithic applications, where the time between context switches is much larger than the context switch overhead. However, this is not true for microservices.

To assess the impact of context-switch overhead on the request tail latency, we simulate the execution of the 1024-core ScaleOut manycore invoking the services of the SocialNetwork application from DeathStarBench with a Poisson distribution with 5K, 10K, and 50K RPS. We add a certain amount of Context Switch overhead cycles (CS) every time they suffer a context switch. We vary CS from zero to 8K cycles. Figure 3.4 shows the tail latency of the requests. The tail latency is normalized to the one with zero CS cycles. The figure shows the range of CS cycles that are typical for Linux, and for the state-of-the-art Shenango, Shinjuku, and ZygOS software schedulers [48].

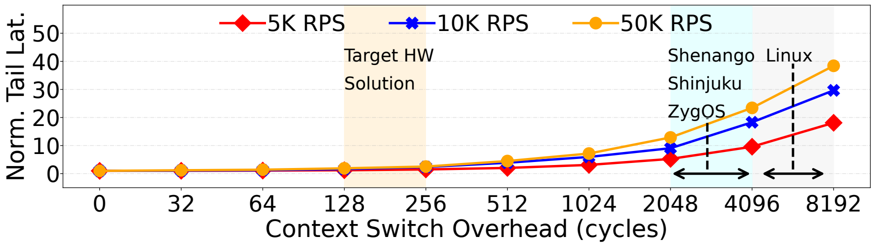


Figure 3.4: Impact of the context switch overhead on the tail latency with different requests per second (RPS).

These context-switch overhead cycles delay request processing. We see that the impact is significant, especially for larger loads. For 50K RPS, the context-switch overhead of Linux degrades the tail latency of requests by 26–38×; the context switch overhead of state-of-the-art software schedulers degrades it by 13–23×. Ideally, we would like a CS of around 128–256 cycles which, as shown in the figure, barely impacts the tail latency. Such CS requires hardware support.

3.3.4 The Interconnect Impacts Tail Latency

In microservice environments, request execution triggers interconnection network (ICN) messages, as it issues storage requests and calls to other services. Such messages compete for ICN links, and potentially suffer contention delays. Such delays directly impact the tail latency. Consequently, the design and implementation of the ICN play a significant role in determining the tail latency. In this chapter, we are interested in the on-package ICN.

To assess this effect, we take the DeathStarBench applications and run them on the simulated 1024-core ScaleOut manycore, issuing Poisson-distributed requests with 1K, 5K, 10K, and 50K RPS. Cores are grouped in 32-core clusters, and the clusters are interconnected with either a 2D mesh or a fat-tree ICN. The contention-free hop-to-hop latency of the ICN is 5 cycles. Service requests are issued to cores randomly. Figure 3.5 shows the resulting request tail latency. Each bar is normalized to the tail latency of the same environment without ICN contention.

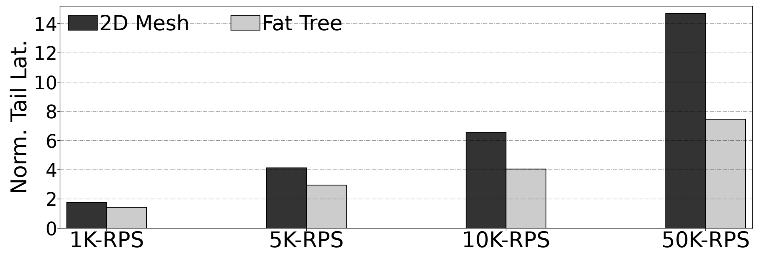


Figure 3.5: Impact of contention in the on-package interconnection network (ICN) on the tail latency of requests. Each bar is normalized to the tail latency without ICN contention.

The figure shows that contention in the ICN has a substantial impact on tail latency. With 50K RPS, contention in the 2D mesh ICN increases the tail latency by 14.7× on average. For the fat-tree ICN, the increase is 7.5× on average. Therefore, the ICN should be carefully designed to minimize contention.

3.3.5 Large Read-Mostly Memories of Service Instances & Small Working Sets of Requests

When a service instance is created, it initializes its state, which includes its container, runtime, and libraries. To save initialization overhead, microservice systems may store *Snapshots* of services in memory with all the initialization state. This is especially important in FaaS, where containers are created much more frequently [84, 85, 86]. Then, when a new instance is created, all that it needs to do is to simply read its corresponding snapshot. Hence, for performance reasons, it is important to keep snapshots in a *near read-mostly memory.* For DeathStarBench applications, snapshots reduce the boot time of a service instance from over 300ms to less than 10ms, while using less than 16MB of memory per service [86].

In addition, every time a request is received for a service, the service instance spawns a new handler. All handlers of a service instance read some of the instance's initialization data. Moreover, as the handlers execute the same code, they read mostly the same instructions. As a result, different handlers of the same service instance have very similar instructions and read-data footprints.

A handler's memory footprint is small. On average for the DeathStarBench applications, it is only 0.5 MB. Figure 3.6 considers the memory footprint of a handler (normalized to 1). In the *Handler-Handler* bars, it shows what fraction of the footprint is *common* (and hence can be read-shared) with another handler of the same service instance. In the *Handler-Init* bars, the figure shows what fraction of the handler footprint is common (and hence can be read-shared) with the initialization process of the service instance. In each of the two groups, from left to right, the normalized bars show the data footprint in pages, the data footprint in cache lines, the instruction footprint in pages, and the instruction footprint in cache lines. Pages are 4KB and cache lines are 64B. All bars are averaged across all applications.

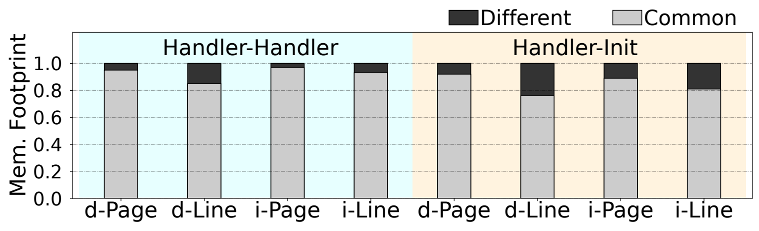


Figure 3.6: Handler-handler and handler-initialization sharing of data and instruction pages and cache lines.

The figure shows that, on average, the fraction of pages or cache lines that are common between two handlers or between a handler and its initialization process is 78–99%. Consequently, a manycore architecture for microservices can benefit from having read-shared memories that are accessed by multiple requests of the same service instance.

Because of the small footprint of handlers, requests put little pressure on the cache hierarchy. This is in contrast to monolithic applications, which require ever bigger caches [76, 77, 87]. Figure 3.7 shows the average hit rates of L1 and L2 TLBs and caches, both for data and instructions, for the architecture in Table 3.2, which will be discussed later. We observe that, for the L1 TLB and cache, the hit rates of both data and instructions are above 95%. Hence, the working sets fit in L1 TLB and cache. The L2 TLB and cache have lower hit rates; this is because the L1 structures act as filters, intercepting the high-locality accesses. As a result, a manycore architecture for microservices can use small caches and reduced-depth cache hierarchies (e.g., hierarchies of only two levels of caching). The resources saved can be invested in supporting more parallelism.

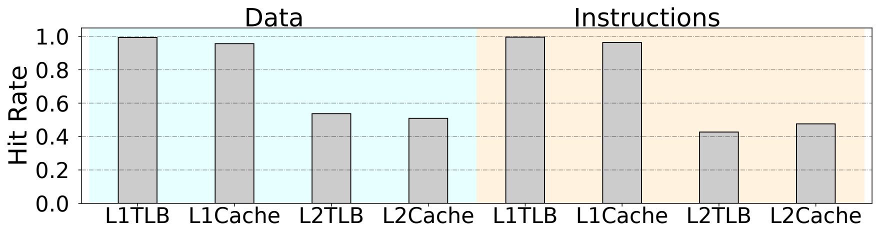


Figure 3.7: L1 and L2 data/instructions TLB and cache hit rates.

3.4 *μMANYCORE*: A CLOUD-NATIVE CPU

This chapter proposes *μManycore*, a processor designed for microservices. In microservice environments, a key objective is to minimize the tail latency of requests. Hence, *μManycore* is designed to minimize the primary overheads that contribute to the tail latency. Some of these overheads impact both tail and average latency—i.e., overheads that, to a large extent, affect all service requests. Other overheads impact mainly tail latency—i.e., overheards that disproportionately impact some requests, such as overheads resulting from contention effects. *μManycore* addresses both types of overheads.

The characterization of Section 3.3 gives insights into the main sources of tail latency in microservice environments. Table 3.1 lists such sources, the reason why they exist, and how the *μManycore* design avoids them. In the following, we consider each of these sources in turn. We assume a large manycore with 1024 cores.

Table 3.1: Main sources of tail latency.

Source	**Reason**	***μManycore* Solution**
Monolithic cache coherence	Remote directory/cache/network accesses (some due to migration) and contention	Multiple small cache coherent domains
Request scheduling	Synchronization and queuing of requests	Request enqueuing, dequeuing, and scheduling in hardware
Context switching	OS invocation and saving & restoring state	Hardware-based context switching
On-package network	Network link/router latency (some due to contention)	On-package hierarchical leaf-spine network

1. Monolithic Cache Coherence. As indicated in Section 3.3.1, requests for different service instances do not share memory state. They communicate through remote storage accesses and through service calls, both of which use RPCs. Hence, they do not require monolithic cache coherence. Providing monolithic cache coherence in a manycore typically results in remote directory and network accesses, which increase tail latency. The only reason to provide monolithic cache coherence would be to support service instance migration across cores for load balance. However, unrestricted instance migration across a large manycore results in (1) remote cache accesses to obtain data from caches in cores where the instance used to run, (2) more remote directory accesses, (3) additional network traffic, and (4) increased contention. The result would be increased tail latency.

In practice, there are some reasons to support modest-size cache coherence domains. First, some services are multithreaded. Second, allowing requests for a given service instance to migrate between the cores used by the instance can improve load balance. Finally, supporting some hardware cache coherent domain ensures that the manycore remains general purpose. Consequently, in *μManycore*, we eliminate monolithic hardware cache coherence and, instead, have multiple small hardware cache-coherent domains. These domains are called *Villages*. Each service instance is assigned to a village. A service request is allowed to migrate between the cores of its village for load balance, and to execute in parallel on the cores of its village.

Message-passing designs such as Intel's Single Chip Cloud [88, 89] and Sony's Cell Processor [90] completely abandon hardware cache coherence. Such designs could also be used to run microservice environments. However, they are suboptimal. Beyond not supporting multithreaded services efficiently, they also fail to efficiently handle request migration in the presence of frequent context switches. Specifically, recall that a request is frequently blocked on I/O. When the request gets restarted, to better utilize CPUs, the system may want to run it on another core. Unless there is cache coherence support, the state left by the request in the caches before blocking will not be automatically reused after restarting.

Section 3.3.5 showed that handlers executing requests for the same service instance share substantial read-only data and instruction state. *μManycore* takes advantage of this fact, as it maps requests for the same service instance to the same village. Their handlers read the same cache state, thereby improving overall performance.

When a village fills to capacity, the system may need to allocate a new instance of the same service in another village. Such new instance will be initialized faster if it can read a *Snapshot* of the service (Section 3.3.5). A snapshot takes 10s of MBs. Consequently, *μManycore* provides a large *Memory Pool* of fast mostly-read SRAM next to the villages to keep snapshots. Service instances in nearby villages can access the memory pool.

2. Request Scheduling. As indicated in Section 3.3.2, requests come in bursts, potentially creating queues of requests to be processed. As the request execution granularity is often in the scale of microseconds, the overheads of queuing and scheduling are noticeable.

To provide efficient request handling, *μManycore* supports request enqueuing, dequeuing, and scheduling in hardware. Each village has its own hardware queue for requests to local service instances. When a request external to the *μManycore* package arrives at the *μManycore*'s top-level NIC or a request is generated internally in the *μManycore* package, the request is routed in hardware to the village that runs the corresponding service instance and enqueued in a queue. Then, a local core dequeues it. Both enqueuing and dequeuing are performed in hardware, without any OS or other software involvement.

3. Context Switching. A request spends most of its execution time blocked on I/O, waiting on remote storage accesses or calls to other services (Section 3.3.3). Cores avoid

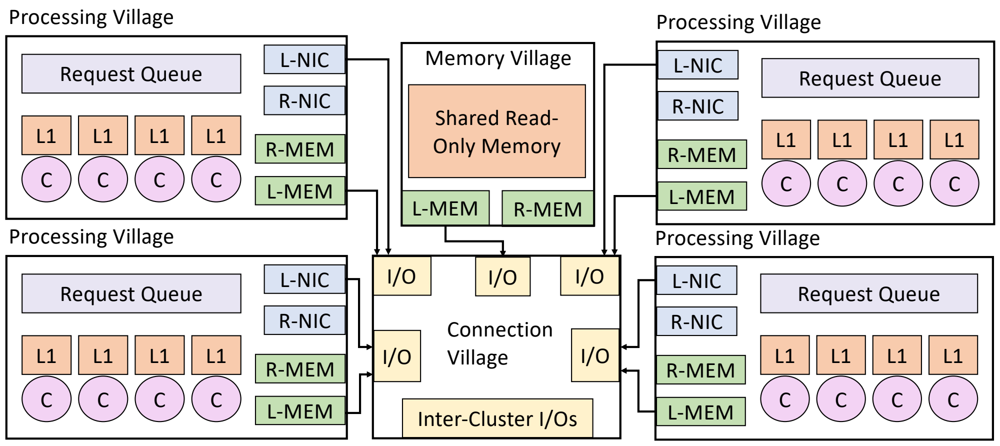


Figure 3.8: Organization of a *μManycore* cluster.

stall time by frequently switching between requests. However, each context switch involves thousands of cycles, directly degrading the tail latency.

To address this problem, *μManycore* has hardware support for context switching. A core saves and restores state in a context switch without any OS or software intervention.

4. On-package Interconnection Network (ICN). Messages between different villages and memory pools traverse ICN links and routers. Network traversal can take substantial time, especially if compounded by contention effects. The resulting latency directly affects the tail latency.

To minimize this latency, *μManycore* uses an on-package *Leaf-Spine* ICN topology [67, 68] (Figure 3.10), which has many redundant, low-hop-count paths between any given source and destination villages. Messages are less likely to suffer contention than in other networks. Even multiple messages with the same source and destination villages can proceed in parallel without delaying one another.

In the following, we describe these four main components of *μManycore* in detail.

3.4.1 *μManycore* Organization

Villages and Clusters. The basic unit of a *μManycore* is a hardware cache-coherent village. A village contains a set of cores (e.g., 8-16) with private caches and a shared L2, a *Request Queue* module that will be described later, and two I/O ports. Since the working set of service requests is small (Section 3.3.5), there is no need for a deeper cache hierarchy.

The combination of a few villages (e.g., 4), a memory pool, and a network hub forms a cluster. Figure 3.8 shows a cluster. We envision the combined villages, the memory pool, and the network hub to be implemented as three different chiplets. Finally, a *μManycore* package is composed of many clusters (Figure 3.9 shows two of them) interconnected with a hierarchical leaf-spine ICN (Figure 3.10).

Communication Modules. In a village, the local (L) I/O port is for communication within the μ*Manycore*, and the remote (R) I/O port is for communication outside the μ*Manycore*. Each port contains a NIC and a hardware module to perform bulk memory transfers (*MEM*)—useful to prefetch or write-back data chunks. The reason why a village has two NICs is that the L-NIC is simpler. The L-NIC runs on a lossless on-package network and, therefore, does not need to support complicated transports (e.g., TCP) for re-transmissions and congestion control. The network has back-pressure support; the source waits for the network to become available before sending messages. There is never the need for retransmission to handle loss or for flow or congestion control. On the other hand, the R-NIC operates on a lossy network when communicating with the external world, and needs to support retransmission, reordering, flow control (to avoid hogging the sender), and congestion control (to avoid saturating the network). It estimates congestion using acknowledgment (ACK) packets (e.g., in TCP or RDMA).

The *Network Hub* (NH) connects to the local and remote ports of all the villages in the cluster. In addition, it is connected to the on-package ICN (via the intra-package port) and to the μ*Manycore*'s top-level NIC (via the inter-package port) to communicate with the outside world (Figure 3.10). The local ports of the villages communicate with the intra-package port; the remote ports of the villages communicate with the inter-package port.

Figure 3.9 shows two clusters with their NHs as leaf switches of the ICN, connected to another NH that acts as a second-level switch. As we will see, groups of non-leaf NHs are placed in chiplets. All the chiplets in clusters, plus the non-leaf NH chiplets, form the processor package. This design can scale up to one thousand cores or more.

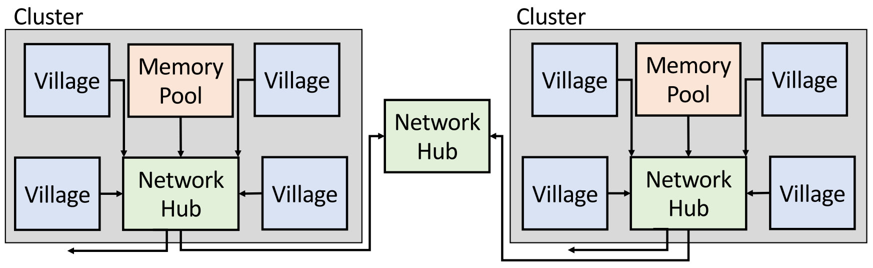


Figure 3.9: Two clusters connected via non-leaf network hub.

Memory Pool. A memory pool (implemented as a separate SRAM chiplet) contains a large volume of fast-access mostly-read data that multiple service instances in the villages of the local cluster may read (Figure 3.8). As indicated before, it contains snapshots of services so that, when a new service instance is created in the cluster, it can fetch the snapshot and skip instance boot-up and initialization overheads. The memory pool also has hardware modules to perform bulk memory transfers to and from on-package memory (*L-MEM*) or to and from off-package memory (*R-MEM*).

Resource Allocation. Each village runs its own light-weight operating system such as a microkernel, and communicates with other villages using messages. A service instance always stays within one village. When the number of concurrent requests for a given instance exceeds the capacity of the village, the system creates another instance of that service in another village. The two instances are independent and serve different arriving requests.

A village can run instances of different services. Then, *μManycore* partitions the cores within the village across colocated service instances based on the instances' load. Each core is assigned a Service ID, which is stored in a separate register. Hence, the system ensures a more predictable performance and minimizes any negative interference between services.

A security-sensitive service instance can exclusively own a village. In this way, we reduce the chances that a malicious program performs side-channel attacks.

3.4.2 Hierarchical Leaf-Spine Interconnection Network

The network hub (NH) in each cluster is a leaf switch of an on-package hierarchical *Leaf-Spine* interconnection network (ICN). The leaf-spine is a topology that provides high connectivity between nodes [67, 68]. The left part of Figure 3.10, inside a dashed box, shows the leaf-spine topology. The per-cluster NHs are the leaf switches inside the box. Each NH is connected all-to-all to a set of fewer, second-level NHs (4 in the figure). These second-level NHs are standalone (i.e., not associated with any cluster), as shown in Figure 3.9. This topology allows any pair of leaf NHs to communicate in two hops, and using as many different paths as there are switches in the second level of the tree.

Since a *μManycore* has many clusters, we build the leaf-spine topology hierarchically. Figure 3.10 shows how the original topology (now called a Pod) is connected to other pods with a third level of NHs. This is the topology used in a *μManycore*. In our 1024-core *μManycore* design, we have 4 pods and 8 third-level NHs. Thanks to this topology's ability to connect many clusters with low hop counts and with redundant paths, this topology minimizes tail latency.

To provide connectivity to the outside world, the leaf NHs are also directly connected to the top-level NIC of the package (Figure 3.10). Incoming external requests flow from the top-level NIC to a leaf NH and then to the remote I/O port of a village. Outgoing requests flow in the opposite direction. The top-level NIC schedules incoming requests to the villages in hardware. Specifically, it maintains a *ServiceMap* table that stores, for each service ID,

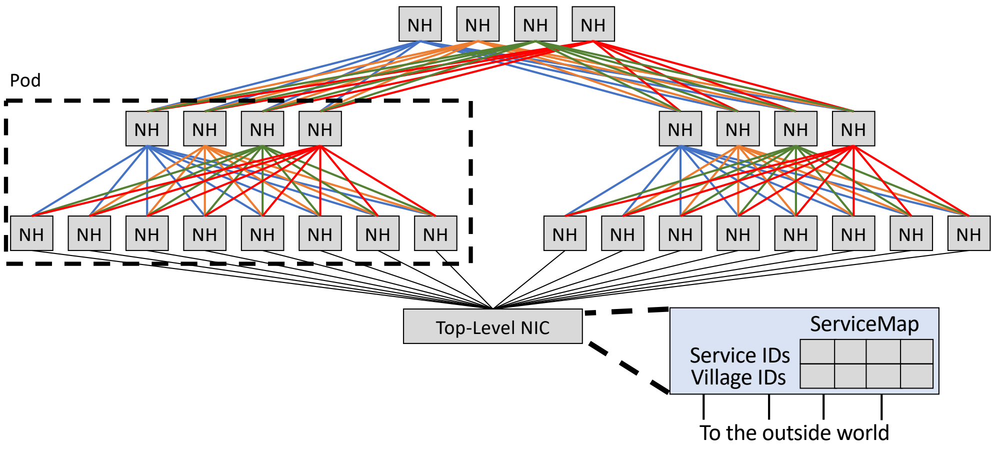


Figure 3.10: Hierarchical leaf-spine interconnection network, where NH stands for network hub. The figure also shows the connection to the package top-level NIC.

the set of villages that host an instance of that service. The ServiceMap is populated by the system software every time a new service instance is initialized in any village. When a request for a given service arrives at the top-level NIC, the hardware checks which villages are able to serve the request. Then, the hardware forwards the request to one of those villages in a round-robin manner.

3.4.3 Hardware Support for Request Queuing and Scheduling

As a village receives requests to execute locally, it is important to minimize the overheads of (i) depositing them on a queue and (ii) picking them up from the queue and executing them on local cores. Minimizing these overheads reduces request tail latency. Consequently, *μManycore* performs these operations in hardware.

To support these operations, each village includes a hardware *Request Queue* (RQ) (Figure 3.11). The RQ is implemented as a circular buffer, with head and tail pointers. Each full RQ entry corresponds to a service request that is executing or wants to execute in the local village. Each RQ entry contains three fields. The first one, *Status*, is the status of the request, which can be: running, ready to run, blocked on an RPC, or finished. The second field, *Service ID*, is the ID of the service that the request invokes. Recall that a village can have instances of multiple services. The third field, *Req Ptr*, is a pointer to a local memory called *Request Context Memory* that contains the context of the request. The context includes: the input data, the destination service that should receive the results of this request's execution, the ID of the process assigned to execute the request, and the core where this request runs (if known).

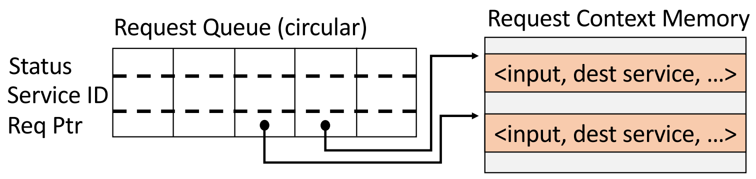


Figure 3.11: Hardware-based Request Queue.

On request arrival, the village NIC hardware performs all the RPC layer processing, such as header parsing, payload de-serialization, and service dispatching [44]. Then, it places the request at the tail of the RQ. Idle cores spin on a per-core local *Work* flag that is automatically set when the RQ contains work to do. When the flag is set, a core executes a *Dequeue* instruction that takes as argument the ID of the service that the core is tasked to execute. Recall from Section 3.4.1 that, when multiple services are co-located in a village, individual cores are assigned to specific services. The Dequeue instruction atomically accesses the RQ and returns the highest-priority entry (i.e., the one closest to the RQ head) that matches the service ID and is ready to run. It also sets the entry's status to running.

After a core completes the execution of a request, it executes a *Complete* instruction, passing as argument a pointer to the RQ entry. The hardware atomically accesses the RQ, sets the request status to finished and, if the entry was at the RQ head, advances the head to the first unfinished entry. With this hardware, *μManycore* minimizes the tail latency effects of request queuing and scheduling. Moreover, by processing requests in FCFS order, this scheme further minimizes tail latency.

An alternative scheduling policy to use is Shortest Remaining Processing Time First (SRPT). However, in microservice environments, SRPT is unlikely to improve much over FCFS for two reasons. First, requests for a given service tend to have similar execution times. Second, request execution is frequently interrupted by I/O calls, which in our case will provide frequent opportunities to schedule other ready-to-run requests.

If a request finds a full RQ, it is temporarily queued in the NIC. If the NIC has exhausted its buffering space, it rejects the request.

A more advanced design of the RQ would involve dynamically partitioning it into multiple RQs—each partition devoted to a different service. Specifically, when the system co-locates a second service instance in a village, the system would partition the RQ and record the new RQ structure in an RQ_Map hardware table. The proportion of entries assigned to each service can be the same as the proportion of cores assigned to each service. Since each core maintains a register with the ID of the service it is assigned to execute and passes it as an argument to the Dequeue instruction, all that is needed is to augment the Dequeue instruction to check the RQ_Map first. This additional hardware would eliminate contention of different-service cores for the same RQ, likely reducing the tail latency further. We do not consider this design in the evaluation.

3.4.4 Hardware Support for Context Switching

State-of-the-art schemes for efficient scheduling of microservice workloads [47, 48, 91, 92, 93] use a "run-to-completion" model: a core is assigned to execute a request until it completes. At best, the process is pre-empted if it runs for very long, to prevent head-of-line blocking [48]. In practice, as shown in Section 3.3.3, the process executing a request is blocked most of the time, due to issuing storage accesses or calling other services. In the meantime, the cores context switch and execute other requests. The frequent context switching induces overhead and expands tail latency.

To assess this overhead, we run the highly-optimized Shinjuku software scheduler [48] in our simulated 1024-core ScaleOut manycore (Section 3.5). Shinjuku needs to (1) run on a dedicated core, (2) detect when a process on a core blocks, (3) save the context of the blocked process, (4) find a ready request by checking the RQ, and (5) restore the context of the ready request from memory. We find that this centralized software easily becomes a bottleneck and limits the overall throughput. It consumes time and, as shown in Figure 3.4, results in a high tail latency.

To address this problem, *μManycore* adds hardware support to reduce the overhead of context switching. The idea is that, when the execution of a request blocks, special hardware in the core saves the process state to memory. Then, the core is ready to access the RQ to get a new request. Also, when a core obtains from the RQ a request that had partially executed in the past, the hardware restores from memory the state of the request. The state saved and restored includes general-purpose and special-purpose registers; interrupt, exception, debugging, and privilege level information; and cached storage descriptors [94, 95]. The state is a few hundreds of bytes.

To support this design, the entry for a request in the Request Context Memory (Figure 3.11) is expanded to include space for the saved process state. Further, when a process executing on a core issues an RPC and is about to get blocked, the core executes a new *ContextSwitch* instruction. In hardware, this instruction saves the process state in the corresponding entry of the Request Context Memory, and sets the Status field in the RQ to blocked. The core is now free to spin on the *Work* flag to see if there is work to do.

When the NIC receives the RPC response, it puts the response in the Request Context Memory entry of the corresponding request, and then changes the Status field of the RQ entry from blocked to ready to run. At this point, an idle core will see a set *Work* flag and execute a Dequeue instruction. *μManycore* augments the Dequeue instruction to also upload the state of the selected request from the Request Context Memory to the core registers. The other functionality of Dequeue is unchanged.

Overall, with this support, cores keep context-switching overheads to a minimum, and can perform useful work practically all the time—effectively reducing tail latency.

3.5 METHODOLOGY

We model a *μManycore* package with 1024 cores organized into 32 clusters. Each cluster has 4 villages of 8 cores each, one memory pool, and a network hub (NH). Each village has a 64-entry request queue (RQ). *μManycore* uses a leaf-spine interconnect with a three-level hierarchy. There are 32 leaf-level NHs organized into 4 chiplets. For the second level, there are 4 groups of 4 NHs, organized into 4 chiplets. Each second-level NH in a group connects to all 8 NHs of the first level. For the third level, there are 8 NHs in two chiplets, where each third-level NH is connected to all 16 second-level NHs. The longest communication path is only 4 hops. Overall, a *μManycore* package has 32 clusters, 128 villages, 32 memory pools, and 56 NHs, for a total of 74 chiplets.

μManycore has simple, energy-efficient cores similar to ARM A15 [96]. They are 4-issue and run at 2GHz. They have a small ROB (64 entries) and LSQ (64 entries), private L1 caches, a single-level TLB, and a shared L2 cache.

We model two baseline hardware-coherent processors: the *ServerClass* multicore and the *ScaleOut* manycore. *ServerClass* is a beefy server-class processor, similar to Intel's IceLake [97]. Its cores are 6-issue and run at 3GHz. They have a large ROB (352 entries) and LSQ (256 entries), private L1 and L2 caches, two levels of TLBs, and a shared L3 cache. For comparison to *μManycore*, we evaluate two sizes of *ServerClass* processors: one with 40 cores that consumes the same power as *μManycore*, and one with 128 cores that has the same area as *μManycore*. The former is like a current high-end IceLake; the latter is an unrealistically power-hungry multicore.

ScaleOut is a 1024-core manycore organized into 32 clusters. *ScaleOut* uses the same cores and cache hierarchy as *μManycore*, including L2 caches shared by 8 cores. *ScaleOut* does not include the *μManycore* novelties: no global cache coherence, leaf-spine ICN, hardware support for request queuing and scheduling, and hardware support for context switching.

ServerClass and *ScaleOut* use conventional ICNs, namely a mesh and a fat-tree, respectively. For comparison to *μManycore*, the fat-tree topology has 63 NHs and its longest path is 10 hops. Both baselines use an optimized state-of-the-art software-based context-switching scheme [48] and techniques that reduce NIC-to-core communication overheads [43, 46].

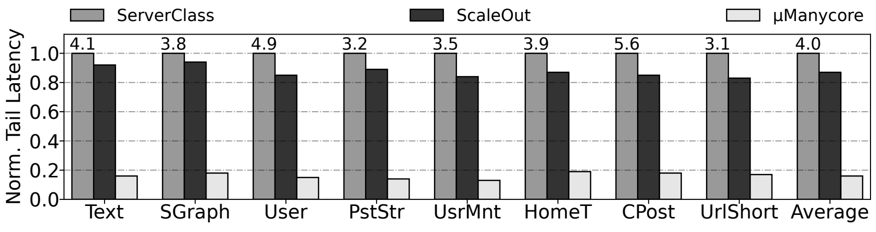
We model 10-server machines with each of the three types of processors. Table 3.2 shows the parameters of the architectures. To model these machines, we use the SST architectural simulator [98] connected to the DRAMSim2 memory simulator [99]. We use Pin [100] to collect traces and feed them to the SST simulator.

To compute the area and power consumed by each of the processors, we use CACTI [101] for the memory structures and McPAT [102] for the cores. We use the 32nm technology available with the tools, and then scale to 10nm technology [103]. The combined dynamic and static power consumed by one core and its portion of the cache hierarchy is: 10.225W for *ServerClass*, 0.396W for *ScaleOut*, and 0.408W for *μManycore*.

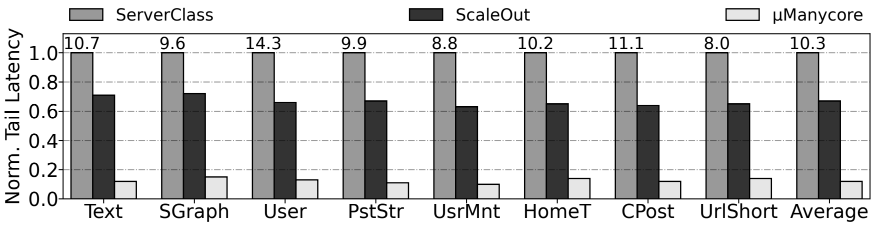
Table 3.2: Architectural parameters used in the evaluation.

ServerClass Multicore	
Multicore	40 (or 128) 6-issue cores, 352-entry ROB, 256-entry LSQ, 3GHz
L1 cache	64KB, 8-way, 2 cycles round trip (RT), 64B line
L2 cache	2MB, 16-way, 16 cycles RT, 20 MSHRs
L3 cache	2MB/core, 16-way, 40 cycles RT, 20 MSHRs
L1 DTLB	256 entries, 4-way, 2 cycles RT
L2 DTLB	2048 entries, 12-way, 12 cycles RT
Network	2D mesh
μManycore and *ScaleOut* Manycores	
Manycore	1024 4-issue cores, 64-entry ROB, 64-entry LSQ, 2GHz
L1 cache	64KB, 8-way, 2 cycles RT, 64B line
L2 cache	256KB, 16-way, 24 cycles RT, 20 MSHRs
L1 DTLB	128 entries, 4-way, 2 cycles RT
Network	Fat tree (*ScaleOut*), leaf-spine (*μManycore*)
Network	
Intra server	5 cycles/hop (4 router delay + 1 wire delay) [104]
Inter server	1μs RT; 200GB/s
Main-memory per Server	
Capacity	80GB
Channels; Banks	4; 8
Frequency; Rate	1GHz; DDR
Mem bandwidth	8 memory controllers; 102.4GB/s per controller

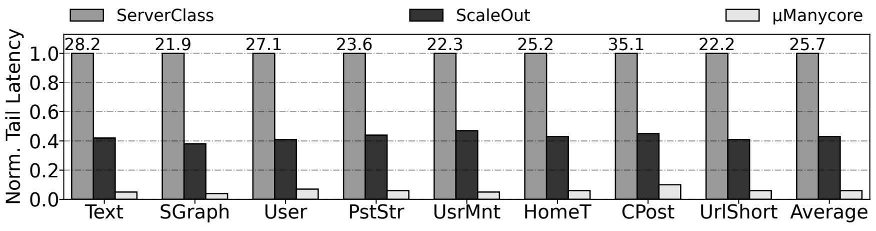
Applications. We use applications from the open-source DeathStarBench [40] microservice benchmark suite with commit ID c86920a. Due to space limitations, we show the results for only the 8 Social Network applications. The results are similar for the other applications of the benchmark suite. We use Poisson distributions for the request inter-arrival time. We use average loads of 5K, 10K and 15K requests per second (RPS) per server, which correspond to average CPU utilizations of <30%, 30-60%, and >60%, respectively. We collect the tail and average response time and throughput for each application.



(a) Load of 5K RPS.



(b) Load of 10K RPS.



(c) Load of 15K RPS.

Figure 3.12: Tail latency in *ServerClass*, *ScaleOut*, and *μManycore* normalized to *ServerClass*. The numbers on top of the *ServerClass* bars are the absolute latency values in ms.

3.6 EVALUATION

In this evaluation, response time (i.e., latency) is measured end-to-end, from when the client sends a request to when it receives the result. We give both the average and the P99 (i.e., 99^{th} percentile) values. Unless otherwise indicated, *ServerClass* has 40 cores and, therefore, consumes approximately the same power as *μManycore*.

3.6.1 End-to-End Tail Latency

Figure 3.12 shows the tail latency in the three architectures when running the DeathStarBench applications, normalized to *ServerClass*. On top of the *ServerClass* bars, we show the absolute latency values in ms. The systems are tested with three load levels: (a) 5K, (b) 10K, and (c) 15K RPS. We see that *μManycore* significantly reduces the tail latency for all applications across all loads. On average, *μManycore* reduces the tail latency over *ServerClass* by 6.3×, 8.3×, and 16.7× for loads of 5K, 10K and 15K RPS, respectively, and over *ScaleOut* by 5.4×, 6.5×, and 7.4× for the same loads.

μManycore achieves greater reductions with higher system loads, especially for the applications that are blocked more frequently, i.e., the ones that invoke a larger number of downstream services such as SocialGraph service (SGraph). In these cases, the impact of the *μManycore* techniques is more notable.

3.6.2 Tail-Latency Reduction Breakdown

Figure 3.13 shows the contributions of the four main *μManycore* techniques to the reduction of tail latency for 15K RPS. Latency reductions are normalized to the latency of *ScaleOut*. We apply the four techniques one by one in order: villages (Section 3.4.1), leaf-spine topology (Section 3.4.2), hardware scheduling (Section 3.4.3), and hardware context switching (Section 3.4.4). On average, the cummulative application of these techniques reduces the tail latency by 1.1×, 2.3×, 3.9×, and 7.4×, respectively. All techniques deliver major reductions except for the village organization, which reduces the tail latency by a modest 10%. The reason for this modest reduction is that we have favored the *ScaleOut* baseline. Specifically, while *ScaleOut* uses global cache coherence, it has one queue per 32-core cluster, and only allows processes to migrate between the 32 cores of a cluster. If we allowed processes to migrate between all 1024 cores, *ScaleOut* would perform worse. In any case, the main attractive of villages is not higher performance, but a reduction in manycore area, power, and complexity by eliminating global cache coherence—potentially allowing hardware resources to be used for other goals.

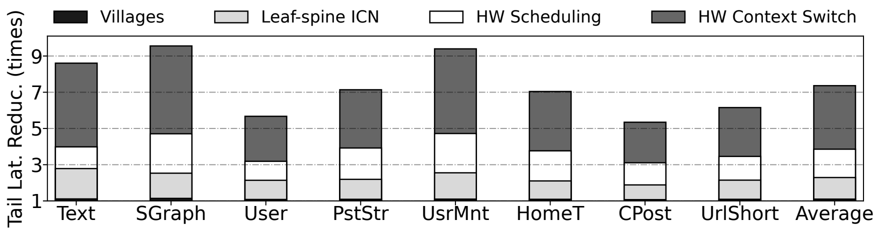
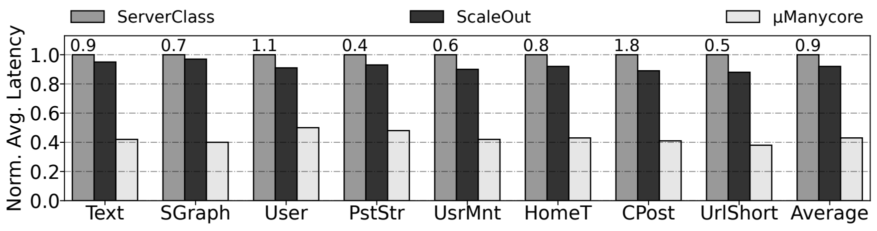
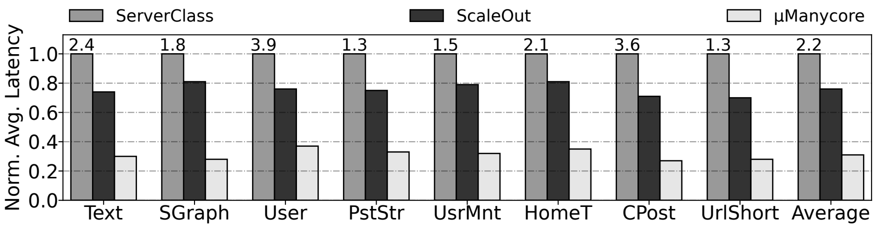


Figure 3.13: Contributions of the four main *μManycore* techniques to the reduction of tail latency for 15K RPS. Latency reductions are normalized to the tail latency of *ScaleOut*.

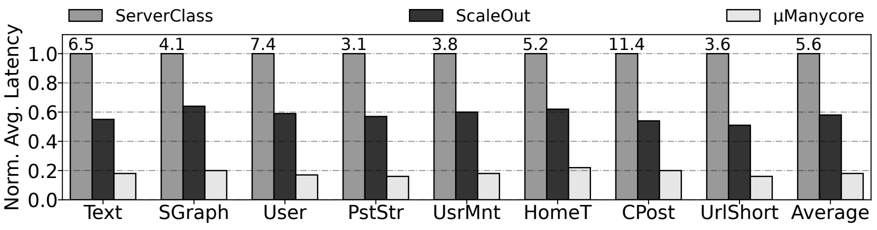
In applications that frequently use the ICN, e.g., Text and SGraph, the leaf-spine ICN is very effective. These same applications also substantially benefit from the hardware scheduling and context switching techniques of *μManycore*. They often have requests stalled, waiting for remote storage accesses or service calls, and *μManycore* mitigates these overheads.



(a) Load of 5K RPS.



(b) Load of 10K RPS.



(c) Load of 15K RPS.

Figure 3.14: Average latency in *ServerClass*, *ScaleOut*, and *µManycore* normalized to *ServerClass*. The numbers on top of the *ServerClass* bars are the absolute latency values in ms.

3.6.3 End-to-End Average Latency

Figure 3.14 shows the normalized *average* latency in the three designs and for the three load levels. The numbers on top of the *ServerClass* bars are the absolute latency values in ms. *µManycore* reduces the average latency for all applications across all loads. The average latency reductions are smaller than the tail latency reductions in Figure 3.12. This is because the *µManycore* design is more tailored to minimizing the tail latency, by removing the major sources of contention and interference. On average, *µManycore* reduces the average latency over *ServerClass* by 2.3×, 3.2×, and 5.6× for loads of 5K, 10K, and 15K RPS, respectively, and over *ScaleOut* by 2.1×, 2.5×, and 3.2× for the same loads.

3.6.4 Reduction in Tail-to-Average Ratio

The goal of *µManycore* is to minimize the tail latency, but also to bring it closer to the average. In this way, the response time becomes more predictable, and more users can be served within the guaranteed QoS. Figure 3.15 shows the normalized tail-to-average latency ratio per application for the three designs averaged across all the loads. The numbers on top of the *ServerClass* bars are the absolute ratios.

In *μManycore*, the tail to average latency ratio is significantly smaller than in the other architectures. On average, it is 2.7× and 2.3× lower than in the *ServerClass* and *ScaleOut* baselines, respectively. In *UsrMnt*, the tail to average latency ratio in *μManycore* is 3.3× lower than in *ServerClass*.

In the baselines, the ratio between the tail and the average is especially notable under high loads. Some requests are served quickly, but the slowest ones are substantially slowed down due to queuing and contention. In such environments, *μManycore* reduces the tail latency while keeping the average latency low, thus shrinking the ratio between tail and average.

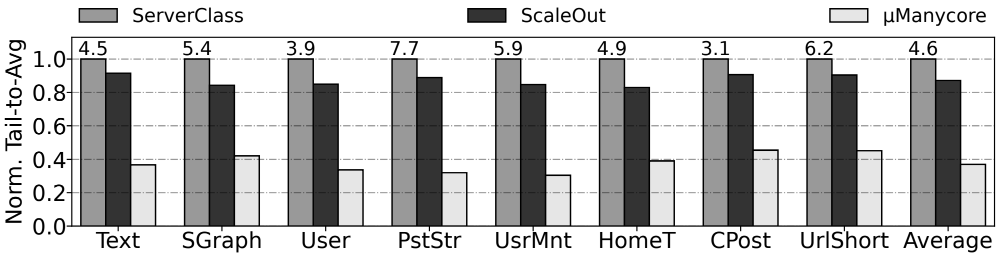


Figure 3.15: Tail-to-average latency ratio of *ServerClass*, *ScaleOut*, and *μManycore* normalized to *ServerClass*. The numbers on top of the *ServerClass* bars are the absolute ratios.

3.6.5 End-to-End Throughput Improvements

We measure the number of requests that can be served by the system (i.e, the throughput) without violating QoS guarantees. A QoS violation occurs if the request execution time is higher than 5 times the contention-free average request execution time. Figure 3.16 shows the normalized maximum throughput that each of the three designs can achieve without violating QoS guarantees. The numbers on top of the *μManycore* bars are the absolute throughput values that *μManycore* achieves in KRPS. *μManycore* reaches a throughput that is 13.9–17.1× higher than the *ServerClass*. On average, *μManycore* improves the throughput by 15.5× and 4.3× over the *ServerClass* and *ScaleOut* baselines, respectively.

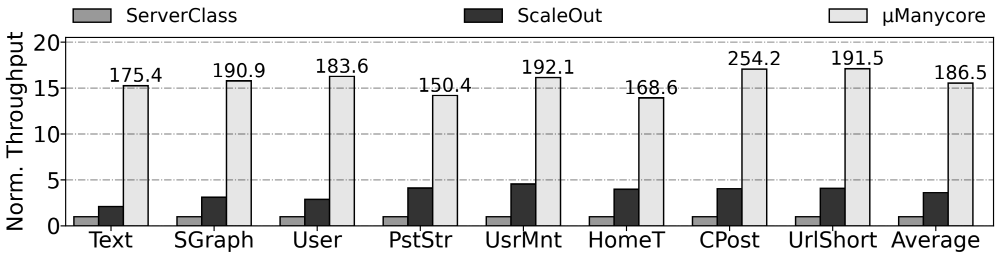


Figure 3.16: Normalized maximum throughput a system can achieve without violating QoS guarantees. The numbers on top of the *μManycore* bars are the absolute throughput values that *μManycore* achieves.

3.6.6 Sensitivity Analysis

μManycore can be organized with different configurations of number of cores per village, number of villages per cluster, and number of clusters. Figure 3.17 shows the tail latency in four configurations. The bars are normalized to the tail latency of the first configuration, which is the default configuration used in all the previous experiments.

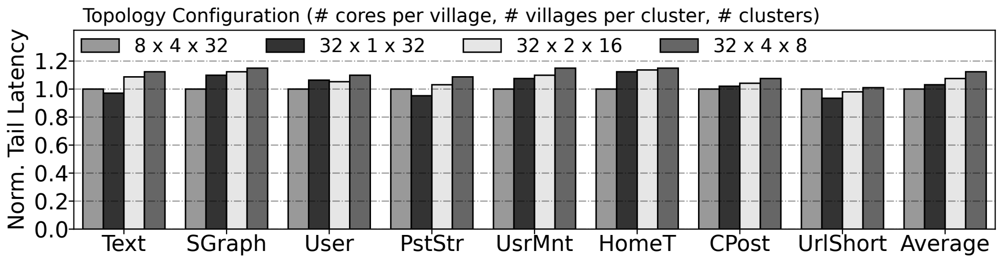


Figure 3.17: Normalized tail latency with different *μManycore* configurations.

All configurations are within 15% of each other's tail latency. Interestingly, different services are best suited to different configurations. Specifically, services that do not call other services such as *UrlShort* perform better in larger villages (32x1x32). On the other hand, services that frequently invoke other services such as *HomeT* and *SGraph* have better performance with many smaller villages (8x4x32). Overall, our default configuration has the overall lowest tail latency.

3.6.7 Comparison to an Iso-Area ServerClass CPU

The evaluation so far has used iso-power configurations for *ServerClass*, *ScaleOut*, and *μManycore*. In this section we compare iso-area configurations. As indicated in Section 3.5, we use CACTI [101] and McPAT [102] for our computations. In the iso-power configurations, *μManycore* has 2.9% more area than *ScaleOut* and 3.1× more area than the 40-core *ServerClass* (i.e., 547.2mm^2 for *μManycore* versus 176.1mm^2 for *ServerClass*). Hence, for an iso-area comparison, we keep *μManycore* and *ScaleOut* unchanged and we scale *ServerClass* to 128 cores, while leaving all the other parameters unmodified. The new *ServerClass* processor improves the performance significantly, matching and sometimes slightly outperforming the tail latency of *ScaleOut*. However, *ServerClass* still has a tail latency that is on average 7.3× higher than the *μManycore* one across all loads and applications. Also, the 128-core *ServerClass* processor uses an unacceptably large amount of power, namely 3.2× more than *μManycore*.

3.7 RELATED WORK

Software schedulers. Recently, researchers have explored various software solutions for μ-second scale scheduling and context switching [47, 48, 91, 93, 105, 106, 107, 108, 109, 110, 111]. IX [93] schedules a batch of requests at high throughput, but it degrades the tail latency of heavy-tailed service time distributions due to using a per-core distributed scheduler. ZygOS [91] minimizes head-of-line blocking and load imbalance by allowing cores to steal requests from other cores via expensive software operations. It can potentially degrade the tail latency of short requests. Shinjuku [48] uses a centralized scheduler with request preemption to tolerate different service time distributions. It cannot scale to a large number of cores, and may degrade the tail latency due to the cost of software context switches. Shenango [47] dedicates a core to perform scheduling, possibly limiting its throughput and scalability. *μManycore* performs scheduling and context switching in hardware, thus reducing the overheads and increasing the throughput over such software schemes.

Message passing operating systems (OSes). Fos [112] and Barrelfish [113] are distributed OSes: a core runs a local OS and communicates with the OS of other cores only via message passing. There is no cache coherence. However, as per Section 3.4.1, this architecture is not well-suited for microservices. It is possible to use some ideas from fos and Barrelfish to support the communication between the shared-memory OSes running on each village.

RPC accelerators. Researchers proposed hardware accelerators to improve the efficiency of the RPC-based communication [43, 44, 45, 46, 92, 114, 115, 116]. RPCValet [45] uses the on-chip NICs to monitor per-core load and to steer RPCs to lightly-loaded cores. Nebula [43] provides hardware support for efficient in-LLC network buffer management, and sends incoming RPCs into the CPU cores' L1 caches. The nanoPU [46] bypasses the cache and memory hierarchy and places the arriving messages directly into the CPU register file. Cerebros [44] executes all RPC layers in hardware without involving the CPU. While *μManycore* has been inspired by these systems, none of them considers services that invoke other services and are waiting idly for long durations. Therefore, they execute in the run-to-completion manner and do not focus on efficient support for context switching, as in *μManycore*.

Duplexity [51] is a processor architecture that, when there are not enough latency-critical jobs (e.g., microservices) to run, it reconfigures and uses the idle resources to run batch workloads. *μManycore* could add this approach to increase core utilization in low loads.

Packet processing's execution model, such as supported by the Event Machine [117] can in principle be applied to process microservice invocations. In packet processing, arriving packets are queued up in multiple queues. Then, the system dequeues packets with some notion of priority, and sends them to execute on available cores. In *µManycore*, service invocations are queued and dequeued in hardware. A key characteristic of microservice processing is that service invocations frequently stall on I/O. In addition, some individual service invocations may execute in a multithreaded manner.

Hardware queuing. Hardware queues [118, 119, 120] have been proposed to support low latency communication between producer and consumer threads running on different cores. Existing proposals target traditional task-parallel systems and bind the queue state to an application's context. In *µManycore*, queues are not per application, but contain entries for different service requests. Hence, entries are not saved and restored on a context switch. The producer is a NIC and consumers are cores in the village. ALTOCUMULUS [115] proposes a scheme for RPC scheduling in hardware. It does not consider that requests are idle for most of the time due to blocking calls. The scheduler in *µManycore* takes into account blocked requests and only schedules those that are runnable.

Chiplet-based processor designs. Recently, both industry and academia have shown great interest in chiplet-based processor designs [121, 122, 123, 124, 125, 126, 127]. These designs improve yield, allow the integration of heterogeneous components, and simplify processor design. In some designs, processors are grouped in clusters or core complexes [121]. One way in which *µManycore* goes beyond these designs is that, in *µManycore*, cache coherence is only supported inside these clusters (called villages), not across them.

3.8 FURTHER ENHANCEMENTS

µManycore can be enhanced in a variety of ways to improve the performance of microservice environments. These enhancements add additional costs.

In *µManycore*, when different service instances are co-located in the same village, *µManycore* apportions the cores to the different service instances based on the expected load. It is possible that, as requests arrive, the distribution of load across services is different than expected—e.g., the cores of one of the instances are mostly idle while those of the other are unable to keep up with the requests. In this case, an enhancement to *µManycore* would be to allow an instance to temporarily steal cores assigned to another instance.

In *µManycore*, all villages have the same hardware. It is, therefore, a homogeneous architecture. A possible enhancement is to have different hardware in different villages. For example, some villages might have bigger cores. This approach would enable the assignment of different types of services to different types of villages—hence tailoring the hardware to the needs of the service instances. However, it is unclear what different types of villages and how many of each are needed. Moreover, services would likely have to be instrumented to declare what type of village they would prefer.

3.9 CONCLUSION

To address the imbalance between emerging microservice environments and current processors, this chapter proposed μ*Manycore*, an architecture optimized for microservices. Based on a characterization of microservices, μ*Manycore* is designed to minimize unnecessary microarchitecture and reduce tail latency. Instead of supporting manycore-wide hardware cache coherence, μ*Manycore* has multiple hardware cache-coherent smaller domains called villages. Clusters of villages are interconnected with a leaf-spine network, which has many redundant, low-hop-count paths between clusters. To minimize overheads, μ*Manycore* schedules and queues service requests in hardware, and saves/restores process state in a context-switch in hardware. Our simulation-based results showed that μ*Manycore* delivers high performance for microservices. A cluster of 10 servers with a 1024-core μ*Manycore* in each server delivered 3.7$\times$ lower average latency, 15.5$\times$ higher throughput, and 10.4$\times$ lower tail latency than a cluster with iso-power conventional server-class multicores. Similar results were attained compared to a cluster with power-hungry iso-area conventional server-class multicores.

CHAPTER 4: Hardware Support for Core Harvesting of Cloud-Native Services

4.1 INTRODUCTION

Microservice *instances* are Virtual Machines (VMs) or containers that serve microservice invocations (i.e., *requests*). An instance is created with a specified number of cores and amount of memory, and serves requests for that microservice. Although requests are short-running (typically, hundreds of μseconds), instances are long-lived: they can be up and serve requests for days [13].

The frequency of request arrival for a given microservice varies substantially with time, and exhibits bursty patterns. To attain good performance even at peak loads, users typically provision instances to handle these infrequent load spikes. As a consequence, instances are typically greatly *overprovisioned*—e.g., in number of cores needed. The result is that, in microservice environments, *allocated but idle* cores are a major waste [5]. As an example, using open-source production-level microservice traces from Alibaba [5], we see that 50% and 90% of microservice instances have an average core utilization lower than 16.1% and 40.7%, respectively.

To combat resource inefficiency under *general loads*, providers have implemented various techniques in their software stacks [128, 129, 130, 131, 132]. Amazon allows a Spot VM to seize unallocated cores if needed [131]. Further, Microsoft introduced a new type of VM called *Harvest* VM [133, 134, 135, 136] that can dynamically grow by harvesting cores. In such environments, there are two types of VMs: Primary and Harvest VMs. *Primary* VMs run latency-critical applications, need predictable high performance, and are created with a specified number of cores; *Harvest* VMs run batch applications, have loose performance requirements, can tolerate resource fluctuations, and are charged at a lower cost. Harvest VMs dynamically grow by harvesting unallocated cores in the server [133] or, additionally, by taking temporarily idle cores allocated by a Primary VM [134]. When the Primary VM needs its cores, it reclaims them back.

In practice, re-assigning a core from one VM to another has high overhead. First, a scheduler must perform two hypervisor calls: one to detach the core from the first VM, and the other to attach it to the second VM. Second, an expensive cross-VM context switch is performed. In addition, the state left in the re-assigned core's private caches and TLBs is a potential source of leakage. Specifically, a malicious tenant could force conflicts in these structures and learn information from an earlier tenant such as cryptographic keys [137]. Hence, the core's private caches and TLBs are typically flushed and invalidated during the reassignment [137, 138, 139, 140, 141, 142, 143, 144]. Unfortunately, flushing and invalidating private caches and TLBs, and the resulting cold-cache and cold-TLB restart add substantial overhead. Overall, we find that the sum of all these overheads can easily exceed 5ms. These overheads are particularly insidious when a core is reclaimed by its owner Primary VM, as they directly impact the response time of latency-critical applications.

The Harvest VM concept has been applied only to setups where Primary VMs run relatively long monolithic applications [133, 134, 135, 136]. In such applications, a reassigment overhead of a few ms can be considered negligible. However, in microservice environments, it is not tolerable to suffer a few-ms reassignment overhead every time that a 100-μs microservice request is received and needs to execute. The situation is even more challenging in an aggressive environment that can re-assign a Primary VM idle core not just when the core has terminated a request execution, but also when the execution is stalled on I/O—as such events happen frequently.

To address this problem, this chapter proposes the first architecture that supports core harvesting *in hardware*, called *HardHarvest*. The goal is three-fold: attain high core utilization, introduce minimal or no increase in the tail latency of Primary VM microservice requests, and deliver substantial increases in the throughput of batch workloads in Harvest VMs. To accomplish these goals, HardHarvest targets the two main overheads present in software-based core harvesting.

The first overhead is the core re-assignment. To minimize it, HardHarvest adds hardware queues for microservice requests. A microservice request arrives as a network packet that contains the name of the microservice function to invoke and the input data required by that function. The message payload is deposited into the LLC and a pointer to the payload is stored in a hardware request queue. A core is re-assigned from one VM to another by being allowed to dequeue requests from the new VM's hardware queue when the original queue is empty. There is no need for detach/attach system calls and the context switch is accelerated in hardware.

The second overhead is flushing and invalidating TLBs and private caches on core reassignment, and the resulting cold restart. HardHarvest leverages the fact that microservices typically have small working sets, and partitions TLBs and private caches into two regions: *Harvest* and *Non-Harvest* regions. When a core executes a Primary VM, it can use both regions; when it executes a Harvest VM, it is allowed to use only the Harvest region. When a core transitions between VMs, only the Harvest region is flushed and invalidated; the Non-Harvest region preserves the Primary VM's state during harvesting. In addition, HardHarvest enhances the effectiveness of such partitioning with a smart cache/TLB replacement algorithm that retains important state in the Non-Harvest region.

We evaluate HardHarvest with full-system simulations of an 8-server cluster, where each server has a 36-core IceLake-like processor [97]. Our evaluation shows that HardHarvest is very effective. On average, compared to state-of-the-art software core harvesting, HardHarvest increases core utilization by 1.5× and Harvest VM throughput by 1.8×, while reducing Primary VM tail latency by 6.0×. Compared to a system without core harvesting, HardHarvest increases core utilization by 3.4× and Harvest VM throughput by 3.1×, without increasing the tail latency of Primary VMs.

This chapter's contributions are as follows:

• A characterization of the opportunities of supporting hardware-based core harvesting in microservice environments.

• HardHarvest, the first architecture for hardware core harvesting.

• An evaluation of HardHarvest for microservice environments, comparing it to state-of-the-art core harvesting and no harvesting.

4.2 BACKGROUND

Resource Harvesting in the Cloud. Traditionally, a good way to improve server utilization in datacenters has been to co-locate batch workloads with user-facing applications [145, 146]. Batch workloads, such as machine learning training [147] or in-background data processing [136], can then use resources left idle by user-facing applications to improve their throughput. Recently, the same approach has been used for VMs in the cloud, where software support allows VMs running batch applications to harvest cores [133, 134, 136], memory [135], and storage [148].

Core harvesting has received the most attention, and has been explored through a new type of VM called *Harvest VM* [133, 134, 136]. Harvest VMs dynamically change their size by temporarily stealing idle cores. In the initial designs, Harvest VMs could steal only cores that were not allocated to any latency-critical VM (i.e., *Primary* VM), and had to return the cores once a new Primary VM was created [133]. State-of-the-art harvesting schemes, such as SmartHarvest [134], further improve core utilization by temporarily stealing idle cores allocated to Primary VMs. The system monitors the core utilization of all Primary VMs in the server. Then, based on the predictions of core utilization in the near future, the system may decide to re-assign some cores to the Harvest VM. When a Primary VM needs the cores back, SmartHarvest returns the borrowed cores. Since current approaches to re-assign a core from one VM to another involve substantial software overheads, SmartHarvest keeps a few idle cores on stand-by in an emergency buffer. If needed, these cores can be reclaimed by Primary VMs.

The Harvest VM concept has been applied only to environments with long-running monolithic applications in Primary VMs [133, 134, 135, 136]. For these applications, the overhead of core re-assignment may be tolerable. For microservices, however, which typically run for hundreds of μseconds, we will see that the core re-assignment overhead is too high. The situation is even more challenging in a microservice environment where idle Primary VM cores can be stolen not just after a core has terminated the execution of a microservice request, but also when a core has stalled execution due to I/O—which is frequent.

Microarchitecture Structure Flush and Invalidation on Core Re-Assignment. In multi-tenant clouds, when a core is reassigned from one VM to another, the state left in the core's private microarchitectural structures such as caches and TLBs is a potential source of information leakage. The new VM being scheduled could observe private state that the old VM being preempted has left in these structures. Hence, structures such as private caches and TLBs are flushed and invalidated when a core switches from one VM to another. This is both described in research papers (e.g., [138, 139, 140, 141, 142, 143]) and documented in literature from cloud providers. For example, a 2024 blog from Microsoft production [144] and a paper from Microsoft Research [137] say that they use microarchitecture state flushing and scrubbing in their production systems when a core moves from one VM to another.

Following these ideas, in this chapter: 1) we partition the shared last-level cache into one partition per VM and 2) we require that, in our baseline design, on a core context switch from one VM to another, all the levels of private caches and TLBs in the core are flushed and invalidated.

4.3 MOTIVATION FOR IN-HARDWARE CORE HARVESTING

To understand the opportunities of hardware-supported core harvesting for microservices, we analyze a large set of applications: production-level traces of Alibaba's microservices [5], and DeathStarBench [40] microservices (acting as latency-sensitive workloads running in Primary VMs). The traces provide a time series of average, maximum, and minimum core utilization of microservice instances. We run the microservices on an Intel IceLake server with 36 2.4GHz cores, 256GB of DRAM, and an LLC with 1.5MB per core. The server runs Ubuntu 22.04 and the KVM hypervisor. To generalize the insights, we also run the microservices as Docker containers and observe similar results. We identify several opportunities for hardware-based core harvesting.

Opportunity: **Cores allocated to microservice instances in the cloud are heavily underutilized.** When a user deploys a microservice instance in the cloud, they specify the resources needed for the instance, including the number of cores [149, 150, 151]. The instance is typically sized to guarantee it can handle the peak load, leading to low average utilization [5, 83, 134, 152]. Figure 4.1 shows the distribution of the average and maximum core utilization of Alibaba's microservice instances. The utilization is low: half of the instances have an average core utilization lower than 16.1%, and 90% of instances have a maximum core utilization lower than 40.7%. These numbers represent great opportunities for harvesting.

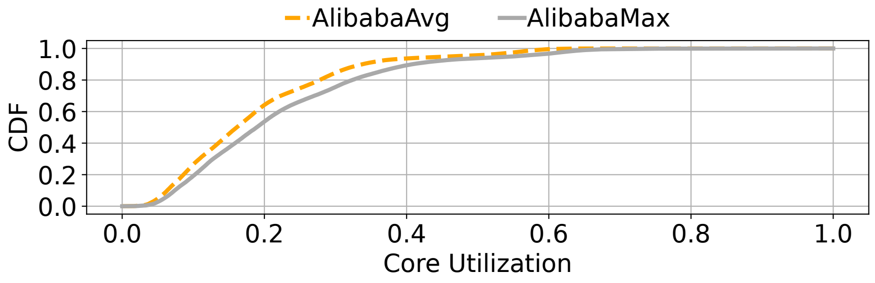


Figure 4.1: Core utilization of Alibaba's microservice instances.

Opportunity: **If the overhead of hardware core harvesting is low, significant performance gains can be attained.** To understand this opportunity, we make three observations.

• **There are large fluctuations in a VM's core utilization or load over time.** We analyze Alibaba's traces, which provide a granularity of 30-second measurements. Figure 4.2 shows the core utilization of a representative Alibaba VM over time.

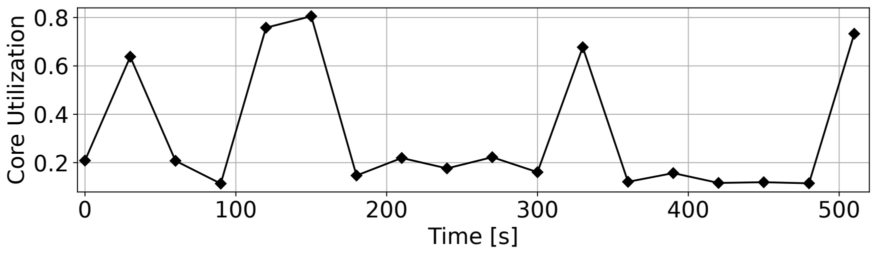


Figure 4.2: Core utilization of an Alibaba microservice.

While core utilization is often low, we observe that it can suddenly increase due to bursts of requests. Let us assume that these VMs are Primary ones. During the low-load periods, some of their cores can be harvested by Harvest VMs. Then, when the load spikes, the harvested cores must be quickly returned to Primary VMs to prevent increases in the tail latency of Primary VM requests.

In current systems, there are two main sources of overhead to move a core between VMs: 1) software overhead of invoking the hypervisor to reassign the core from one VM to another, and 2) flushing and invalidating the caches of the reassigned core (so that no cache state leaks across a VM transition) and subsequent cold-cache re-start of the execution.

• **Core re-assignment via hypervisor is costly.** In state-of-the-art software-based harvesting [134], a user-space agent monitors the utilization of all the cores in Primary VMs and, based on the utilization, may decide to migrate a core to a Harvest VM. To do so, it performs a hypervisor call to detach the core from the first VM and another to attach it to the second VM, using `cgroup` tools. With the KVM hypervisor, moving a core across VMs takes ≈5ms. Half of this time is spent on detaching/attaching the core, and half on loading the new VM's context. The state-of-the-art SmartHarvest design [134] reduces the cost of detaching/attaching the core to 100s of μs (which is about the execution time of a microservice).

We quantify the impact of core reassignment on the tail latency of microservices. We run DeathStarBench microservices [40] on our server with 4-core VMs, inducing the same core utilization as the one in the Alibaba traces. We detach an idle core from a Primary VM and attach it to a Harvest VM. Later, when the Primary VM receives a request, we move the core back. In our experiments, the Harvest VM is always idle. Hence, the caches of the reassigned core are not flushed/invalidated. Figure 4.3 shows the tail latency of microservices in Primary VMs without (*No-Move*) and with the overhead of core reassignment. We execute the system with the open-source KVM hypervisor [153] and move a core from the Primary VM either on termination of a request invocation only (*KVM-Term*), or on both termination and at every blocking I/O call in the invocation (*KVM-Block*). In either case, we move the core from a Primary to the Harvest VM *only* if the Primary VM has no other requests ready to run. We observe an average of 11 and 36 core reassignments per second with *KVM-Term* and *KVM-Block*, respectively. We also emulate the optimized reassignment latencies reported in SmartHarvest [134] (*Opt-Term* and *Opt-Block*).

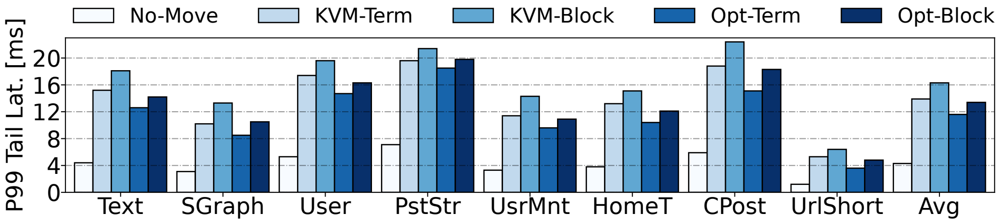


Figure 4.3: P99 tail latency of microservices in Primary VMs with the hypervisor overheads of core reassignment.

We see that core reassignment with *KVM-Term*, *KVM-Block*, *Opt-Term*, and *Opt-Block* increase the P99 tail latency substantially: by 3.2×, 3.8×, 2.7×, and 3.1×, respectively, on average. It can be shown that the overheads increase even further with higher numbers of cores and VMs, and with higher loads. Since these overheads are high, SmartHarvest keeps some idle cores on stand-by in an "emergency" buffer, resulting in even lower core utilization. These cores are reclaimed by Primary VMs when they receive new work.

• **Cache flush and invalidation on core re-assignment and subsequent cold-cache restart are expensive.** As discussed in Section 4.2, we flush and invalidate a core's private caches and TLBs when the core is reassigned from one VM to another. Unfortunately, current processors do not have an efficient way to do it. For example, Intel's `wbinvd` [154] flushes and invalidates the *whole* cache hierarchy of a given core, and takes 300–500μs. Further, Intel's `clflush` flushes only one cache line, so one needs to execute it many times. In addition, as an invocation starts on a re-assigned core, it finds cold caches and TLBs.

We again consider two cases: a conservative design where an idle core is taken from a Primary VM only when it has finished executing a request, and an aggressive design where an idle core is also taken when it is blocked on I/O. We note that starting (or restarting) a request on a cold cache and TLB is costly. Even threads that handle different invocations of the same microservice share a large fraction of their memory footprint.

Figure 4.4 quantifies the impact of flushing and invalidating caches and TLBs (via the `wbinvd` instruction) and restarting with cold caches and TLBs. The figure shows the tail latency of microservices in Primary VMs without flushing, with flushing in the conservative design (*Flush-Term*), and with flushing in the aggressive design (*Flush-Block*). With wbinvd, the processor does not wait for the external caches to complete their write-back and invalidation operations. It only waits for internal caches (L1/L2) to be written-back and invalidated. As we use the existing wbinvd instruction, these experiments do not include the time it takes to write back and invalidate external caches. Hence, this implementation is not safe. In our evaluation (Section 4.6), when we simulate this architecture, we place a fence after the wbinvd instruction. The figure has two extra bars (*Harvest-Term* and *Harvest-Block*) which add the overhead of core re-assignment using the optimized hypervisor software shown in Figure 4.3. These bars represent the current true cost of core re-assignment [134].

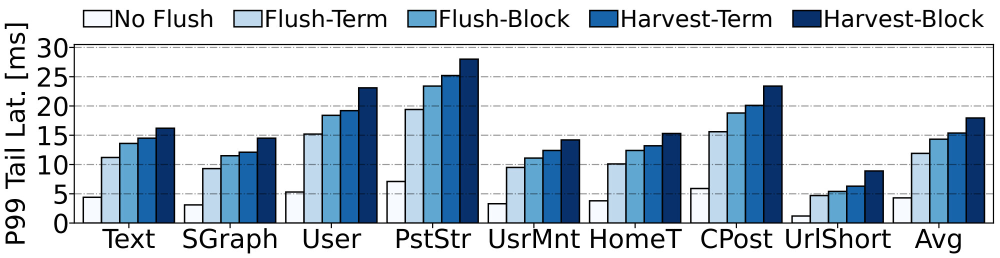


Figure 4.4: P99 tail latency of microservices in Primary VMs with cache/TLB flushing and, for the last two bars, both cache/TLB flushing and hypervisor core reassignment.

On average, cache/TLB flushing increases the P99 tail latency by 2.7× (*Flush-Term*) and 3.3× (*Flush-Block*). Further, if we add the hypervisor reassignment overhead, the tail latencies of *Harvest-Term* and *Harvest-Block* are 3.6× and 4.2× higher, respectively, than the no-flush design. These are the substantial costs that hardware-supported harvesting may help minimize.

Putting it all together, Figure 4.5 shows the execution time of a single service request in steady state without and with core harvesting. The core harvesting environment includes the optimized hypervisor core reassignment design in [134] and cache/TLB flushing and invalidation. The figure shows two bars per service: one with no harvesting (left bar) and one with harvesting (right bar). The latter is broken down into three components: hypervisor reassignment of cores (*Core Reassign*), flush/invalidation of the caches and TLBs (*Flush/Inval*), and execution of the request (*Execution*). We see that, on average, a request takes 1.9× longer with core harvesting. In addition to the core reassignment and flush/invalidation overheads, the execution time itself under core harvesting takes 1.2× longer than before due to using cold microarchitectural structures.

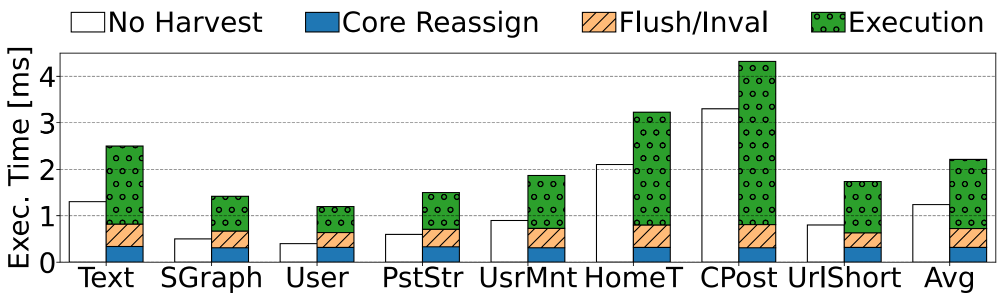


Figure 4.5: Execution time of a single service request in steady state without core harvesting (left bar) and with core harvesting (right bar). The latter is broken down into its components.

Opportunity: **Microservice invocations have relatively small working sets.** In an environment with frequent cache and TLB flushing and invalidation, it may be reasonable to reserve a section of these structures for the Primary VM, and not flush this section. But, this approach is only plausible if microservice invocations have small working sets. In practice, this is the case. We assess the working set sizes of DeathStarBench microservices in two ways. First, we configure our server with a smaller LLC and do not observe any performance change. Specifically, our IceLake server has a 54MB LLC organized in 12 ways. We use Intel CAT [155] to partition the LLC and allow a microservice to use either the full LLC, 3/4, 1/2, or 1/4 of the LLC. We observe that, even with a 1/4 of the LLC, the tail latency of microservices does not degrade more than 1%.

Second, we simulate a server where *all* caches and TLBs are smaller: we model the server with the full caches/TLBs, and then we reduce the number of ways in all structures to 75%, 50%, and 25%, while keeping the number of sets constant. We also model the performance of a simulated environment with infinite caches and TLBs. We use the SST simulator [98] with QEMU [156], and validate the simulation accuracy by calibrating the results with the real system, with and without LLC partitioning. Figure 4.6 shows the tail latency of microservices when running with different sizes of all caches and TLBs. All microservices see a very small impact even when operating with 1/2 of the whole cache/TLB hierarchy.

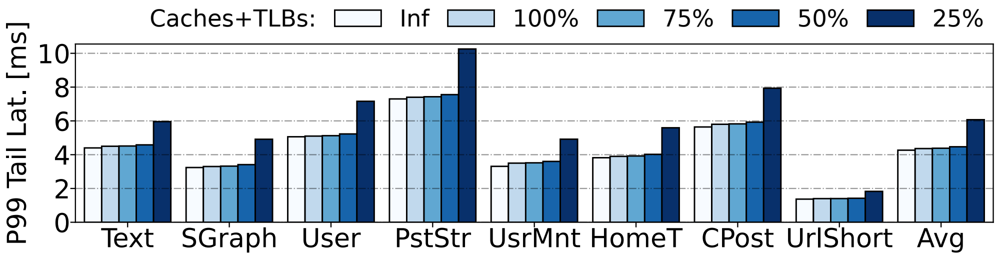


Figure 4.6: Tail latency of microservice invocations on a system with a fraction of the whole cache and TLB hierarchy.

4.4 HARDHARVEST: CORE HARVESTING IN HARDWARE

Based on the previous observations, we propose *HardHarvest*, the first processor microarchitecture that delivers high-performance core harvesting by *supporting it in hardware*. HardHarvest allows Harvest VMs to steal idle cores from Primary VMs with very low overhead, and return them on demand with minimal impact on the tail latency of Primary VMs. In this way, HardHarvest simultaneously enables high utilization of cores, high throughput for Harvest VMs, and minimal impact on Primary VMs. HardHarvest targets the two main sources of overhead in core harvesting: (i) core reassignment and (ii) cache and TLB flush/invalidation and cold restart. We present how we address each source in turn.

4.4.1 Minimizing Core Reassignment Overhead

4.4.1.1 Main Idea of the Proposed Solution

To minimize the latency of core reassignment, we propose to support part of its operation in hardware. Specifically, we design hardware schedulers that schedule requests for VMs. The schedulers optimize core migration between VMs. Further, cores dequeue requests from their own VM's queue by using a low-overhead *dequeue* instruction. When a core cannot find work, it is automatically reassigned to a Harvest VM by simply allowing it to dequeue a job from that VM's queue. There is no need to issue (de)attach system calls as scheduling is done in hardware.

A conventional *detach* operation involves: 1) issuing a hypervisor call (switching from user-space to privileged mode) [157], 2) acquiring a lock [158], and 3) sending an interrupt to the affected core [159]. An *attach* operation follows the same steps. In each case, HardHarvest's hardware avoids the first two software overheads. First, it bypasses the hypervisor and directly re-assigns the cores across VMs in hardware. Second, as hardware schedulers work in a decentralized manner, there is no need to acquire the global lock.

In addition, when a Primary VM receives a request and all its cores are busy, if any of its cores is executing a request for a Harvest VM, an interrupt is sent to one of such cores. That core then saves the state of its Harvest VM's request, performs a context switch, and dequeues the new request from the Primary VM's queue without issuing any hypervisor (de)attach system calls.

With this hardware, we estimate that a core re-assignment from Harvest to Primary VM takes a few μs. If, in addition, we speed-up context switching by adding hardware support for saving and restoring the process state, we estimate that a core re-assignment takes *a few 10s of ns.* Recall that a software approach to reassign a core across VMs takes from a few hundreds of μs to a few *ms.*

4.4.1.2 Detailed Hardware Design

In HardHarvest, a processor has a hardware controller with request queues. Figure 4.7 shows the design. There is a single hardware request queue (RQ) that is dynamically divided into different (logical) subqueues, one for each running VM. To ensure isolation, VMs cannot access each other's subqueues. In addition, there are a number of hardware *Queue Managers* (each of which can control a request subqueue) and a number of *VM State Register Sets* (each set can store a VM state shared by all the threads of a VM). Such state includes registers such as the VMCS pointer, CR0, CR3, CR4, GDTR, LDTR, and IDTR. Each subqueue is given a Queue Manager and a VM State Register Set.

When a user allocates a VM, they specify the number of cores to use. Those many cores are then logically bound to the VM. This is done by setting a core register (*MyManager*) with the ID of the Queue Manager in charge of the VM. The relative number of cores bound to each VM determines the relative fraction of the RQ entries assigned to each VM's subqueue. Hence, the sizes of the individual subqueues may dynamically change as new VMs arrive to the server and old VMs are removed.

To allow such flexibility, the physical RQ is broken into chunks, and each VM's subqueue is composed of one or more chunks. When a new VM is spawned on a server, it gets a few

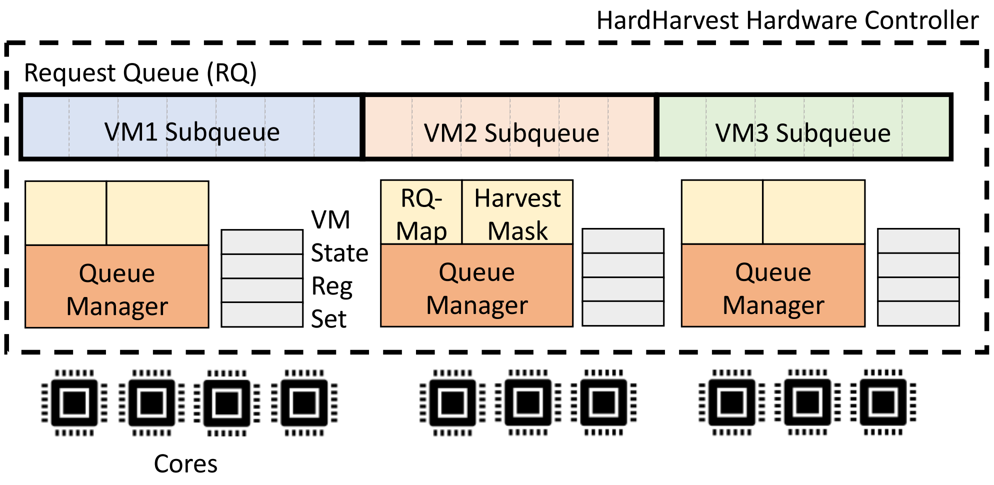


Figure 4.7: HardHarvest hardware controller.

chunks from the currently-active VMs. A VM donates one or more chunks from the tail of its subqueue. If some of the entries in those chunk(s) are full, the corresponding entries are moved to a software-based *In-memory Overflow Subqueue* for that VM. We discuss this structure later. Likewise, when a VM leaves the server, its chunks are assigned to the tails of the subqueues of the remaining VMs.

While a subqueue is logically contiguous, its chunks do not need to be physically contiguous. Every Queue Manager has an *RQ-Map* that maps the logical chunks of a VM's subqueue to their physical chunks. Therefore, when a VM's subqueue sheds a chunk, it simply invalidates the entry in its RQ-Map; when a VM's subqueue gets a new chunk, it inserts a new entry at the tail in its RQ-Map. In our implementation, the RQ has 32 chunks of 64 entries each. The total storage of an RQ-Map is 24B, i.e., up to 32 entries with 5 bits for the physical chunk ID and 1 valid bit.

To enable concurrency, each chunk of the RQ has its own access port. Since individual chunks are exclusively owned by a given Queue Manager, different Managers do not contend on such ports. All 32 chunks can be accessed in parallel.

4.4.1.3 Request Arrival and Processing

Every VM has its own network address. When the NIC receives a packet with a request (Figure 4.8(a) 1), it deposits the message payload in the LLC via DDIO 2, and reads the message's destination VM. Then, it checks a local software table that tells which Queue Manager (QM) is in charge of which VM. It informs the corresponding QM 3, which stores in its Request Subqueue a pointer to the request payload 4. If the Request Subqueue is full, the QM instead stores the pointer in the In-memory Overflow Subqueue of its VM. In either case, the request is marked as ready.

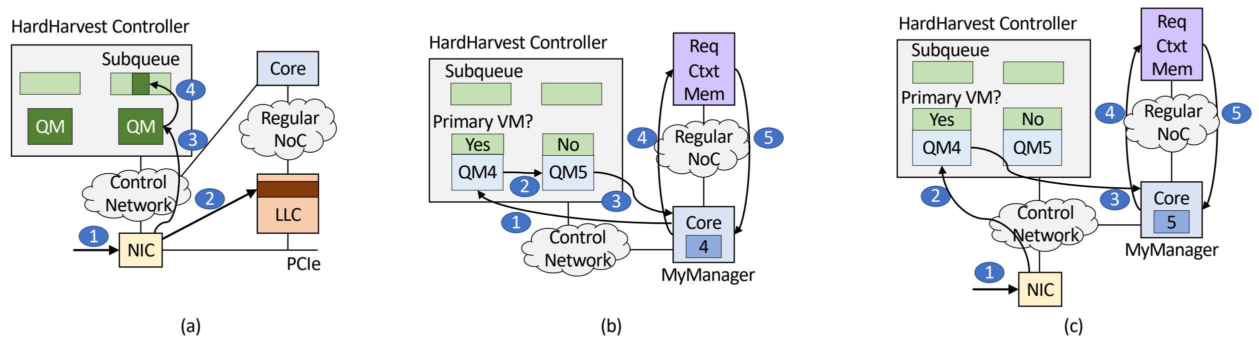


Figure 4.8: Path events in HardHarvest: (a) request arrival, (b) core re-assignment, and (c) core reclamation.

A core is bound to a QM through the *MyManager* register. Cores have instructions to spin on a Request Subqueue for work, to dequeue a request, and to inform the Request Subqueue when the request has been completed or when it is blocked on I/O. Such instructions access the QM corresponding to the core's *MyManager* register. This QM identifies the Request Subqueue to spin on, the request to dequeue, the request to remove as completed, and the request to mark as blocked, respectively. Moreover, each QM knows if it is managing a Primary or a Harvest VM and, if the former, which of its bound cores are currently "on loan" executing requests of a Harvest VM.

4.4.1.4 Operation of Core Reassignment

As a core bound to a Primary VM spins on the subqueue of its *MyManager* QM and there is no request to process (Figure 4.8(b) ①), the QM forwards the core's request to a Harvest VM's QM ②. A Harvest VM runs a batch application and is expected to always have available work. Hence, the Harvest VM's QM sends a process to the requesting core ③, together with the VM State Register Set associated with the QM. On receiving the message, the requesting core may have to save the state of its current process (if it was blocked). It also restores the state of the new Harvest VM process, loads the VM State Register Set of the new VM that it receives from the HardHarvest hardware controller, and proceeds to execute the new process.

To avoid entering the kernel in this case, HardHarvest can use hardware that automatically saves/restores the process register state on a context switch. Specifically, the current state is saved in a special Request Context Memory connected to the on-chip network ④ and the new state is restored from there ⑤. This hardware extends prior proposals for in-hardware context switching across requests [31], to additionally perform a VM context switch.

4.4.1.5 Reclaiming a Core by a Primary VM

A core is quickly reclaimed by its Primary VM on demand. When a new network packet arrives at the NIC (Figure 4.8(c) ①) and is either a new request for a Primary VM or a network response to a blocked request of a Primary VM, the NIC informs the corresponding Queue Manager (QM) ②. The QM checks if: (i) none of its bound cores is idle, and (ii) at least one of its bound cores is executing a request for a Harvest VM. If so, it interrupts one such core and passes it the new process and the correct VM State Register Set ③. Note that the interrupt is sent in hardware by the QM of a Primary VM—not by the VM itself. A Primary VM is never aware that another VM was running on one of its bound cores. The interrupted core immediately performs a context switch as described above: it saves the state of the current process ④, loads the new VM State Register Set received from the QM, and restores the state of the new process belonging to the Primary VM ⑤.

As the Harvest VM loses the core, the process that was running there (i.e., the vCPU) is returned to the queue of the Harvest VM vCPUs. The Harvest VM's QM multiplexes its vCPUs onto its remaining physical cores (pCPUs), similarly to an over-subscribed environment. Since Harvest VMs are configured with as many vCPUs as there are pCPUs in the server [134], the software running in the Harvest VM does not require any changes. However, as the Harvest VM knows the current number of pCPUs available to it, it can adapt dynamically, either by reducing parallelism or rescheduling tasks [134]. There is no risk of deadlock from preempted Harvest VM threads holding locks as they will eventually run on the remaining pCPUs when they are scheduled again, ensuring forward progress.

Figure 4.9 summarizes core reclamation. In Figure 4.9(a), the red core bound to the Primary VM is currently executing request *ID5* of the Harvest VM, and a new request *ID6* of the Primary VM arrives. In Figure 4.9(b), the core is interrupted and forced to execute *ID6*, leaving *ID5* in a ready state for another core to take.

When a core running a request for a Primary VM blocks on I/O, it may be stolen. However, the pointer to the request is kept in the corresponding Request Subqueue. Later, when the NIC receives the network response to the blocked request, the NIC informs the QM. The QM marks the request in the Request Subqueue as ready and the procedure described above is followed to reclaim the core.

Our HardHarvest design uses FIFO scheduling within a VM. Future work could explore customized scheduling policies, either based on application-provided information, or learned by the system. For example, the application could identify periods when bursts are expected or when the SLO is likely to be violated frequently. In response to this, the system could reduce the harvesting aggressiveness by, for example, keeping a buffer of idle cores ready for Primary VM bursts. Alternatively, the system could monitor events such as when requests spend a very short time blocked on I/O. In this case, the system could dynamically switch from harvesting on blocking call to harvesting only on request completion.

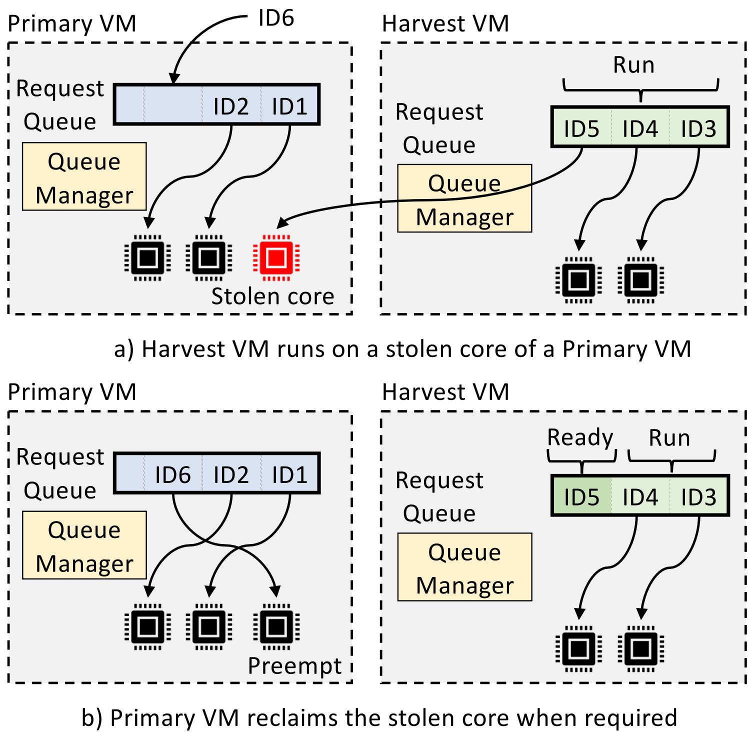


Figure 4.9: In-hardware core reassignment between VMs.

4.4.1.6 The Need for a Hardware Scheduler

HardHarvest uses a hardware scheduler instead of CPU-centric software scheduling for two reasons. First, a hardware scheduler allows fast core notification when a Primary VM request enters the queue, as the QM checks if the core needed by the VM is on-loan to a Harvest VM and instantly alerts it. In contrast, a software scheduler would require cores to poll memory locations, lowering throughput by diverting core cycles from application logic to checking for new requests. Second, a hardware scheduler minimizes queue contention, eliminating the need for locking mechanisms when multiple cores access the same subqueue, which software scheduling requires.

A hardware scheduler can use memory-mapped queues or hardware queues. Memory-mapped queues are inexpensive and flexible. However, hardware queues have better performance for two reasons. First, dedicating special SRAM queues for incoming requests reduces the contention between (i) the scheduler and (ii) the cores and NIC on the cache hierarchy. Indeed, accesses from the hardware scheduler to the special SRAM queues do not compete with regular cache hierarchy accesses by cores or NIC accesses that deposit requests using DDIO. Second, hardware queues and their network can be designed to have low access latency. Hence, in HardHarvest, we design the RQ (Figure 4.7) as a dedicated SRAM hardware queue. However, HardHarvest could also be integrated with memory-mapped queues.

4.4.1.7 Limitations of Prior Hardware Queues for Microservices

Prior work has used hardware queues in microservice environments [31, 115]. However, these designs lack four important supports that HardHarvest provides. First, they assume that all cores can pick requests from the same queue, without any security concerns. In a cloud setup, one must isolate different users. Thus, in HardHarvest, the RQ is split into per-VM subqueues that are managed by different hardware Queue Managers (QMs). These QMs operate in parallel and on distinct subqueues, avoiding any sharing or contention.

Second, prior designs assume non-virtualized environments. Thus, a request's state is only the state of a Linux process. However, cloud workloads run inside sandboxed VMs. Thus, when executing a request, a core needs to load both request and VM contexts. In HardHarvest, each QM keeps the VM state in the VM State Register Set (Figure 4.7).

Third, prior designs do not have the notion of a Harvest VM process being pre-emptable by a Primary VM process. In HardHarvest, a Harvest VM running on a stolen core may be immediately preempted when a new request for the owner Primary VM is received. The QMs have logic to detect such scenarios and enforce the reassignment across VMs.

Finally, prior proposals are inflexible in that they work with fixed-size queues. Instead, in HardHarvest, per-VM subqueues can dynamically change their size as new VMs are allocated or old VMs depart the server. Further, each subqueue has a software In-memory Overflow Subqueue in main memory that holds requests that overflowed the subqueue.

4.4.1.8 Implementation Details

The HardHarvest controller of Figure 4.7 is a centralized hardware module in the chip that is accessed with a dedicated network. We use a special network for two reasons. First, accesses to the controller should not compete with the regular workload traffic. Second, the control network has different needs than the regular network. As it transfers mostly control messages, and it is latency-, not bandwidth-sensitive, it has thin links. In our design, we use a tree topology.

Cores communicate only with the QMs and not with the Request Subqueues. This approach is both secure and avoids data races. The enqueue, dequeue, and other instructions are user-level instructions that are embedded in libraries. These instructions are transparent to the application developers. For example, `CompletionQueue::Next` [160] in gRPC and `TServerSocket::listen` [161] in Thrift are augmented with the HardHarvest dequeue instruction.

The processor chip also includes the special Request Context Memory where the hardware for fast context switch proposed by μManycore [31] (Chapter 3) saves and restores the core state in a context switch. Such memory is connected to the regular NoC. As saving and restoring is done in hardware, there are no new instructions.

4.4.2 Cache/TLB Flush/Inval & Cold Restart

4.4.2.1 TLB and Cache Partitioning

Following existing practice [137, 138, 139, 140, 141, 142, 143, 144], HardHarvest would need to flush and invalidate the L1/L2 caches and L1/L2 TLBs of a core when the core is re-assigned across VMs, to ensure that the structures do not leak information. The LLC does not need to be flushed because it is partitioned using Intel's CAT [155]. To minimize this overhead, HardHarvest uses the fact that microservice invocations have relatively small working sets for both data and instructions [14, 40]. Specifically, HardHarvest proposes to partition the L1 caches (D and I), L2 cache, L1 TLBs (D and I), and L2 TLB in a way that minimizes the overhead for Primary VMs and still allows Harvest VMs to attain good performance.

Each of these structures is way-partitioned into one *Harvest Region* and one *Non-Harvest Region*. When a Primary VM runs, it uses the whole structure; when a Harvest VM runs, it can only use the harvest region—which may be, e.g., 1/2 or 1/3 of the ways of the structure. The non-harvest region is inaccessible to the Harvest VMs, and keeps state of the Primary VM that will be reused when the core is returned to the Primary VM.

When a core transitions from a Primary to a Harvest VM, the harvest region is flushed and invalidated. The Harvest VM is not allowed to start execution until a certain time has elapsed equal to the longest possible duration of the flush/invalidate operation. This is done to eliminate a timing side-channel.

When a core transitions from a Harvest to a Primary VM, again only the harvest region is flushed and invalidated. However, the Primary VM restarts execution *right away* when the core is reclaimed, reusing the data stashed away in the warmed-up non-harvest region. In the background, the harvest region is flushed and invalidated. When the invalidation is completed and the worst-case time has elapsed to ensure there is no timing side-channel, the empty ways also become visible to the Primary VM. Hence, flushing and invalidating the harvest region is not in the critical path.

For a Primary VM, what fraction of the ways in the L1/L2 caches and TLBs are the harvest region can be a default value or specified by the software. This information is saved in a *HarvestMask* hardware register in the Queue Manager of the VM (Figure 4.7). HarvestMask contains a bit per way for each of the structures, for a total of 5B. A bit is set if the way is in the harvest region. When a core is assigned (or re-assigned) to a VM, the system knows whether the VM is Primary or Harvest, and obtains the VM's HarvestMask.

Then, before the core starts executing, the HarvestMask is used to reconfigure the private caches/TLBs in a way similar to CAT [155]. For example, if the core is executing a Harvest VM, the non-harvest ways of the L1/L2 caches and TLBs are inaccessible to the requests. Coherence messages such as invalidations are still received for data in either the harvest or the non-harvest ways, since data is not remapped. To improve performance, HardHarvest could profile a workload and recommend an initial non-harvest region size. Then, the system could transparently learn during execution the best non-harvest region size by opportunistically trying to change it and seeing the performance impact.

A Harvest VM such an ML training job could make use of more space than the harvest region partition. However, HardHarvest is still attractive to these workloads because, while the harvested cores have reduced cache capacity, renting them has a lower price—thus improving cost-efficiency for latency-tolerant applications.

4.4.2.2 Improving Cache Allocation for Primary VMs

To improve the performance of Primary VMs, HardHarvest tries to keep in the non-harvest region the state that is most likely to be reused when a core is reclaimed back by a Primary VM. To understand what this state is, we consider two types of pages: those that are *shared* across different invocations of the same service, and those that are *private* to a particular invocation of the service. Shared pages include program code, libraries, read-only input data and, generally, data pages that are allocated in a microservice before forking a process to execute a particular invocation of the microservice. Private pages are those allocated after forking a process for a particular microservice invocation. Generally, shared pages are more likely to be reused across core reassignments. So, HardHarvest tries to keep entries from shared pages in the non-harvest region.

We can assume that all instructions and all data objects allocated by the process that initializes a microservice are potentially shared. For example, in microservices implemented using the Thrift [79] or gRPC [41] frameworks, HardHarvest assumes that all data allocated from the start of the microservice until executing `server.serve()`[162, 163] is shared data. If the shared data gets reallocated to expand its size, the new pages are also assumed shared. On the other hand, data allocated by the threads in individual invocations [164] is assumed to be private. Our profiling of more than 60 microservices from open-source DeathStarBench [40], TrainTicket [78], and μSuite [50] benchmark suites confirms this behavior.

Based on these assumptions, when a page is allocated by a Primary VM, HardHarvest sets a *Shared* bit to 1 or 0 in its page table entry. Such bit is copied to the TLB entries, and determines, on an access, the *preferred* ways where a TLB entry or a cache line is placed.

4.4.2.3 Cache/TLB Replacement Algorithm

HardHarvest changes the algorithm that picks the victim way when inserting an entry in private caches or TLBs. The aim is to steer shared entries to the non-harvest region (*Non-Harv*) ways and private entries to the harvest region (*Harv*) ways, while keeping the algorithm simple. Algorithm 4.1 shows which way to pick when inserting entry *E* in Set *S*. In the cases when the algorithm can pick one of multiple victim candidates, it picks the victim using the default replacement algorithm, which is LRU.

Algorithm 4.1: Inserting an entry in a private cache or TLB.

```
Inputs: E = entry to insert; S = set where E maps
Result: Cache/TLB way that takes E
if S has empty slots then
    if empty slots in both Non-Harv and Harv then
        if E is shared then
            Take an empty slot in Non-Harv
        else // E is private
            Take an empty slot in Harv
        end
    else
        Take an empty slot
    end
else
    if E is shared then
        if any Non-Harv slot has a private entry then
            Take the slot of one of them
        else
            if any Harv slot has a private entry then
                Take the slot of one of them
            else // All S slots have shared entries
                Take a slot
            end
        end
    else // E is private
        if any Harv slot has a private entry then
            Take the slot of one of them
        else
            if any Non-Harv slot has a private entry then
                Take the slot of one of them
            else // All S slots have shared entries
                Take a slot
            end
        end
    end
end
```

First, assume that there are empty slots in *S*. If there are both *Non-Harv* and *Harv* empty slots, a shared entry takes a *Non-Harv* empty slot and a private entry takes a *Harv* empty slot; otherwise, *E* takes an empty slot. If, instead, there is no empty slot in *S*, the action depends on whether *E* is shared or private. If *E* is shared, the algorithm proceeds as follows. First, it checks *Non-Harv* for any slots with a private entry. If there are any, *E* evicts one of them. Otherwise, *Harv* is checked for any slots with a private entry. If there are any, *E* evicts one of them. Otherwise, all the slots in *S* have shared entries, and the algorithm picks one of them as the victim.

Instead, if *E* is private, the algorithm swaps steps one and two above. First, it checks *Harv* for any slots with a private entry. If there are any, *E* evicts one of them. Otherwise, the algorithm checks *Non-Harv* for any slots with a private entry. If there are any, *E* evicts one of them. Otherwise, all the slots in *S* have shared entries, and the algorithm picks one of them as the victim. The algorithm swaps steps when *E* is private because a private entry in *Non-Harv* will not remain cached for too long. A shared entry may soon evict it.

Since *Shared=1* for all instruction pages, this algorithm does not change the default behavior of the L1 instruction cache/TLB. The L2 cache/TLB have instructions and data. There, the algorithm prioritizes instructions over private data. In practice, both data and instructions in microservices have small working sets [14, 40]. Thus, assigning about half of the ways to *Harv* during harvesting is typically tolerable.

Continuously prioritizing shared entries within a set can penalize the performance of private entries. For example, if a set at some point has only shared entries, from that point on, all private entries will compete for a single way, making the set appear as direct-mapped. To avoid this issue, HardHarvest uses Algorithm 4.1 to pick an eviction victim only among the *M* least-recently used entries in the set. These entries are called *Eviction Candidates*, and can be, e.g., 1/2 or 3/4 of all the entries of the set. The other, more-recently-used entries are not considered. In this way, popular private data avoids eviction, allowing services that have large private memory footprints to operate with high performance.

With this design, HardHarvest makes a best effort to keep shared entries in the *Non-Harv* ways without fully sacrificing the associativity for private entries.

4.4.2.4 Hardware Implementation of the Replacement Algorithm

We implement Algorithm 4.1 in a simple manner. Specifically, each TLB/cache entry has a *Shared* bit that is set to 1 if the entry has Shared state. Also, each way has a *Harvest* bit, which is set to 1 if the way is a harvest way. These bits, together with the *Invalid* bit of an entry, are used as inputs to two priority multiplexers that determine the victim entry that should be replaced. One of the multiplexers is used for incoming shared entries and the other for private ones.

The multiplexer for incoming shared entries selects the victim entry based on this decreasing priority: Invalid and Not Harvest; Invalid; Not Harvest and Not Shared; Harvest and Not Shared. The multiplexer for incoming private entries uses this decreasing priority: Invalid and Harvest; Invalid; Harvest and Not Shared; Not Harvest and Not Shared.

4.4.2.5 Other Issues

Current processors flush and invalidate caches inefficiently (Section 4.3). Hence, for HardHarvest's mechanisms to be effective, they must be coupled with support for efficient flush/invalidate as proposed elsewhere [165, 166]. In our evaluation, we will assume such support and evaluate its contribution to performance separately.

4.5 METHODOLOGY

Modeled Architectures. We model a cluster of 8 servers, where each server has 36 beefy cores and 128GB of memory. Cores and caches are modeled after the Intel Sunny Cove microarchitecture [167, 168, 169] present in the IceLake server processors [97]. Each core has private L1 and L2 caches and TLBs, and a shared, physically distributed L3 cache. Each server has 8 Primary VMs, each with 4 cores, and 1 Harvest VM, which starts with 4 cores and harvests additional cores from Primary VMs. Detailed architecture parameters are shown in Table 4.1. We evaluate five systems:

• *NoHarvest* is a conventional system where no VM performs core harvesting. As a result, many cores remain idle.

• *Harvest-Term* is a state-of-the-art *software* core harvesting system as described in SmartHarvest [134]. A Harvest VM harvests cores only when the core is idle because it terminated a request, and based on load prediction. We use *Harvest-Term* as the baseline. We also model a more aggressive software design where, in addition, a Harvest VM also harvests cores when a request issues a blocking call and the core it was running on becomes idle (*Harvest-Block*).

• *HardHarvest* is the design proposed in this chapter. We consider two versions: one where a Harvest VM harvests idle cores only when they are idle because a service request terminates (*HardHarvest-Term*), and one where, in addition, a Harvest VM also harvests cores when they are idle because a request issues a blocking call and the core becomes idle (*HardHarvest-Block*). The latter is our proposal.

All schemes use Intel's DDIO technology. In the baseline schemes (*NoHarvest* and *Harvest-Term/Block*), the NIC deposits the whole request in the LLC.

Simulation Infrastructure. We evaluate the architectures with full-system simulations

Table 4.1: Architectural parameters used in the evaluation.

System and Processor Parameters	
Machine	Cluster of 8 servers
Server processor	36 6-issue cores at 3GHz
OoO Execution	352-entry ROB, 200-entry LSQ
L1 D-Cache	48KB, 12-way, 5 cyc. round trip (RT), 64B line
L1 I-Cache	32KB, 8-way, 5 cyc. RT, 64B line
L2 Cache	512KB, 8-way, 13 cycles RT, 32 MSHRs
L3 Cache	Per core: 2MB, 16-way, 36 cyc. RT, 32 MSHRs
L1 TLB	128 entries, 4-way, 2 cycles RT
L2 TLB	2048 entries, 8-way, 12 cycles RT
Network	
Intra Server	2D mesh, 5 cycles/hop
Inter Server	1μs RT; 200GB/s
Virtual Machines	
Primary VMs	8 VMs/server, each with 4 cores (fixed)
Harvest VMs	1 VM/server, with 4 cores + harvested cores
Main Memory per Server	
Capacity; Rate	128GB; DDR4-3200; 4 memory controllers
Mem. Bandwidth	102.4GB/s per socket
HardHarvest Parameters	
Num chunks in RQ	32
Num entries/chunk	64
Num Queue Manag.	16
Num regs in VM State Regs	16
Ways in Harv. Region	50% of all ways
Eviction Candidates (M)	75% of all ways
Flus+Inv HarvRegion	1000 cycles

using QEMU [156] and SST [98]. QEMU captures both user-space and kernel-space instructions, memory accesses, and system calls. QEMU forwards all the events to SST, which models the architectures and performs cycle-level simulations. Thus, the simulation models the whole software stack: OS (Ubuntu 20.04), hypervisor, harvesting agents, and application logic. The main memory system is modeled with DRAM-Sim2 [99].

Our modeled cluster is comprised of 8 servers to test a different type of Harvest VM workload in each server, as we discuss later. Such servers execute without requiring any communication between them. This is because microservices do not communicate across servers. Specifically, each server hosts an instance of each of the evaluated microservices, but microservices only communicate with other microservices that are placed on the same server. We use the 1μs inter-server communication latency to model the network latency when a service accesses remote caches (e.g., Memcached), key-value stores (e.g., Redis), or databases (e.g., MongoDB). These backend services (Memcached, Redis, and MongoDB) run on dedicated servers. We do not simulate the execution of the queries on the backend services. Instead, we use the execution times obtained by profiling them on a real server.

To make the simulation time tolerable, we run the simulations in parallel. Each server is simulated on a different physical machine because servers do not communicate with each other. In addition, within each simulated server, SST also runs in parallel. With this design, the longest simulation (corresponding to 30 seconds of wall-clock time) takes 4 days.

Applications. For the latency-critical Primary VMs, we use 8 SocialNet microservices from DeathStarBench [40]. For the batch workloads executed in the 8 Harvest VMs, we use graph applications from GraphBIG (BFS, CC, DC, and PRank) [170], ML training from Function-Bench (LRTrain and RndFTrain) [171], data analytics from CloudSuite (Hadoop) [172] and bioinformatics from BioBench (MUMmer) [173]. We deploy each Primary VM with 4 cores because this is the most common size for Alibaba's microservice instances [83]. Each Harvest VM also starts with 4 cores. As indicated before, each server has one Harvest VM running one of the batch applications, and 8 Primary VMs, each running one of the microservices.

We pick 8 representative services from Alibaba's production-level open source traces [5] and we mimic their behavior with our 8 DeathStarBench services. Hence, *we execute with real-world invocation rates*, using an open-loop load generator that keeps the load the same across all systems (i.e., the client is independent of the server) [50]. The average load per Primary VM core is 65-250 requests per second (RPS). We report average and tail latency after executing 100K microservice invocations across all 64 Primary VMs. As done in prior work [174, 175], we match a service in the production trace to the service in the DeathStarBench suite that has the most similar service execution time.

4.6 EVALUATION

4.6.1 End-to-End Tail Latency of Primary VMs

Figure 4.10 shows the P99 tail latency of microservices running in Primary VMs for the 5 evaluated architectures. Software harvesting schemes (*Harvest-Term* and *Harvest-Block*) have a high tail latency due to software overheads. The average tail latency in *Harvest-Term* and *Harvest-Block* is 3.4× and 4.1× higher, respectively, than in *NoHarvest*. On the other hand, *HardHarvest* reduces the tail latency substantially. Compared to *Harvest-Term*, *HardHarvest-Term* and *HardHarvest-Block* reduce the tail latency by 83.3%. In fact, the tail latency in these architectures is even lower than in *NoHarvest*: the average tail latency in *HardHarvest-Term* and *HardHarvest-Block* is 30.5% and 28.4% lower, respectively, than in *NoHarvest*. The reason is that some *HardHarvest* optimizations such as improved cache/TLB replacement and request queuing in hardware not only speed-up harvesting, but also microservices in general as well. The reductions relative to *Harvest-Term* and *Harvest-Block* are more significant in services that *(i)* operate mostly on shared pages such as *HomeT*, or *(ii)* frequently block on I/O such as *User*.

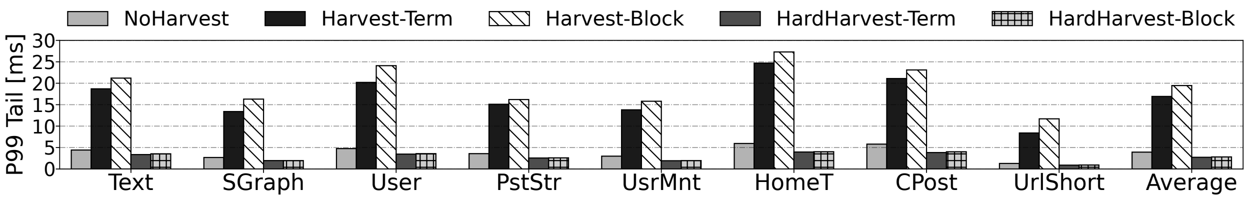


Figure 4.10: P99 tail latency of microservices running in Primary VMs for the 5 evaluated architectures (lower is better).

4.6.2 Tail Latency Reduction Breakdown

Figure 4.11 shows the *cumulative* impact of individual optimizations on the tail latency of Primary VMs. The figure starts with the tail latency of software harvesting with *Harvest-Term* and *Harvest-Block*. It then applies the following optimizations to *Harvest-Block* one by one, in order: hardware request scheduler (*+Sched*), hardware request queues (*+Queue*), in-hardware context switching (*+CtxtSw*), cache/TLB partitioning with LRU replacement (*+Part*), efficient cache/TLB flushing (*+Flush*), and optimized replacement policy (*HardHarvest*). Recall that we borrow *CxtSw* [31] and *Flush* [165, 166] from the literature, as they are needed to take full advantage of our optimizations.

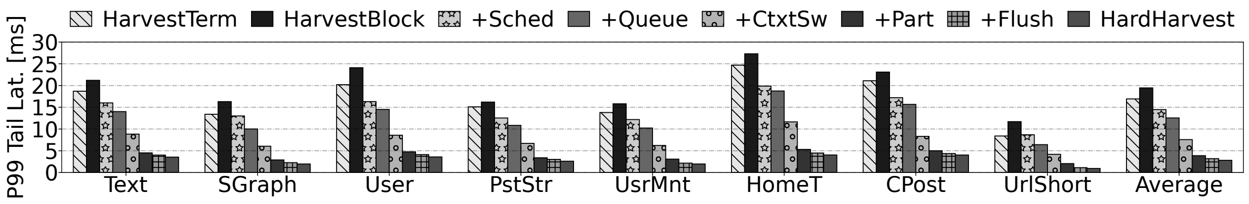


Figure 4.11: Cumulative impact of individual optimizations in *HardHarvest* on the P99 tail latency of Primary VMs. Core harvesting is enabled.

All techniques help reduce tail latency. On average, the gradual application of these optimizations reduces the tail latency of *Harvest-Block* by 25.6%, 35.5%, 61.1%, 80.1%, 83.6%, and 85.6%, respectively. In-hardware request scheduling (*+Sched*) is effective because, e.g., when a service that is blocked on I/O receives the response, the scheduler ensures that it is scheduled right away. Without the scheduler, a polling core would discover the ready service much later, which hurts tail latency. Hardware queuing (*+Queue*) is effective because it reduces contention on the cache hierarchy relative to memory-mapped queues, and also reduces the latency of request fetching. Cache/TLB partitioning (*+Part*) also helps, even with LRU replacement and without advanced hardware flushing. Advanced hardware flushing (*+Flush*) has a small impact because, after a Primary VM resumes, flushing occurs in the background while the Primary VM is already running. Finally, our proposed cache line replacement (*HardHarvest*) further reduces the tail.

To see the relative impact of *Sched* and *CxtSw*, we perform an ablation study in Figure 4.12. The figure takes *Harvest-Block* and applies only *CtxtSw*, then only *Sched*, and then both *CtxtSw* and *Sched*. We see that both *Sched* and *CxtSw* have a similar impact, and when applied together, they have a partially additive effect.

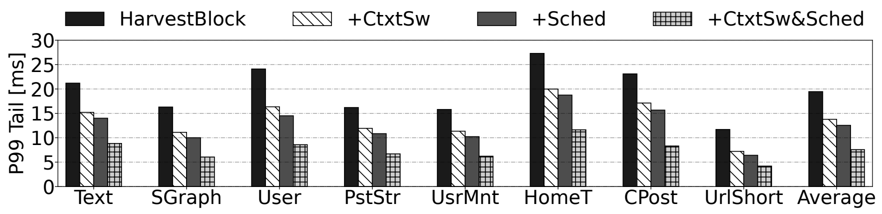


Figure 4.12: Ablation study on the effectiveness of in-hardware context switching and hardware request scheduling.

4.6.3 Impact of the Optimized Cache Replacement Policy

To understand the impact of the HardHarvest cache replacement policy, Figure 4.13 shows the measured L2 cache hit rate in four different environments: vanilla LRU, the RRIP advanced replacement [176], our proposed policy (Algorithm 4.1), and an ideal cache replacement policy (Belady [177]). We see that, on average, our algorithm (*HardHarvest*) increases the L2 cache hit rate over LRU and RRIP by 11.3% and 8.2%, respectively. Since RRIP does not differentiate between Primary and Harvest processes accessing the same cache, its re-reference interval calculations get polluted, leading to sub-optimal performance. *HardHarvest* is within 3.1% of the ideal replacement algorithm. Results are similar for L1 and TLBs.

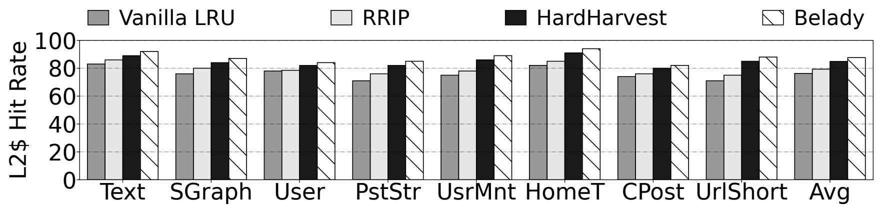


Figure 4.13: L2 hit rate with different replacement policies.

Since shared pages across invocations of the same service include both code and data, we investigate whether prioritizing instruction over data pages in the replacement algorithms of caches and TLBs improves performance. We performed this experiment in *HardHarvest* via Code-Data-Prioritization (CDP) [178] in Intel's CAT. We find that such an approach is not beneficial. It increases the tail latency by 8% over our proposed replacement policy.

4.6.4 HardHarvest Optimizations without Core Harvesting

Figure 4.14 shows the tail latency of Primary VMs as we add HardHarvest optimizations to the NoHarvest baseline without performing core harvesting. We evaluate *+Sched*, *+Queue*, *+CtxtSw*, and *+ReplPolicy* (which is our optimized replacement policy). Since there is no harvesting, cache partitioning and cache flushing are not relevant and not evaluated. We see that all four techniques are effective: they cumulatively reduce the tail latency by 14.5%, 20.1%, 28.6% and 33.6%, respectively. In-hardware request scheduling (*+Sched*) is effective because, when a service that is blocked on I/O receives the response, the scheduler ensures that it is scheduled right away. The scheduler offloads CPU polling. Hardware queues (*+Queue*) reduce contention in the cache hierarchy and the latency to fetch the request. Finally, our replacement policy (*+ReplPolicy*) helps by preserving shared entries in the caches/TLBs across invocations.

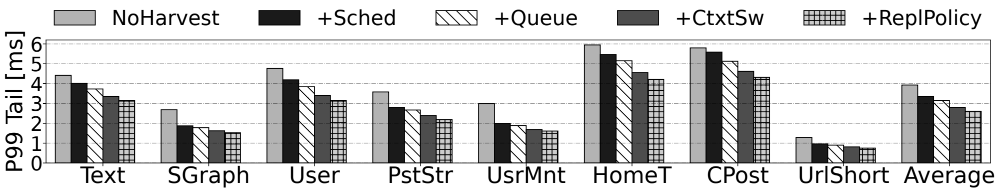


Figure 4.14: Cumulative impact of optimizations on the P99 tail latency of Primary VMs. Core harvesting is disabled.

4.6.5 Median Latency of Primary VMs

Figure 4.15 shows the median latency of microservices in the five evaluated architectures. Although we saw that software harvesting significantly degrades tail latency, it has a modest impact on the median latency. The median latency of *Harvest-Term* is only 7.9% higher than *NoHarvest*. On the other hand, *HardHarvest* not only reduces tail latency but is also effective at reducing the median latency as well: *HardHarvest-Block* reduces the median latency by 26.1% over *NoHarvest*.

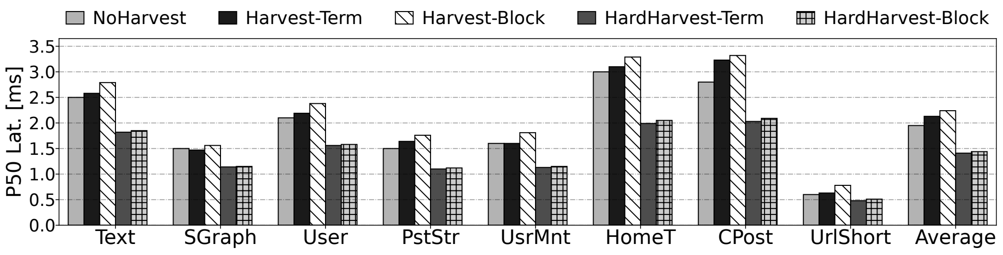


Figure 4.15: Median latency of microservices in Primary VMs.

4.6.6 Throughput of Harvest VMs

The target metric for Harvest VMs is throughput (i.e., the number of jobs executed per unit of time). Figure 4.16 shows the throughput of Harvest VMs with the evaluated architectures normalized to *NoHarvest*. On average, *Harvest-Term* [134] and *HardHarvest-Block* (our proposal) improve throughput by 1.7× and 3.1×, respectively. Memory-intensive applications, e.g., RndFTrain, see slightly lower throughput gains. *HardHarvest-Block* improves the throughput over *Harvest-Term* because it (i) steals cores whose service is blocked on I/O (i.e., it harvests more cores), and (ii) reduces the overheads of core reassignment (i.e., Harvest VMs start running on stolen cores sooner).

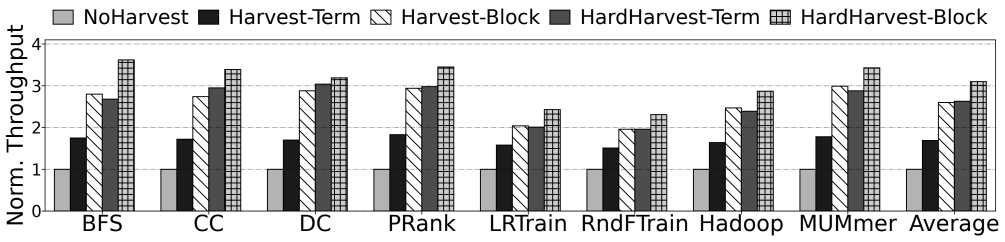


Figure 4.16: Throughput of Harvest VMs with the five evaluated architectures normalized to *NoHarvest*.

4.6.7 Core Utilization

HardHarvest benefits cloud providers as it increases the utilization of the cores. It can be shown that *NoHarvest*, *Harvest-Term*, *Harvest-Block*, *HardHarvest-Term*, and *HardHarvest-Block* have an average utilization of 10.3, 23.8, 26.5, 28.7, and 34.8 cores out of the available 36 cores in the server, respectively. *HardHarvest* has higher core utilization because it performs core harvesting in hardware, using cores efficiently and eliminating emergency buffers. Overall, *HardHarvest-Block* increases core utilization by 1.5× over *Harvest-Term*.

4.6.8 Storage Cost

HardHarvest adds the hardware controller in Figure 4.7 to each server. The storage cost of a controller is a 2K-entry RQ, where each entry has 66 bits (2 bits for the request status and 64 bits for a pointer to the request payload) and, for each of the 16 pairs of QMs and VM State Register Sets: 1) 16 VM State registers of 8B each, 2) a 24B RQ-Map, and 3) a 5B HarvestMask register. The total storage cost per controller is 18.9KB (or 0.53KB per core). On top of that, each entry in the TLBs, L1 D-caches, and L2 caches has an extra *Shared* bit, which results in a total storage cost per server of 67.8KB (or 1.9KB per core). We use McPAT [102] to estimate the power and area overheads of these storage structures. Scaling to 7nm technology [103], the resulting overheads are only 0.19% and 0.16% increases in area and power (dynamic plus static), respectively, of the multicore.

4.6.9 Sensitivity to LLC Size

Throughout the evaluation, we used an LLC with 2MB per core. In this section, we perform a sensitivity study to see the impact of different LLC sizes on the effectiveness of HardHarvest. Note that the LLC is non-inclusive of the L2. Figure 4.17 shows the P99 tail latency of microservices running Primary VMs in HardHarvest-Block with different LLC sizes. When we increase the LLC size to 2.5MB per core, the tail latency reduces because there are fewer misses, while when we decrease the LLC size, the tail latency increases. Overall, the changes in latency are small because microservices have relatively modest footprints.

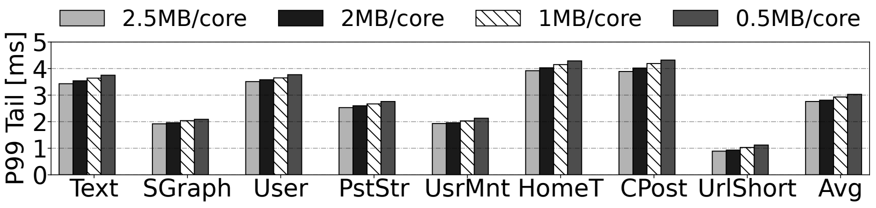


Figure 4.17: P99 tail latency of microservices running in Primary VMs with HardHarvest-Block and different LLC sizes.

4.6.10 Sensitivity to Eviction Candidate Set Size

Throughout the evaluation, we set the eviction candidate set to be 75% of all ways in a set. In this section, we perform a sensitivity study to see the impact of different sizes of the eviction candidate set on the effectiveness of HardHarvest. Figure 4.18 shows the P99 tail latency of microservices running Primary VMs in HardHarvest-Block with different sizes of the eviction candidate set. We see that, when we decrease the eviction candidate set size (to *25%* and *50%*), the tail latency increases because the algorithm is unable to preserve some shared lines. When we increase the eviction candidate set size to *100%*, the tail latency again increases because the algorithm keeps evicting needed private cache lines.

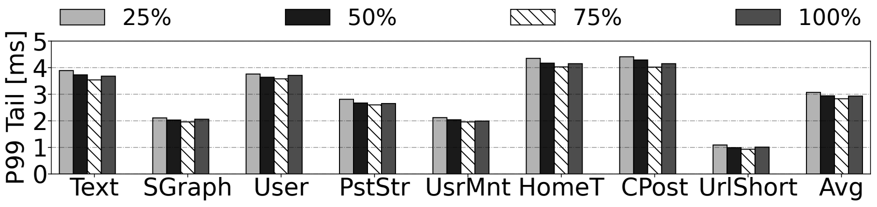


Figure 4.18: P99 tail latency of microservices in HardHarvest with different sizes of the eviction candidate set.

4.7 RELATED WORK

Resource Harvesting. Currently, spare cloud resources are harvested via software techniques [133, 134, 136, 148, 179, 180, 181, 182]. To minimize the interference on latency-critical tasks, co-located workloads can be isolated via cache partitioning and power control [145], or memory bandwidth control [146]. We show that software-only harvesting techniques introduce overheads that are not tolerable for emerging microservices workloads. HardHarvest proposes a hardware solution for core harvesting with much lower overheads.

Researchers worked on managing shared resources in SMT cores. Some proposals optimize throughput via fetch policies [183]; others enhance throughput while keeping QoS guarantees by dynamically distributing shared microarchitectural resources [184]. For security, processes from different VMs cannot run concurrently as SMT threads of the same core.

Hardware and Software for Scheduling and Context Switching. A large body of work proposed software solutions for fast scheduling and context switching [47, 48, 91, 93, 105, 106, 107, 108, 109, 110, 111, 179]. ZygOS [91] allows cores to steal requests from other cores for load balance. Shenango [47] dedicates a core for scheduling. In cloud environments, on every cross-VM context switch, these systems perform expensive cache flushes and core reassignments. HyperPlane [185] proposes a hardware solution to avoid fruitless core spinning on empty queues. However, while the core is busy, HyperPlane does not detect the arrival of higher priority requests and does not notify/interrupt the core. Thus, it cannot be directly used for core harvesting.

4.8 CONCLUSION

This chapter proposed *HardHarvest*, the first architecture for core harvesting in hardware. HardHarvest eliminates software-based overheads by using hardware request queues to speed-up core reassignment, and by partitioning private caches/TLBs while using a smart replacement algorithm. Compared to state-of-the-art core harvesting, HardHarvest increased core utilization by 1.5×, increased Harvest VM throughput by 1.8×, and reduced Primary VM's tail by 6.0×.

CHAPTER 5: Microarchitecture for Cloud-Native Services

5.1 INTRODUCTION

In serverless environments, cloud providers provision all resources and system services needed to run the users' functions. Hence, providers have the opportunity to co-locate many short-lived function containers on the same server and discard them once completed. However, in any realistic environment, these lightweight functions share the infrastructure with various other workloads, including long-running monolithic applications. As a result, providers must efficiently allocate resources across diverse workloads [136].

As indicated before, serverless workloads are a significant departure from workloads in conventional cloud environments. A typical function is *short-running* [63, 186] and its execution environment is *short-lived* [19, 63]. These properties introduce a range of challenges that undermine the efficiency of existing software and hardware. Prior work has addressed various software inefficiencies [35, 37, 38, 53, 63, 64, 86, 187, 188, 189, 190, 191, 192, 193, 194, 195]. Hardware inefficiencies have received attention [196, 197, 198], but to a more limited degree.

In this work, we observe that serverless workloads use modern processor microarchitectures inefficiently. In particular, large stateful hardware structures (caches, TLBs, and branch predictors) are poorly exploited, bringing marginal performance benefits while consuming substantial power. Indeed, frequent context switches in oversubscribed serverless environments [35, 196, 197, 198] cause functions to often interleave their executions on the cores, preventing the state in these micro-architectural structures from being reused.

Our goal is to enhance the microarchitecture of cloud servers to improve their performance for cloud-native workloads without hurting their performance for general-purpose monolithic workloads. To understand what processor microarchitecture changes would benefit cloud-native environments, we characterize serverless functions on conventional processors. Using a production workload from *Microsoft Azure*, one of the largest serverless providers, we find that executing functions with a cold *micro-architectural* state increases their response time by 4×. In addition, typical functions are short-running, have small data and instruction footprints, and execute a small number of branches.

Based on these insights, we propose *Mosaic*, a microarchitecture optimized for serverless environments. Mosaic has two components: (1) *MosaicCPU*, a processor architecture that efficiently runs both serverless and traditional workloads, and (2) *MosaicScheduler*, a software stack for serverless systems that maximizes the benefits of MosaicCPU. With Mosaic, serverless workloads efficiently share the same servers with traditional cloud workloads.

Mosaic is based on four key ideas. First, MosaicCPU slices oversized hardware structures into chunks and assigns collections of chunks called tiles to individual functions. Second, MosaicCPU assigns resources to each function based on the needs of the function. Third, MosaicScheduler uses a performance model to predict a nearly-optimal assignment of tiles to functions after profiling a few invocations of the functions. Finally, MosaicCPU and MosaicScheduler are tightly coupled: the hardware exposes its current state to the software, which uses it for off-line performance modeling and on-line state-aware scheduling.

We prototype MosaicScheduler on an Intel Sapphire Rapids system [199]. Our evaluation with production-level function invocation traces shows that MosaicScheduler reduces the functions' tail latency by 28.3%. We evaluate the combination of MosaicCPU and MosaicScheduler using full-system simulations. On average, and compared to server-class processors, Mosaic reduces the functions' tail latency by 74.6%, improves their throughput by 225%, and uses 22% less power, while adding only 0.05% area overhead. Compared to an iso-area manycore processor with many lean cores, Mosaic reduces the functions' throughput by only 13%, without slowing down monolithic applications; however, the manycore processor reduces the throughput of monolithic applications by 68%. Thus, Mosaic efficiently runs various co-located workloads, reducing the cost for cloud providers.

This chapter makes the following contributions:

- A characterization of the sensitivity of serverless functions to the sizes of the micro-architectural structures of general-purpose server-class processors.
- MosaicCPU, a general-purpose processor, highly optimized for serverless environments.
- MosaicScheduler, a software stack readily deployable on existing hardware that optimizes the execution of serverless workloads on MosaicCPU.
- An evaluation of Mosaic.

5.2 CHARACTERIZING SERVERLESS WORKLOADS ON CURRENT PROCESSORS

To understand the micro-architectural inefficiencies of hosting serverless workloads, we characterize the execution of common serverless functions on conventional processors. We run experiments on an Intel Sapphire Rapids [199] server at 3.6GHz with a 2MB 16-way per-core L2 cache and a 1.875MB 15-way per-core last level cache (LLC) slice (more details in Table 5.2). We use open-source functions [171, 200, 201, 202] and production-grade functions from *Microsoft Azure*. The evaluated functions are from different domains: image processing (*ImgProc* and *Thumbn*), video processing (*VidProc*), machine learning inference (*CnnSrv*, *RnnSrv*, *LrSrv*, grouped as *MLSrv*), data analytics (*EvStr* and *RiskQ*), document processing

(*WordCnt*) and web services (*HotelB*, *SocNet* and *WebSrv*). These functions cover popular serverless use cases [203, 204, 205]. Next, we describe our main observations.

1. What is the impact of micro-architectural state on the performance of functions? Prior work observed that serverless functions often execute on polluted micro-architectural state due to frequent context switches [35, 189] and core oversubscription [85, 206, 207]. We quantify the impact of micro-architectural state loss by measuring the execution time of functions while varying the number of functions interleaved. Figure 5.1 shows the execution time of a function when, between two of its invocations, there are 2, 4, 8, or 16 different functions executed, normalized to the execution time of the function's isolated execution. To compare with a complete loss of state, we also run the *ClearAll* environment, which flushes all cache, TLB, and branch predictor state on context switch. The figure shows a few representative functions and the average of all our functions.

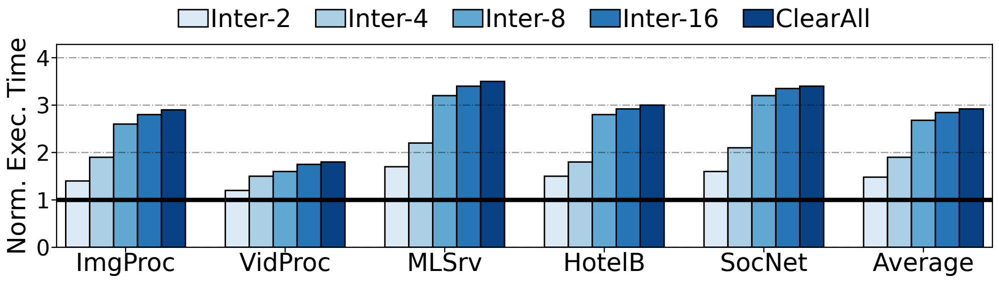


Figure 5.1: Function execution time while varying the number of functions interleaved, normalized to isolated execution.

We see that all functions degrade performance when executing on a core with polluted micro-architectural state. As the number of interleaved functions increases, the execution time gradually increases. For functions that have significant state reuse across invocations (*e.g.*, *MLSrv*), or that frequently context switch within an invocation (*e.g.*, *SocNet*), completely losing the state increases the execution time by more than 3×. On average, *ClearAll* increases the execution time by 2.9×.

We also measure function interleaving on a core in a real-world deployment at *Microsoft Azure*. Given a function f, we observe that there are at least 8 and 16 other functions interleaved on a core, for 21% and 9% of consecutive invocations of f, respectively. Note that f is not evicted from memory, but it executes with polluted micro-architectural state. Thus, preserving the micro-architectural state of functions is of great importance in real-world serverless deployments.

2. How to preserve the micro-architectural state of functions? Servers are optimized for long-running applications with large data and instruction footprints. The size of stateful structures such as caches increases with every new processor generation. However, many serverless functions have substantially different needs. In this section, we contrast serverless functions with traditional monolithic cloud applications [172].

First, we measure the applications' LLC occupancy via Intel's *pqos* tool [208]. The right part of Figure 5.2 shows the LLC occupancy of different serverless functions. Most of the functions use around 2MB. The average LLC occupancy of all our 10 functions is 2.9MB. On the other hand, the left part of Figure 5.2 shows the LLC occupancy of monolithic applications. We see that all applications use the maximum size of the LLC, which is 15MBs for our 8-slice experiments. Compared to serverless functions, monolithic applications are longer-running (a few minutes vs. a few milliseconds), have significantly larger memory footprints (10s of GBs vs. 10s-100s of MBs), and occupy the whole LLC and could benefit from even larger caching space (15MB vs. 2.9MB).

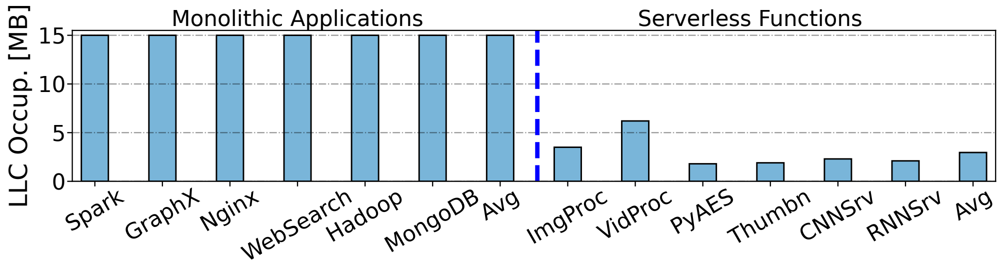


Figure 5.2: Last Level Cache (LLC) occupancy for long-running monolithic applications and serverless functions.

The small LLC occupancy of serverless functions indicates that they can execute with a reduced cache capacity and still perform well. Hence, we use Intel's CAT [155] to execute functions and monolithic applications with different numbers of LLC ways per slice: 15, 10, 5 or 2 LLC ways per slice. Figure 5.3 shows the resulting execution time of monolithic applications (left) and functions (right). The bars are normalized to 15 ways per slice, which is the default design. We see that monolithic applications experience a severe increase in execution time if they run on smaller caches. For example, *DataSrv* increases its execution time by 50% when using 2 LLC ways rather than the default 15 ways. In contrast, the right part of Figure5.3 shows that all functions barely change their execution time even when running with only 2 LLC ways. On average, using 2 instead of 15 ways per LLC slice increases the functions' execution time by only 2.8%.

We now reduce the size of both the L2 cache and the LLC. Figure 5.4 shows the execution time of a representative function, *ImgProc*, with different numbers of ways in the L2 and in the LLC slice. The groups of bars correspond to different LLC slice sizes and, within a group, there are bars for different L2 sizes. All bars are normalized to the case of full L2 and

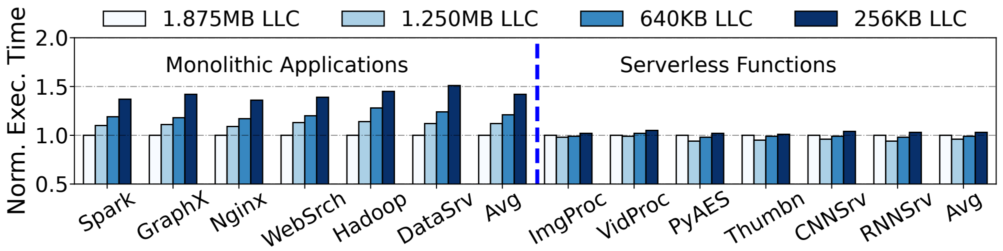


Figure 5.3: Normalized execution time of monolithic applications and serverless functions executed with different LLC slice sizes.

LLC slice size. From the figure, we see that even with a modest number of ways in the L2 and in the LLC slice, the execution time of the function does not increase much. Therefore, serverless functions can still run efficiently with relatively small L2 and LLC caches.

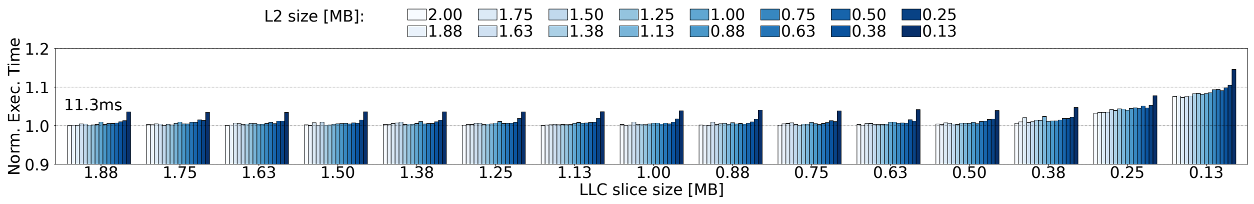


Figure 5.4: Normalized execution time of the *ImgProc* serverless function with different sizes of L2 and LLC slice. The number on top of the leftmost bar is the execution time with full L2 and LLC slice sizes.

Finally, we collect instruction traces with Pin [100] and simulate different sizes of branch predictors with the SST simulator described in Section 5.4. Our baseline architecture of Section 5.4 has a 32KB TAGE-SC-L [209] branch predictor and an 8K-entry branch target buffer. Figure 5.5 shows the hit rate of the branch predictor and branch target buffer as we reduce the size of these structures for the *ImgProc* function. We consider structures with a fraction of the entries in the baseline structures and normalize the hit rates to that of baseline structures. We can see that 32× smaller branch predictor table and BTB reduce the hit rates by only 0.9% and 3.9%, respectively.

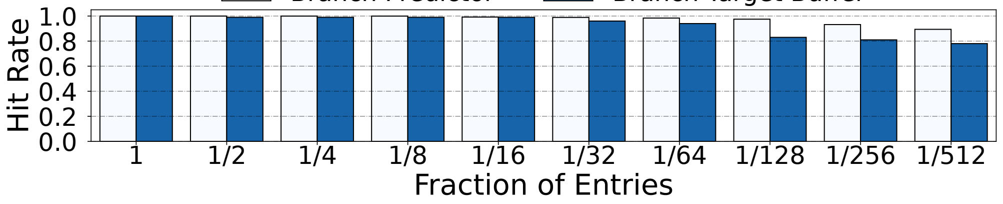


Figure 5.5: Normalized hit rate of the branch predictor table and branch target buffer as we reduce the number of entries in the structures for the *ImgProc* serverless function.

In Mosaic, we exploit the small footprints of functions to partition structures and preserve the functions' state in their partition. Mosaic overcomes the inefficiencies of existing partitioning schemes (*e.g.*, Intel CAT [155]). Such schemes partition only some levels of the cache hierarchy, rather than additionally partitioning stateful structures (*e.g.*, branch target buffer). Moreover, they induce non-negligible ms-scale overheads when they change the core's allocation policy. Finally, they support only a small number of classes of service, limiting the number of concurrently stored states on the server.

3. Why maintain processor generality? Instead of creating a specialized core/accelerator for serverless workloads [210], Mosaic aims to maintain processor generality and introduce modest changes that allow a general-purpose server-class CPU to efficiently run both traditional and serverless workloads. There are three reasons for this decision: (1) handling inter-function heterogeneity, (2) reducing the provider's TCO, and (3) accommodating end-to-end cloud workflows.

First, although many functions execute acceptably in low-performance cores (*i.e.*, cores with small hardware structures, low frequency, and low issue width), some functions benefit from executing in high-performance server-class cores. We simulate the execution of our functions on a beefy core modeled after Intel's Sapphire Rapids [199] at 3.6GHz and on a small core modeled after ARM's A15 [211] at 2.5GHz. It can be shown that while WebSrv and SocNet see minimal performance degradation, MLSrv and ImgProc increase the response time by more than 47%. Some heterogeneous platforms such as ARM's big.LITTLE[212] include both high-performance beefy cores and energy-efficient small cores. However, they have a fixed number of each core type and cannot dynamically adapt to the workload.

Second, serverless workloads are only a fraction of the workloads in the cloud, and often share the same server with monolithic applications that require large cores [136]. Creating a separate cluster dedicated to serverless workloads substantially increases the TCO for providers due to the introduced fragmentation (as shown in Figure 5.22).

Finally, end-to-end serverless applications are often composed of both serverless functions and monolithic services. As an example, many serverless functions [213] use databases such as MongoDB [214] as their backends. For high performance, these different pieces of an application should run close to each other—ideally, on the same physical server.

4. What is the heterogeneity across functions? Different functions can have very different hardware requirements. As an example, Figure 5.6 shows the execution time of three functions, *RnnSrv*, *ImgProc* and *WordCnt*, when executing with different numbers of L2 ways. The execution time is normalized to the one with all 16 ways. We see that *RnnSrv* is highly sensitive to L2 size, and needs at least eight L2 ways, while *ImgProc* is modestly sensitive to L2 size and needs at least two L2 ways, and *WordCnt* is insensitive to L2 size.

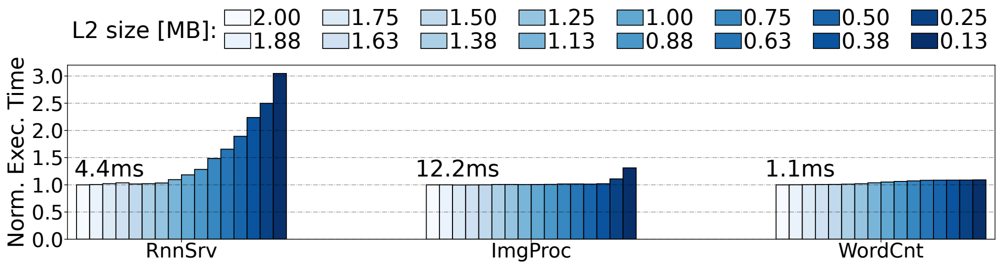


Figure 5.6: Normalized execution time of functions executing on a core with a single LLC way per slice and different L2 sizes. The numbers on top of the bars are the execution times with a full L2 size.

To generalize a function's needs, we categorize functions into *Low*, *Medium*, and *High* intensity for data, instruction, and branches. These categories are determined by the function's data working set size, instruction working set size, and branch working set size (*i.e.*, the working set size of the cache lines that contain branches), respectively. Table 5.1 shows the categorization of some of the functions we use. Functions that fall in the same bucket typically need the same hardware structure size for optimal cost-performance.

Table 5.1: Categorization of functions into low, medium, and high intensity for data, instructions, and branches.

Function	Data	Instructions	Branches
RiskQ	Low	Low	Low
EvStr	Low	Low	Medium
WordCnt	Low	Low	Medium
ImgProc	Medium	High	High
MLSrv	High	Medium	High
HotelB	Low	Low	Low
SocNet	Low	Low	Low
WebSrv	Medium	Low	Low

We use this categorization to classify a subset of popular production-level functions at *Microsoft Azure*. The majority of the evaluated functions have *Low* data intensity with an average data working set size of 2MB, *Medium* instruction intensity with an average instruction working set size of 12MB, and *Medium* branch intensity with an average branch working set size of 1.5MB. For comparison, monolithic applications [172] have 8GB, 0.7GB, and 0.3GB average data, instruction, and branch working set sizes, respectively.

5.3 MOSAIC: SERVER ARCHITECTURE CO-DESIGN FOR SERVERLESS FUNCTIONS

To address the observed inefficiencies, we introduce *Mosaic*, a system optimized for serverless environments. Mosaic has two components: (1) *MosaicCPU*, a processor architecture that efficiently runs both monolithic applications and serverless functions, and (2) *Mosaic-Scheduler*, a software stack for serverless systems that maximizes the benefits of MosaicCPU.

Mosaic materializes the main insights of our characterization via four principles. First, MosaicCPU slices oversized hardware structures into fine-grained chunks and assigns collections of chunks called *Tiles* to individual functions. Each tile preserves the state of a function in the structure, maximizing the opportunities for state reuse across context switches and minimizing the interference between co-located functions. Second, MosaicCPU assigns a different tile size to each function based on the needs of the function. A tile spans non-contiguous entries in a structure. Third, MosaicScheduler uses a performance model to predict a nearly-optimal assignment of tiles to functions after profiling a few invocations of the functions. Profiling is performed online, while performance modeling is performed offline, outside of the functions' critical path. Finally, MosaicCPU and MosaicScheduler are tightly coupled: the hardware exposes its current state to the software, which uses it for off-line performance modeling and on-line micro-architectural state-aware scheduling. Next, we detail each of the principles.

5.3.1 Fine-grained Per-Function Hardware Partitioning

1. Overview. In current processors, a function has access to the entirety of stateful structures of the core it is running on. When a function runs, it displaces the state of the functions that were running on the core before, preventing them from reusing the loaded micro-architectural state after they resume execution on the core. MosaicCPU overcomes this challenge by slicing large stateful hardware structures such as caches, TLBs, and branch predictor units into smaller physical partitions called *Chunks*, and dedicating a group of chunks called a *Tile* to individual functions. Figure 5.7 shows the high-level overview of the MosaicCPU architecture and its partitioning technique.

Hardware structures are partitioned into fixed-sized chunks of the same associativity. A chunk comprises a few physically contiguous sets in a given hardware structure. In principle, Mosaic can partition any stateful hardware structure. However, to minimize complexity and avoid potentially increased access latencies, Mosaic targets the most oversized structures whose state saving brings the most performance benefits and whose access latencies can be

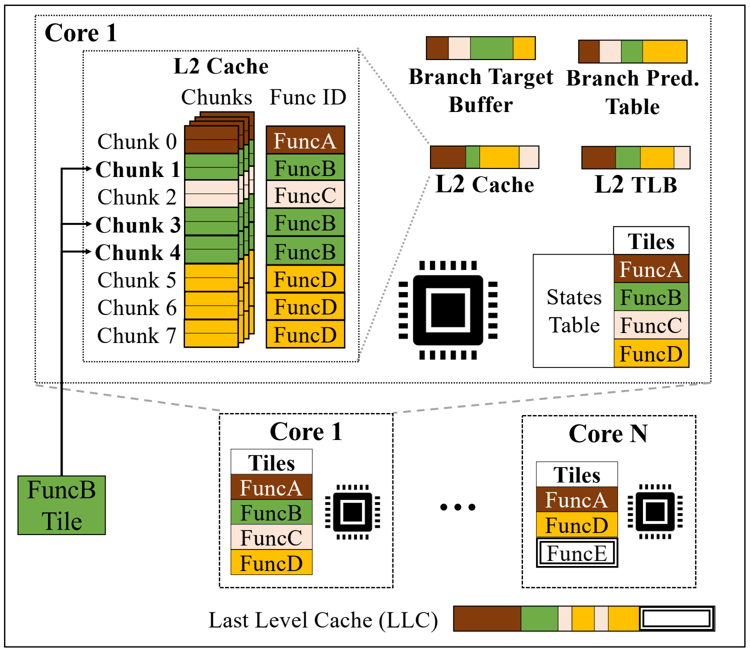


Figure 5.7: High-level overview of the MosaicCPU architecture.

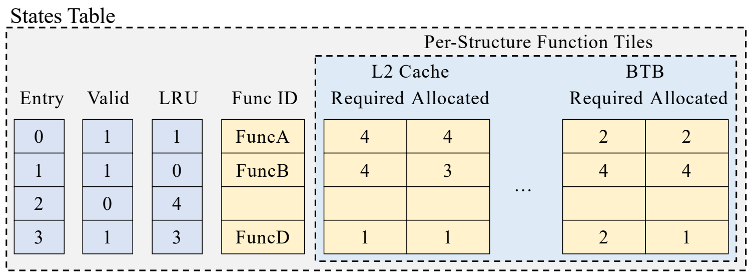
hidden. Specifically, it targets the following 5 structures: L2 cache, L2 TLB, LLC, branch target buffer, and branch predictor table.

A function is assigned a group of chunks called a tile in each of the partitioned structures. The chunks in a tile are not necessarily contiguous. A tile preserves the micro-architectural state of a function for future reuse. For example, assume that $FuncA$ yields a core to $FuncB$. To preserve $FuncA$'s state, MosaicCPU restricts $FuncB$ to access entries only within its tile, keeping the tile of $FuncA$ uncontaminated. During $FuncB$'s execution, MosaicCPU translates the accesses of $FuncB$ to the correct physical chunks.

Each partitioned structure has tags indicating which function owns which chunk. The chunks with state for functions that are not currently running are placed in a low power mode. The goal is to save power while still retaining the state in the chunks. On a context switch, the chunks of the pre-empted function are put in low-power mode and the chunks of the new function are activated.

A MosaicCPU has a hardware *States Table* that tracks which functions currently hold their state in each of the core's structures. As shown in Figure 5.8, the States Table contains as many entries as the maximum number of functions that can concurrently hold state in the core structures. A given entry contains the function ID and, for each of the partitioned structures, the number of chunks currently assigned to the function and the number of chunks that the function requires to run efficiently based on the MosaicScheduler prediction.

In addition, the States Table entries have LRU bits. When a new function needs additional chunks, they are taken from the LRU function. The OS keeps one chunk in each structure.


States Table
Per-Structure Function Tiles

				L2 Cache			BTB	
Entry	Valid	LRU	Func ID	Required	Allocated		Required	Allocated
0	1	1	FuncA	4	4		2	2
1	1	0	FuncB	4	3	...	4	4
2	0	4						
3	1	3	FuncD	1	1		2	1

Figure 5.8: The *States Table* in MosaicCPU tracks which functions currently keep their state in the core's structures.

2. Assigning tiles and chunks. MosaicCPU includes mechanisms to assign and deassign chunks to/from functions. When a core schedules a function invocation for execution, the OS first checks the States Table to see if the function has the required number of chunks in all the core's structures. If so, we call this a *hit*. In this case, for each structure, the OS sets the chunks of the previously-running function to low-power mode and activates the chunks of the new function. The LRU bits in the States Table are updated.

If, instead, the new function is not in the States Table or it is there but does not have all the required chunks in all the structures, a *miss* occurs. If the function is not in the States Table, the OS allocates an entry for the function in the table, which includes filling in the Function ID and the required chunks in each of the structures. Further, in all the miss cases, the OS computes the number of chunks that the new function is missing in each structure, then harvests such number of chunks either from unused chunks or from chunks owned by the LRU function or functions, and finally reassigns these victim chunks to the new function. This process involves updating the States Table and then, for each of the structures, reassign the victim chunks. The latter consists of activating the victim chunks, writing back and invalidating their entries, and updating the function ID tags of such chunks in the structure. Writebacks are only needed for the dirty lines in the caches and the updated state bits in the TLB.

After this, the same operations as a hit occur: the chunks of the previously-running function are put in low-power mode, all the chunks of the new function are activated, and the LRU bits in the States Table are updated.

For the state in Figure 5.8, if the core attempts to execute $FuncA$, a hit will occur. However, if it tries to execute $FuncB$, $FuncD$, or a new function $FuncE$, a miss will occur.

We model the write back overheads in our evaluation and see that they are tolerable: the average overhead of writing back a chunk is 500-600ns, while the highest overhead is 1.5-2 μs.

In a secure cloud environment, the whole cache hierarchy is flushed at a context switch [215, 216], causing overheads of a few ms, while offering isolation equivalent to Mosaic's.

3. Memory de-duplication. Currently, Mosaic does not allow memory de-duplication across users, following security guidance from state-of-the-practice [217] and state-of-the-art [218]. Mosaic can be extended to support deduplication and reduce the total memory footprint of the server, while potentially sacrificing security. In such an environment, shared pages (*e.g.*, libraries and other code) can be marked as read-only and shared in the page tables, and placed in a separate *shared-page* tile. This tile is always active and shared across all functions. When a core issues a request for a shared page (indicated by a bit in the TLB), the structures search the shared-page tile. Otherwise, they search the function's tile.

4. Fine-grained power setting. Mosaic sets the voltage of idle chunks to a lower value than the active chunks, so that the static power of idle chunks is minimized. This is accomplished by having two voltage rails, Vdd$_{high}$ and Vdd$_{low}$, and selecting the rail for each chunk based on an Active bit. As a result, Mosaic does not need a voltage regulator per chunk. The simple logic used is shown in Figure 5.9.

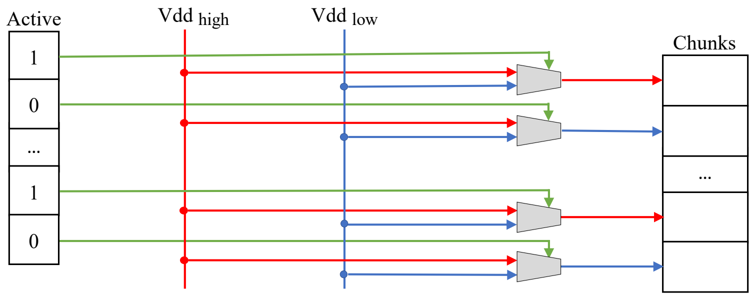


Figure 5.9: Per-chunk voltage selection in Mosaic.

A similar approach has been implemented in prior work, with different voltage sources per section of the cache [219] or even per cache line [220]. Switching the voltage for a chunk has low overhead. The transition time for a drowsy cache line is 1-2 cycles [220], while the transition for the whole core is in the order of 1-2 μs [221]. In Mosaic, the chunks transition between voltage settings only on a context switch; hence the overhead is negligible. To reduce the overhead even further, we can predict when a chunk will need to change rails and perform the change in advance in the background.

5. Maintaining cache coherence. As in conventional systems, different functions communicate with each other via RPC messages and not via shared memory [31]. However, Mosaic caches still need to support data sharing in situations such as multi-threaded functions, concurrent invocations of the same function, and migration of invocations across cores. Thus, Mosaic caches are coherent. To achieve so, Mosaic ensures that snooping hardware can snoop chunks that are in low-power mode [220, 221]. Coherence messages and responses in Mosaic include the FuncID, so that the controller of the receiver cache can use the FuncID to identify the correct destination chunk (Section III.B4). Another possible approach would be to keep an up-to-date translation between core ID and running FuncID for all the cores in the snooping hardware of all the caches. In this case, the receiver cache would identify the correct FuncID based on the ID of the core that sent the coherence message.

5.3.2 Accessing a Function's Tile

1. Translation process. Since functions are heterogeneous, having a uniform tile size for all functions and in all structures is inefficient. Instead, Mosaic sizes the tile of each function in each structure differently, based on the requirements of the individual function. For instance, data-intensive functions may get larger tiles (*i.e.*, tiles with more chunks) in data caches, while branch-intensive functions may get larger tiles in branch prediction units. Allowing non-uniformly sized tiles may fragment the structures. Hence, to avoid fragmentation, function's tile is allowed to span physically non-contiguous chunks in a given structure.

Accessing the tile of a function in a hardware structure requires a level of indirection: the function's addresses need to be mapped to the correct positions in the structures. Consider the case of a cache. Recall that, in a conventional cache, the hardware splits the address into *tag*, *index*, and *offset* bits. In Mosaic, the *index* bits are separated into two parts: if chunks have S sets, then the $\log S$ least significant bits specify the set within a chunk (*Set* bits), and the rest of the bits specify the chunk (*Chunk* bits). This is shown in the upper part of Figure 5.10. The figure shows a 2MB L2 cache organized into 2K sets of 16 ways. Mosaic splits the cache into 16 chunks of 128 sets and 16 ways. For simplicity, we only show 4 ways per set and 4 sets per chunk.

Most of the time, a function's tile will not cover the whole structure and, instead, will have a number of chunks C lower than the maximum number of chunks C_0. In our design, S and C are powers of two. Let us call *index* the value of the index bits. Then, the hardware computes the ID of the chunk of the function that needs to access as $(index/S)\%C$. This operation is performed as simple bitwise shift operations.

Mosaic enables such operation by the *Mask* register shown in Figure 5.10. Mask has its $\log C$ least significant bits set to 1 and the rest to 0. Mask masks out some of the most significant bits (MSBs) of the Chunk field. In the example, assume that C=8. In this case, Mask masks out the MSB of Chunk as it generates the chunk ID.

To identify the desired chunk in the structure, Mosaic tags each chunk with the function

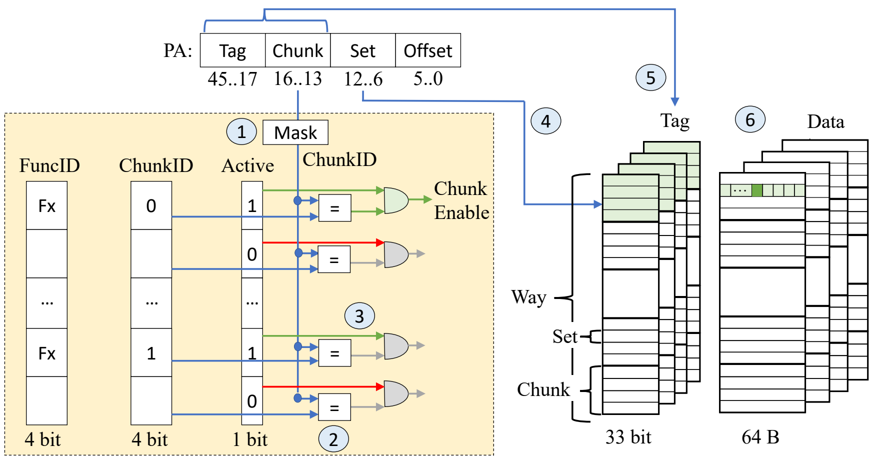


Figure 5.10: Detailed micro-architecture of Mosaic's L2 cache.

ID and the ID of the chunk in the function (*i.e.*, the chunk's logical index within the tile). In addition, there is an Active bit per chunk that is set only for the chunks of the currently running function. This is shown in Figure 5.10, which assumes that function *Fx* is running. With this support, the output of the Mask register is compared to the ChunkID and Active bits of all the chunks to determine the target chunk.

2. Example operation. We showcase the access to two representative partitioned structures: L2 cache and branch predictor. The other structures, namely TLB, BTB, and LLC, follow the same principles.

L2 cache. Figure 5.10 shows how the L2 cache is accessed. We assume that a core issues a 46-bit physical address (PA). Moreover, as multiple logical chunks may map to the same physical chunk, Mosaic uses the combination of *Tag* and *Chunk* bits as cache tag.

The whole translation is as follows. Mosaic takes the *Chunk* bits and uses *Mask* to generate the chunk ID bits (1). These bits are compared to the *ChunkID* field in all 16 L2 chunks (2). As there can be multiple chunks tagged with the same ChunkID but owned by different functions, Mosaic checks the ownership of the chunk (3). There are two ways to implement this functionality: use the chunks' Active bits or compare the FuncID tag with the ID of the currently-running function. Mosaic uses the former approach for single-threaded cores (as shown in the figure), and the latter approach for SMT cores and coherence messages. After choosing the correct chunk, Mosaic uses the *set* bits from the PA to select the set in the chunk (4). Then, it compares the tags of all the ways of the set with the *tag* and *chunk* bits from the PA (5). Finally, Mosaic reads at the offset determined by the PA's *offset* bits (6), as in conventional caches.

Branch predictor. We model a state-of-the-art 32KB TAGE-SC-L branch predictor [209].

The predictor is composed of a base predictor (2-bit counter bimodal table T0) and 15 tagged tables with different history lengths and number of entries per table. The base predictor is directly indexed using the program counter (PC), while the tagged predictors are indexed using a hash of PC and a subset of history length.

Mosaic does not change the predictor's functionality. It only modifies the way the predictor tables are accessed, as shown in Figure 5.11. Specifically, each table (T0 to T15) is split into 16 chunks (for simplicity, the figure shows only three tables). The sizes of the chunks are different in different tables (as the total number of entries differs across tables), and a function is assigned the same number of chunks and the same chunk IDs in all the tables. Thus, the tables share the translation mechanism, Mask, and ChunkID/Active tags. After computing the hashes for all table accesses (a), Mosaic breaks the table index into *chunk* and *set* bits. It translates the chunk bits using the same principle as for the L2 cache (b). Then, it uses the translated *chunk* and the *set* bits to access the correct entry in the table (c). Beyond this, Mosaic does not change the baseline functionality of the branch predictor. Each table makes its own prediction, and the table indexed with the longest history and a tag match produces the final prediction (d). As in L2 cache, the tables are tagged with *tag* and *chunk* bits.

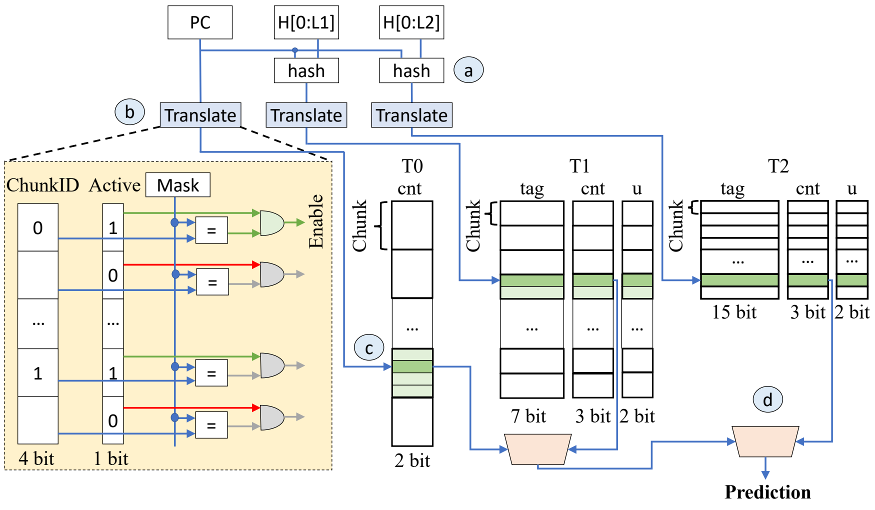


Figure 5.11: Mosaic's branch predictor based on state-of-the-art TAGE-SC-L [209].

Overall, the process of translating addresses for the partitioned structures is simple. It involves only comparing for equality the four bits of *ChunkID* with the PA's masked *chunk* bits, and an AND-gate with the *Active* bit. This takes about one processor cycle.

3. Minimizing function tags overheads. Mosaic software assigns a 64B FuncID to each function—*e.g.*, a hash of the function name and the user ID [63]. Tagging all the chunks of this function with this FuncID in all the partitioned structures is a non-negligible storage overhead. To reduce this overhead, MosaicCPU exploits the fact that only functions that have an entry in the States Table of a core can have an allocated chunk in any of the partitioned structures of that core. Hence, MosaicCPU uses a function's *entry number* in the States Table as its unique ID. For example, in the system of Figure 5.8, the chunks of $FuncA$ and $FuncB$ are tagged with 0 and 1, respectively. This approach saves substantial space in the partitioned structures.

This improved design requires snooping caches to keep a small table that maps FuncIDs to States Table entry numbers. When an incoming coherence message is received, its FuncID is translated into an entry number using this table before checking against the tags of the chunks in the cache.

5.3.3 Tile Sizing for the Functions

1. Overview. To pack the state of many functions in a core, Mosaic tries to allocate, for each function, the minimum number of chunks in each partitioned structure that still deliver acceptable performance for the function. A function execution completes in C_{full} cycles on a core that does not partition structures. Then, Mosaic searches for an assignment of chunks to the function that uses the minimal amount of resources and still completes execution in $C_{par} = C_{full} \times (1 + Threshold)$ cycles, where $Threshold$ is a small value like 0.1.

Determining the optimal tile sizes for a function through exhaustive search is impractical, given the large exploration space. One would need to execute the function with all possible combinations of tile sizes in all of the partitioned hardware structures, resulting in a few thousand invocations.

To address this limitation, Mosaic uses MosaicScheduler to profile a few invocations of a function online with live traffic, and then create a performance model offline to use for predictions. Moreover, to reduce profiling overheads, MosaicScheduler also uses *Transfer ML* to predict a function's optimal tile sizes based on previously-profiled similar functions.

2. MosaicScheduler. We consider the two prediction methods.

Predictions via performance modeling. A function is initially profiled with a few configurations. During the execution with each configuration, MosaicScheduler collects the number of cycles, the IPC, and the misses in each of the five partitioned structures. Then, MosaicScheduler picks the configuration with the smallest tiles that still finishes execution within the required deadline as the *temporarily optimal function size.* Finally, offline, MosaicScheduler takes this configuration and predicts if any of its tiles can be further reduced.

For this prediction, MosaicScheduler considers each partitioned structure in turn. For each one, it examines the trend of misses as the size of the tile in the structure decreases and, using the expected penalty of a miss in that structure, estimates the function execution time if the tile in the structure is reduced by one more notch. For example, as it considers the BTB, MosaicScheduler observes the trend of miss increases with smaller tile sizes in the BTB, predicts the extra number of misses that will occur if the BTB tile size is reduced one more notch and, given the cost of a miss, estimates the overall function execution time with the smaller BTB tile size. If the longer function execution time is acceptable, MosaicScheduler reduces the function's tile size in the BTB.

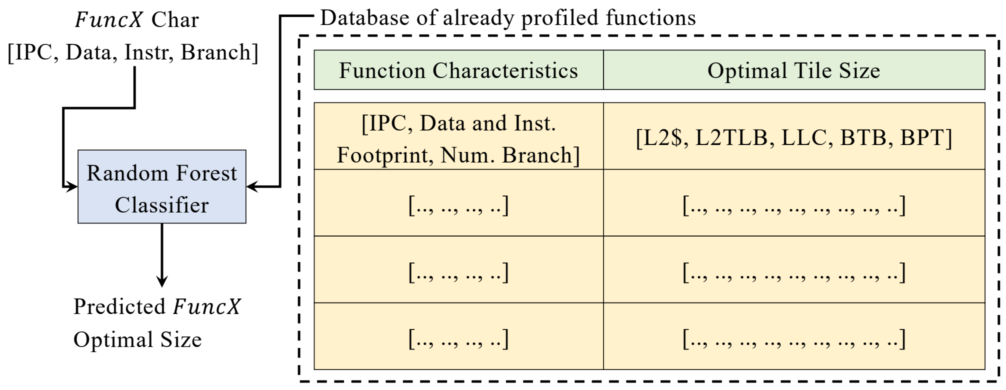
If there are multiple equally-desirable options, *e.g.*, the size of either the L2 cache or the BTB can be reduced, MosaicScheduler stores these options as function alternatives. Different alternatives can be used at different times depending on which other functions are running concurrently and sharing the structures.

All future invocations of a function execute with the predicted tile sizes. Mosaic does not change a function's tile sizes during an invocation of the function, but different invocations of the same function can execute with different configurations over time. MosaicScheduler monitors the execution and may recompute the function's tile sizes if the execution time is too long or too short.

This model yields high accuracy with low overhead. Typically, MosaicScheduler needs only 8-10 function invocations to accurately set the optimal sizes of a function's tiles.

Predictions via transfer ML. To minimize the profiling overheads, MosaicScheduler also uses an approach based on transfer ML. Instead of repeated profiling to establish the best tile sizes for a function, MosaicScheduler takes some high-level characteristics of the function (data and instruction footprint, number of branch instructions, and IPC) and compares them to the characteristics of some already-profiled functions. Then, with a regression model, MosaicScheduler finds the most similar functions and predicts the optimal sizes of the tiles for the new function based on these similarities. To achieve high accuracy, our prediction system needs only about 10 functions of different properties to be initially profiled. Note that the regression model is periodically augmented and retrained with new functions as they execute in the system.

Figure 5.12 shows the approach. A random forest classifier using a database of already profiled functions takes the characteristics of a function (*FuncX* in the figure) and generates the optimal sizes of the tiles for the function. We implement the classifier in Python's scikit-learn library [222]. As the model outputs the tile size for each of the partitioned structures, it is wrapped around a MultiOutputClassifier [223]. The model uses 100 trees, and the minimum number of samples required to split an internal node is 2. The size of a tile is classified into one of five classes: 1, 2, 4, 8, or 16 chunks. The prediction is done in software and off the critical path of function execution.



FuncX Char
[IPC, Data, Instr, Branch]
Database of already profiled functions
Function Characteristics
Optimal Tile Size
[IPC, Data and Inst. Footprint, Num. Branch]
[L2$, L2TLB, LLC, BTB, BPT]
[.., .., .., ..]
[.., .., .., .., .., .., .., .., ..]
Random Forest Classifier
Predicted *FuncX* Optimal Size

Figure 5.12: Transfer ML architecture used to predict the optimal size of new functions.

5.3.4 Scheduling Function Invocations

1. Overview. In conventional serverless frameworks, function invocations are scheduled on a random core or on the least-loaded core [224]. This approach would diminish the benefits of Mosaic. Instead, to maximize performance, MosaicScheduler is aware of the state in the cores when scheduling invocations. It uses the interface exposed by the hardware and a set of heuristics to decide on which core to place an invocation.

2. Scheduling invocations. Figure 5.13 shows how MosaicScheduler schedules function invocations on cores. When a function invocation (such as the one for *FuncX* in the figure) arrives at the server, MosaicScheduler checks the predictor for the function's optimal tile sizes. Based on this information and the state of the cores, MosaicScheduler picks one core to execute the invocation. Then, it deposits the invocation augmented with the tile size information on the software request queue of that core.

As shown in Figure 5.13, MosaicScheduler reads the state of the States Tables of the cores to make its decision on where to schedule the invocation. Generally, it favors a core that already holds state for the function. Specifically, if there is one or multiple cores with function state that are idle, the scheduler randomly picks one of them. If there are no cores with function state, the scheduler picks a core to balance load across cores.

If all cores with function state are busy, the scheduler predicts in which case the invocation execution will complete sooner: 1) if it waits in one of these cores until it can execute and reuse the state or 2) if is assigned to another, possibly idle core and can start executing sooner. MosaicScheduler predicts the waiting time in the queues and the execution time of

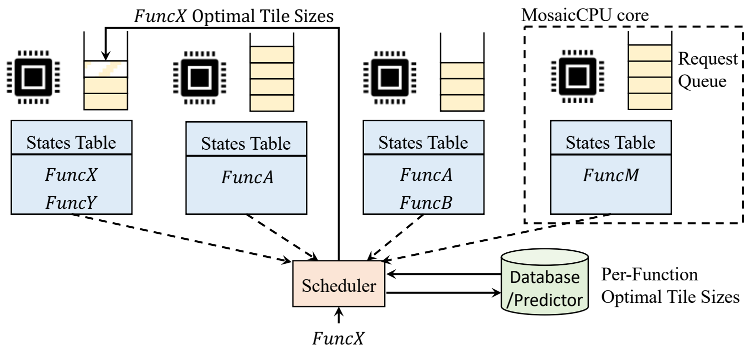


Figure 5.13: Scheduling a function invocation in Mosaic.

the function invocation based on the profiles of the functions. MosaicScheduler then picks the core that would minimize the sum of waiting time and processing time for the invocation.

To deal with occasional mispredictions, the system also uses work-stealing. When a core becomes idle, its worker thread periodically checks if there are cores with queued up invocations. If so, the thread fetches invocations from other cores and executes them locally. In this way, the system is robust to head-of-the-line blocking.

3. Advanced scheduling policies. A scheduler that is unaware of the resource needs of functions may co-locate functions that require intense use of the same resource on the same core. Such core would then suffer high contention on one resource while the other resources would be underutilized. This problem is not present in Mosaic. MosaicScheduler balances resource utilization by spreading functions with similar resource requirements across the cores of the server. Specifically, when scheduling a function invocation that has no state in any of the cores, the scheduler checks which core has enough idle chunks to satisfy the invocation's predicted needs or which core would observe the least number of chunk evictions. This approach could be generalized for function placement across servers in a cluster. The cluster controller could classify functions as cache, TLB, or branch intensive and place only functions with distinct requirements on the same server.

4. Non-serverless workloads. Mosaic cores operate in two modes: serverless and non-serverless. These modes are activated by setting the *FaaSMode* register. In the serverless mode, all Mosaic optimizations are enabled. Conversely, in the non-serverless mode, the cores operate in a conventional manner, without structure partitioning. This enables high-performance execution for monolithic applications. Privileged software such as the Virtual Machine Manager (VMM) sets the FaaSMode register on a cross-VM context switch. When changing the FaaSMode register, the state of all the partitionable structures is written back and invalidated. Hence, the VMM tries to schedule non-serverless workloads on cores that are already in non-serverless mode. The FaaS platform controllers [57, 60], including

MosaicScheduler, run on one or more dedicated cores in non-serverless mode.

5. Harvest VMs. To reduce cost, some serverless environments use harvest VMs [136]. A harvest VM is created with a minimal number of cores, but it can dynamically grow and shrink by harvesting idle cores and releasing them when they are needed by higher-priority VMs (called primary VMs). Primary VMs run latency-sensitive workloads, while harvest VMs run workloads that can tolerate resource fluctuation, including serverless functions. Mosaic can run in such environments. When a core context switches between a primary and a harvest VM, the VMM changes the FaaSMode register of that core.

6. Mosaic beyond serverless workloads. While our focus is on serverless functions, Mosaic can offer benefits to other workloads that also exhibit frequent context switches, small working sets, and short execution times. One example of such workloads is microservices, which are used in many cloud deployments. Recent studies from Google [225] and Alibaba [5] show that microservices often require numerous RPC invocations, leading to frequent context switches that challenge traditional execution environments. For example, an individual microservice may issue hundreds of RPC calls. Mosaic can preserve the micro-architectural state across context switches, thereby enhancing microservice performance.

5.4 METHODOLOGY

1. Systems modeled. Our base architecture is a server with 16 cores and 128GB of main memory. Each core is a 6-issue processor modeled after Golden Cove micro-architecture in Intel Sapphire Rapids (SPR) [199]. Table 5.2 shows the micro-architectural parameters.

Table 5.2: Architectural parameters used in the evaluation.

Processor Parameters	
Multi-core chip	16 6-issue OoO cores, 512-entry ROB, 3.6GHz
L1 data cache	48KB, 8-way, 4 cycles round trip (RT), 64B line
L1 instruction cache	32KB, 8-way, 4 cycles round trip (RT), 64B line
L2 cache	2MB, 16-way, 16 cycles RT, 30 MSHRs
L3 shared cache	Slice: 1.8MB, 15-way, 60 cycles RT, 30 MSHRs
L1 data TLB	256 entries, 4-way, 2 cycles RT
L1 instruction TLB	256 entries, 4-way, 2 cycles RT
L2 TLB	2048 entries, 8-way, 12 cycles RT
Branch predictor	32KB TAGE-SC-L [209], 15 cyc. mispred. penalty
Branch target buffer	12K-entry, 4-way
Main-Memory Parameters	
Capacity;	128GB
Frequency; Rate	1GHz; DDR

We use this base architecture to evaluate six server systems. (1) *Baseline* is a conventional server that does not partition hardware structures and schedules function invocations on the least-loaded cores. (2) *Baseline+Affinity* augments Baseline with simple affinity scheduling: a function invocation is scheduled on the core that last executed the same function if the core is idle. (3) *Baseline+MosaicScheduler* runs the software support of Mosaic on conventional hardware while partitioning L2 and L3 caches using Intel CAT [155]. This system maintains a *States Table* per core in software, and does not require any non-conventional hardware support. (4) *Mosaic* is our complete Mosaic design. (5) *Jukebox* [197] is a recent hardware proposal for serverless environments that enhances Baseline with hardware-supported instruction prefetching. (6) *Manycore* is a conventional server with 128 simple cores similar to ARM Cortex A15 [211] that has the same area as Baseline.

We evaluate the architectures with full-system simulations. We use the QEMU [156] emulator integrated with a modified SST framework [98] and DRAM-Sim2 [99] memory simulator. In this way, we simulate the operating system (Ubuntu 20.04), the serverless software stack, and our benchmark functions. To validate our simulation accuracy, we also run Baseline, Baseline+Affinity, and Baseline+MosaicScheduler on the SPR server used in Section 5.2 and calibrated the simulator. With L2 and L3 cache partitioning with Intel CAT [155], the server has 8 classes of service (COS).

Our simulation infrastructure models the two main Mosaic overheads. On a chunk eviction, we model efficient hardware that walks the tags of the chunk writing back all dirty entries and then invalidates all entries. Further, on changing the voltage rail of a chunk, we add a fixed 5-cycle latency. To model the area and power overheads, we use McPAT [102] and CACTI [101]. Recall that Mosaic increases the tag width of the partitioned structures by 4 bits. The increased tag size does not affect the access latency and only marginally increases the per-access energy. Mosaic further adds a hardware States Table per core, per-chunk ChunkID, FuncID, and Active bits in the partitioned structures, and some other small structures. Overall, this hardware adds 43.5KB of storage per core, which is 1.06% of the total core storage and 0.42% of the core area.

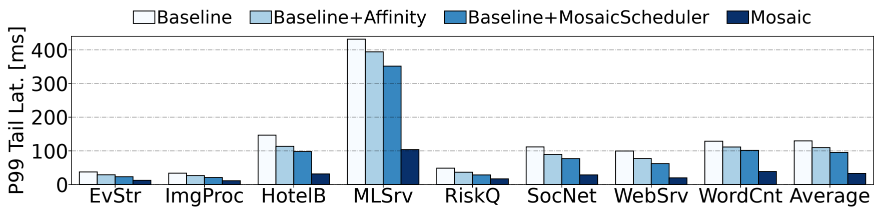
2. Workloads. We execute eight open-source serverless functions described in Section 5.2: *ImgProc*, *MLSrv*, *EvStr*, *RiskQ*, *WordCnt*, *HotelB*, *SocNet*, and *WebSrv*. We invoke the functions with real-world invocation traces from *Microsoft Azure*. The traces cover peak-load and include 71,434 invocations of 64 different functions. We randomly map each production function to one of the eight evaluated functions. When multiple production functions map to the same evaluated function, we create separate instances for such functions, which do not share any data or instructions.

In some experiments, we evaluate monolithic applications from CloudSuite [172] which span multiple domains: data processing (*Spark* and *Hadoop*), graph applications (*GraphX*), web frontends (*Nginx*), web search engines (*WebSrch*), and data serving (*DataSrv*).

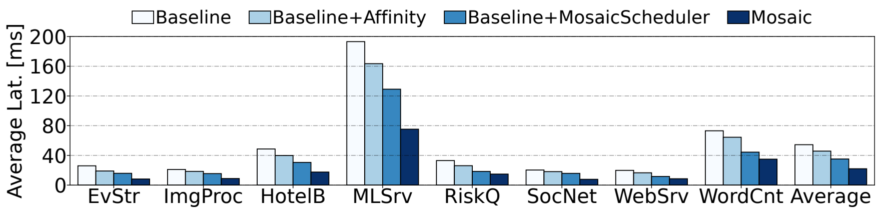
5.5 EVALUATION

5.5.1 Performance Improvements

We measure the end-to-end latency and throughput of the function invocations on the evaluated systems. The end-to-end latency of a function invocation is the time from when the client sends a request until when it receives the result.



(a) Tail latency



(b) Average latency

Figure 5.14: Tail and average latency of the function invocations.

1. Tail latency. Figure 5.14a shows the P99 tail latency of our functions in four architectures. We see that, on average, adding affinity scheduling with Baseline+Affinity reduces the tail latency by 15.5% over Baseline. On top of Baseline+Affinity, partitioning the L2/L3 caches with Baseline+MosaicScheduler reduces the tail latency by 12.8%. Mosaic is much more effective: it reduces the tail latency over the Baseline by 64.8%–79.9%, with an average of 74.6%. The latency reductions are higher for functions with short duration (*e.g.*, *EvStr*), or frequent context switches (*e.g.*, *HotelB*).

Every partitioned hardware structure contributes to the reduction in tail latency. For example, it can be shown that not saving the function state of the L2 cache or the branch target buffer in Mosaic increases the tail latency of functions by 46% or 34%, respectively.

2. Average latency. In addition to reducing the tail latency, Mosaic also reduces the average latency. Figure 5.14b shows the average latency with the four evaluated systems. Baseline+MosaicScheduler reduces the average latency over Baseline by 28.7%. As an average function invocation is more likely to execute on a warm micro-architectural state than the invocations at tail, Mosaic's benefits are slightly lower for the average latency than for the tail latency. On average, Mosaic reduces the average latency by 59.6% and 37.4% over Baseline and Baseline+MosaicScheduler, respectively.

3. Throughput. We define the throughput as the maximum load a system can sustain without violating the SLOs of functions. The SLO of a function is defined to be 5× the execution time of the function on an unloaded system. Figure 5.15 shows the throughput for the evaluated functions in kilo requests per second (kRPS) with Baseline and Mosaic. Mosaic substantially improves the throughput across functions. On average, it improves throughput by 225%. The reason is that, with Mosaic, invocations can reuse their state while still multiplexing their execution on a core with many other invocations of the same or different functions.

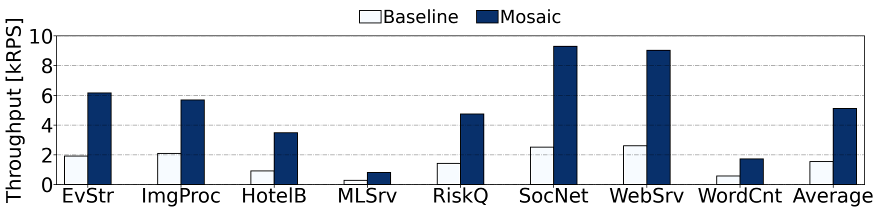


Figure 5.15: Per-function throughput with Baseline and Mosaic.

4. Comparison to prefetching. Figure 5.16 compares the tail latency of Baseline, Jukebox [197], and Mosaic. By reducing the instruction misses, Jukebox reduces the tail latency over Baseline by 12.1% on average. However, Jukebox does not reduce misses in data caches or branch predictors, and introduces additional memory traffic for prefetching. In addition, after a running function invocation is pre-empted and then scheduled to run again, Jukebox does not reprefetch the instructons. Compared to Jukebox, MosaicCPU reduces the tail latency by 71.1% while using less on-chip area.

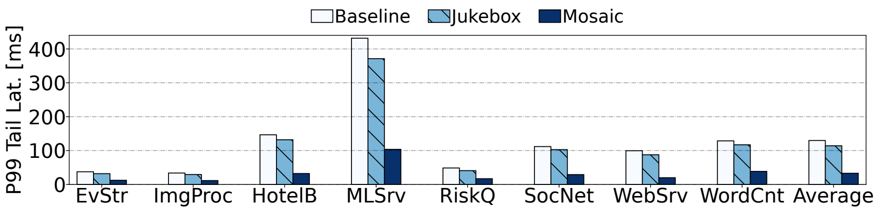


Figure 5.16: Tail latency of Baseline, Jukebox, and Mosaic.

5. Comparison to a manycore. Figure 5.17 shows the throughput of Manycore and Mosaic for monolithic applications and serverless functions normalized to Baseline. Due to space limitations, the figure does not show all the serverless functions, but the average bars include them all. We see that both Manycore and Mosaic substantially improve the throughput of all functions. On average, Mosaic and Manycore increase the functions' throughput by 225% and 271%, respectively. However, the small cores of Manycore prevent it from running the monolithic applications efficiently. On average, Manycore reduces the throughput of monolithic applications by 68% over the Baseline. On the other hand, Mosaic delivers the same throughput as Baseline for these applications. Thus, providers can efficiently run both serverless and non-serverless workloads on Mosaic servers, improving cost-efficiency.

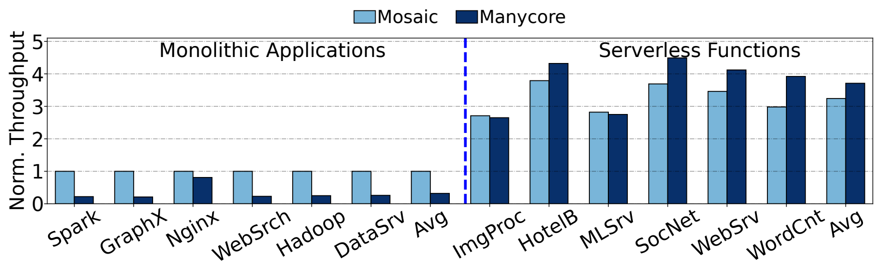
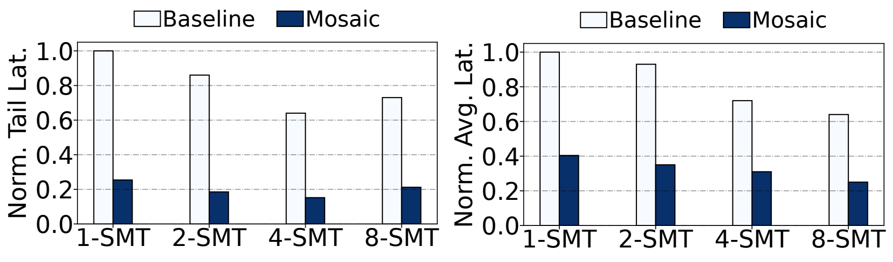


Figure 5.17: Throughput of Mosaic and Manycore for monolithic applications and serverless functions normalized to Baseline.

6. Comparison to SMT. SMT cores can improve the throughput of serverless workloads. However, they (1) compromise security in public clouds due to side channel attacks, and (2) increase tail latency due to resource contention. Figure 5.18 shows the tail and average latency of SMT cores in Baseline and Mosaic averaged across all functions. The results are normalized to single-threaded Baseline. We see that, for both Baseline and Mosaic, 2- and 4-SMTs reduce the latencies due to shorter wait times. As we add more threads (8-way SMT), they compete for the shared resources and increase the tail latency. In all configurations, Mosaic significantly reduces the latency over Baseline.



(a) Tail latency. (b) Average latency.

Figure 5.18: Latency of Baseline and Mosaic with different numbers of SMT threads normalized to single-threaded Baseline.

7. Comparison to MXFaaS. As discussed in Chapter 7, MXFaaS [35] is a software serverless platform that schedules concurrent invocations of the same function on a set of cores "owned" by the function. In addition, MXFaaS allows concurrent invocations of the same function to share the same container—which reduces the memory footprint and the cold-start latency. By binding functions to cores, however, MXFaaS may introduce load imbalance. Mosaic is orthogonal to MXFaaS and can be combined with it to further boost its performance. Figure 5.19 shows the tail latency of Baseline, MXFaaS, Mosaic, and MXFaaS+Mosaic. Compared to Baseline, MXFaaS and MXFaaS+Mosaic reduce the tail latency by 53.1% and 79.3%, respectively.

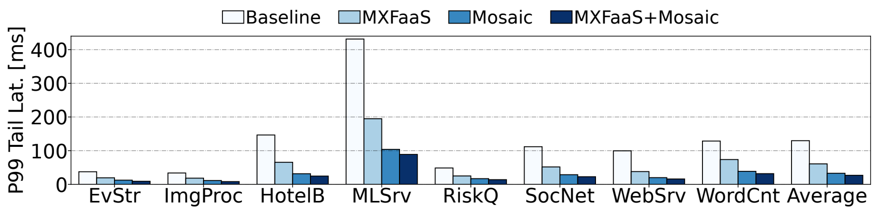


Figure 5.19: Tail latency of Baseline, MXFaaS, Mosaic, and MXFaaS+Mosaic.

5.5.2 Average Power Consumption Reduction

Mosaic keeps the inactive chunks of the partitioned structures at a low voltage level, reducing power consumption. Figure 5.20 shows the average power consumed by each function in Mosaic relative to that in Baseline. We see that, across functions, Mosaic reduces the average power consumption over Baseline by 22%. Power reductions are higher for functions that require smaller tiles in the partitioned structures, such as *RiskQ*. Overall, Mosaic reduces both average response time and power consumption over the Baseline, resulting in an average 80% decrease in the *energy-delay product* of functions. For monolithic applications, Mosaic changes neither performance nor power consumption over Baseline.

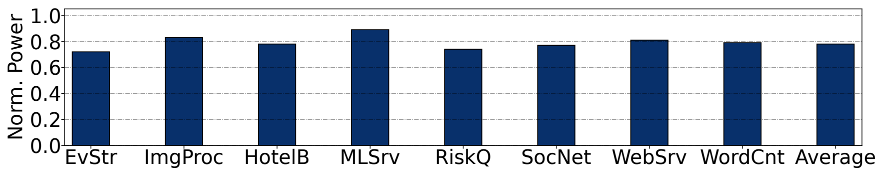


Figure 5.20: Power usage of Mosaic normalized to Baseline.

5.5.3 Co-locating Serverless and Monolithic Workloads

We evaluate Mosaic when co-locating serverless functions with traditional monolithic applications [172] and allowing harvesting of cores assigned to monolithic applications. A monolithic application owns 8 cores, while serverless functions execute on the other 8 cores of the server and can steal more cores when they are idle. We run experiments with each of the 6 monolithic applications of Figure 5.17, and take the average across runs. Figure 5.21 shows the average and tail latency of the serverless functions when running with Mosaic normalized to when they run with Baseline. We see that Mosaic has lower latencies than Baseline even when functions are co-located with monolithic applications. In such an environment, Mosaic reduces the average and tail latencies by 49.8% and 67.8%, respectively.

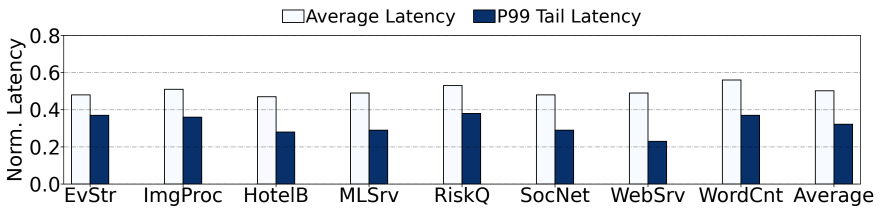


Figure 5.21: Average and tail latency of functions when co-located with monolithic applications normalized to Baseline.

5.5.4 Cost Savings of Mosaic at Scale

To evaluate the cost savings of Mosaic at large scale, we model a datacenter with over a thousand servers running both serverless functions and regular (*i.e.*, non serverless) VMs. We use large open-source production traces [226] for the arrivals, departures, and resource requirements of functions [63] and VMs [152]. Based on the peak utilization, we provision servers using bin-packing. We evaluate three different datacenter designs based on the types of servers they have. *Baseline* uses only traditional servers for both workloads. *Baseline+Manycore* uses a traditional server pool for regular VMs and a separate Manycore pool for serverless workloads. *Mosaic* uses only Mosaic servers for both workloads.

Figure 5.22 shows the number of servers needed in the Baseline, Baseline+Manycore, and Mosaic datacenters while varying the fraction of total CPU hours used by serverless workloads. The number of servers is normalized to the number of servers in Baseline with a 0% fraction of CPU hours for serverless workloads. Note that, in the Baseline+Manycore design, the mix of traditional servers and Manycores is different at different X-axis points—*i.e.*, at different fractions of CPU hours for serverless workloads.

As the fraction of CPU hours devoted to serverless workloads increases, Baseline needs to provision more servers than the other two designs due to its lower throughput for serverless workloads. Baseline+Manycore provisions each server pool independently for each peak, which leaves some of the servers underutilized for certain periods due to fragmentation. Mosaic needs the lowest number of servers because its servers are optimized to execute both types of workloads. Overall, Mosaic reduces the number of servers needed by 10-24% over Baseline+Manycore for a range of fraction of CPU hours for serverless workloads. It never needs more servers than the other designs. Thus, Mosaic has the lowest cost.

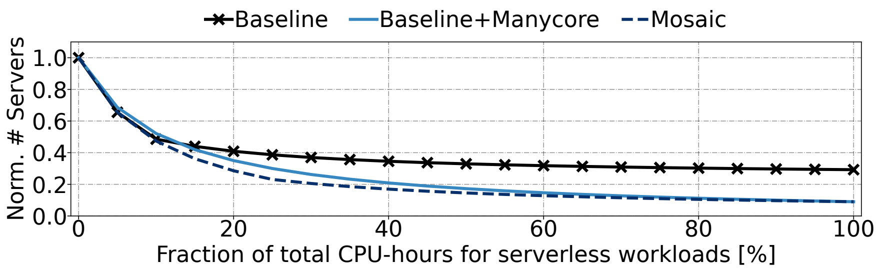


Figure 5.22: Normalized number of servers in Baseline, Baseline+Manycore, and Mosaic datacenters while varying the fraction of total CPU-hours used by serverless workloads.

5.5.5 Sensitivity Studies

We conduct sensitivity studies to analyze the efficiency of Mosaic under various conditions.

1. Sensitivity to system load. We maintain the mix of functions in our workload and invoke the functions with Low, Medium, and High loads using a Poisson distribution. These loads correspond to attaining 25%, 50%, and 70% average CPU utilization as in prior work [35]. Each function is invoked with equal probability. Figure 5.23 shows the tail latency for each function when running on Mosaic with the three loads normalized to the tail latency when running on Baseline with the same load.

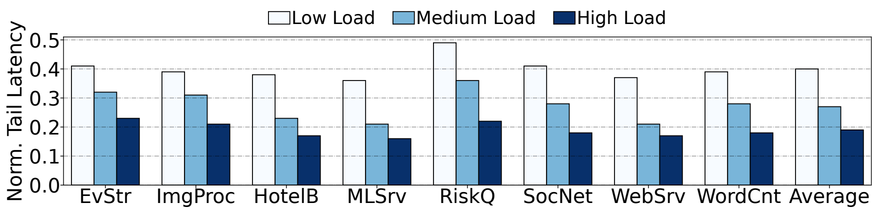


Figure 5.23: Normalized tail latency of Mosaic over the Baseline.

We see that, at all loads, Mosaic has a substantially lower tail latency than Baseline. At high loads, cores context switch more frequently and, therefore, as the load increases, the benefits of Mosaic over the Baseline are even higher. On average, Mosaic reduces the P99 tail latency over Baseline by 59.8%, 72.7%, and 80.9% in Low, Medium and High load, respectively. We observe similar trends for average latency.

2. Sensitivity to core count. Serverless providers may want to reduce their operating cost by reducing the number of cores. We perform a sensitivity analysis to measure how the tail latency changes as the core count decreases. Figure 5.24 shows the tail latency of Mosaic with various core counts normalized to the tail latency of the 16-core Baseline. On average, Mosaic with 16, 12, 8, and 4 cores reduces the tail latency of the 16-core Baseline by 75%, 66%, 39%, and 4%, respectively. Thus, the provider can maintain the current response time of Baseline at 4× lower cost with Mosaic.

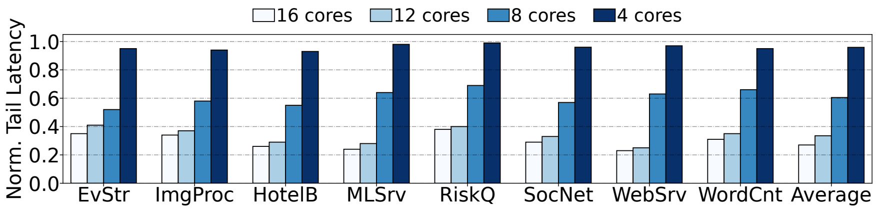
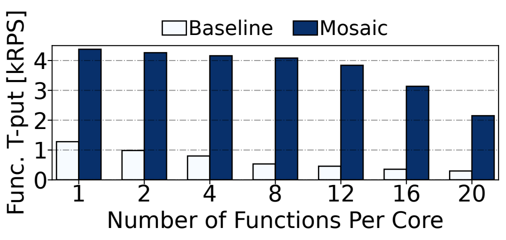
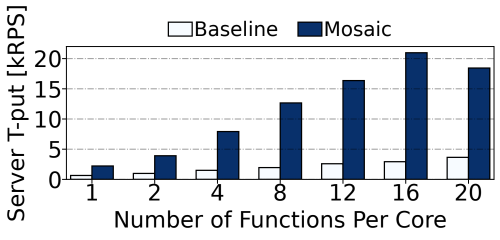


Figure 5.24: Tail latency of Mosaic with different numbers of cores normalized to Baseline with 16 cores.

3. Sensitivity to core oversubscription. In this experiment, we vary the number of *different* functions that are scheduled in a round-robin manner to execute on a given core. We measure the highest sustainable load without SLO violations—*i.e.*, the throughput. With a higher number of different functions, the core oversubscription increases, and the state in the core structures in both Baseline and Mosaic gets polluted more.



(a) Per-function throughput.



(b) Total server throughput.

Figure 5.25: Throughput while varying core oversubscription.

Figure 5.25 shows the changes in throughput in *Baseline* and *Mosaic* as we increase the number of different functions scheduled on a core. Figure 5.25a shows the per-function throughput and Figure 5.25b the total server throughput. We see that, as the oversubscription increases, per-function throughput drops for both Mosaic and Baseline due to more state pollution. On the other hand, total server throughput increases as more functions are running in a server. In all cases, Mosaic delivers much higher throughput than Baseline. Even with 20 functions per core, Mosaic delivers a 68% higher per-function throughput than Baseline with a single function per core. Overall, Mosaic delivers a much higher total server throughput than Baseline.

4. Sensitivity to number of different functions. In all of our experiments before Section 5.5.5, we run 8 different functions. Recall that we set-up the environment so that different instances of the same function do not share any instructions or data (Section 5.4), ensuring that there is no state reuse across them. Running the experiments with a higher number of different functions should not change the results if the rate of function invocation is the same. To prove it, we perform a new experiment with 64 different functions [171, 202, 227, 228] (as many as there are functions in the production traces) with the same total invocation rate. We observe that the results change very little. It can be shown that, with 64 different functions, Mosaic reduces the average and tail latency by 62.1% and 76.5%, respectively, over Baseline. With 8 different functions, Figure 5.14 showed that Mosaic reduces the average and tail latency by 59.6% and 74.6%, respectively. Hence, our methodology mimics environments with many functions, as in real settings.

5.5.6 MosaicScheduler Prediction Accuracy

We compare the tile sizes created by MosaicScheduler for the evaluated functions and the optimal tile sizes obtained via exhaustive search. Figure 5.26 shows the tile sizes created by MosaicScheduler in number of chunks for each function and hardware structure. Most of the functions use 1-2 chunks. It can be shown that these tile sizes closely follow the optimal ones. The only discrepancy occurs in HotelB and SocNet for the BTB, where MosaicScheduler creates a tile larger by one chunk. MosaicScheduler over-predicted the tile sizes in these two cases and never under-predicted them, ensuring no performance degradation.

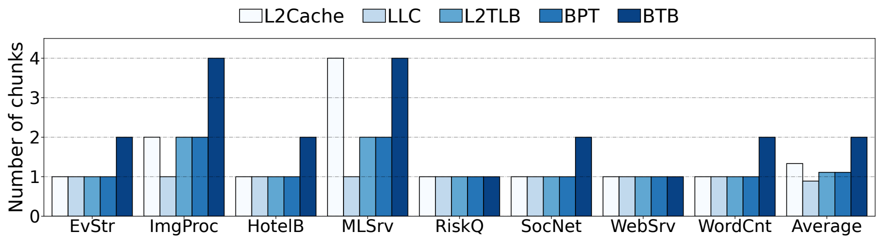


Figure 5.26: Tile sizes created by MosaicScheduler.

Also, we measure the accuracy of our regression model that predicts the size of non-profiled functions. We collect metrics for 70 open-source functions [171, 201, 227, 228, 229], find their optimal configurations via exhaustive search, and create the dataset. We split the dataset into 80% train and 20% test. The resulting model produces accurate predictions. For most of the structures, such as the branch predictor table, the model predicts the optimal tile sizes with 100% accuracy. For some structures, such as the branch target buffer, the model can slightly overpredict the tile size. On average, the model achieves 92% accuracy. The compute requirements of the model are very low: a prediction takes a few hundred μs and is done off the critical path. The model is queried only when a function is admitted to the cluster for the first time.

5.6 RELATED WORK

1. Partitioning schemes. Many researchers have explored resource partitioning of architectural resources so that applications meet quality of service (QoS) (*e.g.*, [230, 231, 232, 233, 234, 235, 236, 237]). In some cases, they consider concurrently-running SMT threads (*e.g.*, [238, 239, 240, 241]). PARTIES [230] tracks the tail latency of services and moves the resources (LLC, memory bandwidth, I/O, or cores) between services based on their deadlines. In these works, resources are managed with a feedback controller [230], Bayesian networks [233], multi-armed bandits [231] or ML techniques [234, 235, 237]. These schemes are efficient for long-running datacenter services and a relatively fixed mix of co-located services. Using them in serverless environments would not prevent a function from losing its state in a core on a context switch. With SMT proposals, one would sacrifice the security by allowing different functions to run concurrently on the same core. Mosaic targets more dynamic serverless environments with the goal of preserving the functions' micro-architectural state across context switches while keeping the security guarantees.

2. Micro-architecture prewarm. A few studies have explored the impact of serverless environments on the underlying hardware. Shahrad *et al.* [196] observed that cold-starts, containerization, and inter-function interference reduce the effectiveness of micro-architectural structures. Jukebox [197] uses on-chip metadata to prefetch instructions for functions that start with a cold micro-architectural state. In non-serverless environments, Ahn *et al.* [242] preserve a VM's context in the LLC on a cross-VM context switch. Mosaic preserves a function's micro-architectural state across context switches without increasing the on-chip area or memory bandwidth consumption.

3. Serverless optimizations. Optimizing serverless software stacks has received substantial attention (*e.g.*, [35, 37, 38, 53, 63, 64, 191, 192]). MXFaaS [35] improves performance by efficiently sharing a server's resources between concurrently executing same-function invocations. REAP [53] records a function's stable working set of guest memory pages and prefetches it from disk. SpecFaaS [37] accelerates the execution of multi-function serverless applications with software-supported speculative execution of functions. EcoFaaS [38] redesigns the serverless software stack to improve energy efficiency while maintaining high performance. Mosaic can further enhance the effectiveness of such systems. Researchers have also proposed using simple cores to host serverless workloads [210]. More advanced designs such as μManycore [31] potentially move the tipping point closer toward processor specialization for serverless environments.

5.7 CONCLUSION

This chapter presented a micro-architectural characterization of serverless environments and proposed Mosaic, an architecture optimized for serverless environments. Mosaic has two components: (1) *MosaicCPU*, a processor architecture that efficiently runs both serverless and traditional workloads, and (2) *MosaicScheduler*, a software stack for serverless systems that maximizes the benefits of MosaicCPU. MosaicCPU partitions micro-architectural structures into small chunks and assigns tiles of such chunks to functions. MosaicScheduler sizes the tiles for functions and schedules function invocations based on the state of the tiles. Compared to conventional server-class processors, Mosaic improves the throughput of serverless workloads by 225% while using 22% less power. Conversely, Mosaic achieves the performance of server-class processors with one quarter of the cores.

CHAPTER 6: Domain-Specific Accelerators for Cloud-Native Services

6.1 INTRODUCTION

Despite their numerous benefits, microservice environments suffer from various software overheads [14, 50, 213, 243]. For instance, microservices are commonly implemented as Remote Procedure Call (RPC) servers [41, 79], which facilitates their distribution across machines and independent scalability. However, RPC processing has overhead. Further, data communicated between services must undergo (de)serialization to ensure compatibility across different programming languages through standardized protocols [244]. Additionally, microservice environments use encryption for security and compression to reduce resource use. Collectively, these auxiliary or "glue" operations are known as *datacenter tax* [243], and can consume a substantial fraction of CPU cycles in datacenters [15, 243, 245].

To minimize datacenter tax, researchers have proposed numerous hardware accelerators or software techniques that target a single source of datacenter tax [44, 45, 46, 114, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258]—e.g., compression, Transmission Control Protocol (TCP), or RPC processing. Given the many sources of datacenter tax [245], these proposals must be augmented with a way to efficiently orchestrate the multiple accelerators of an ensemble of accelerators integrated in a server. Efficiency is key because, in microservice environments, the operations to be accelerated may take only tens of microseconds.

There are proposals to orchestrate on-package accelerators in other domains, such as monolithic applications for image/video processing or ML [259, 260, 261, 262, 263]. In some proposals, a CPU core [259] or a centralized hardware manager [260, 261] orchestrates the accelerators by invoking them and receiving an interrupt when each accelerator completes. This approach is suitable for coarse-grained accelerator operations. However, in the context of microservices with fine-grained accelerator operations, a centralized orchestrator is suboptimal. Other designs allow direct accelerator-to-accelerator coordination, either in hardware [262] or in software [263], without involving a centralized agent. However, they statically link pairs of accelerators, requiring fixed accelerator sequences [262, 263].

Our characterization shows that such an approach is insufficient for microservices. Microservices can leverage sequences of accelerators that vary both across services and across invocations of the same service. Moreover, the sequence of accelerators to use is often affected by "branch conditions," whose real-time resolution determines the set of subsequent accelerators needed. Additionally, the data transferred between accelerators varies in both size and format. This demands a *dynamic and flexible* orchestration framework.

To address these challenges, we propose *AccelFlow*, the first accelerator orchestration framework for microservices. AccelFlow orchestrates an on-package ensemble of fine-grained accelerators. CPU cores build software structures called *Traces* that contain a sequence of accelerator IDs, and store them in an Accelerator Trace Memory (ATM). A trace can include branch conditions whose outcomes determine the control flow inside the trace, or the address in the ATM of the next trace to execute after the current one completes. A CPU core triggers accelerator execution by enqueuing a trace in an accelerator in user mode. From then on, the accelerators in the trace execute in sequence without CPU involvement, passing data from one accelerator to the next, potentially executing branch conditions and accessing the ATM for additional traces. Once the traces have executed, control returns to the CPU.

Each accelerator has a standard interface with input and output hardware queues and input and output controllers (called dispatchers). The input dispatcher schedules requests enqueued in the accelerator. The output dispatcher computes any branch condition in the trace and sends the accelerator's output data to the input queue of the next accelerator in the trace using a DMA engine.

We evaluate AccelFlow with full-system simulations of a server with an Intel IceLake-like processor and nine state-of-the-art accelerators [44, 246, 248, 249, 253, 257]. We run large open-source microservice applications [213] with real-world invocation traces [5]. Our results show that AccelFlow is very effective. Compared to state-of-the-art proposals for accelerator orchestration [260, 263], on average across services, AccelFlow reduces P99 tail latency by 70%, reduces average latency by 38%, and increases throughput by 120%.

This chapter makes the following contributions:

• A characterization of how microservices can use ensembles of on-package accelerators.

• AccelFlow, the first orchestration framework for an ensemble of fine-grained accelerators in microservice environments.

• An evaluation of AccelFlow, compared to the state-of the-art.

6.2 BACKGROUND

Workflow of a Microservice Invocation. A microservice invocation arrives to the server as an encrypted network message. To achieve reliable and ordered delivery of messages, the TCP stack [264] first performs message reassembly, congestion control, and checksum calculation. The TCP forwards the message to the SSL protocol [265] to authenticate the client and decrypt the message using algorithms such as RSA, AES, or SHA.

The decrypted message, containing the name of the function to invoke and its arguments, is forwarded to the RPC stack [41, 79]. A microservice may have multiple entry points or functions that users can invoke. The server keeps a table that maps the name of the function to its *handler* and *descriptor*. The role of the RPC stack is to decode the name of the function from the message and fetch the function handler and descriptor from the table. Then, the RPC stack forwards the function handler, descriptor, and serialized arguments to a deserialization protocol, e.g., Protobuf [244]. Protobuf uses the function descriptor to deserialize the arguments: to translate the arguments' wire format to their application format in a given language. To reduce the network bandwidth use, large arguments can be compressed, e.g., with Zstd [266] or Snappy [267]. Thus, after deserialization, some arguments may be decompressed.

Finally, the invocation is ready to execute. Using a software or hardware algorithm, a load balancer [257] picks a core to execute the function. When execution completes, the steps above are performed in reverse order: compress the results, serialize the results, encode the message in the RPC stack, encrypt the message, and transmit it via TCP. Within an invocation, the function may also invoke other services or access storage, creating nested RPCs with similar steps.

CPUs with Integrated Accelerators. Vendors have long integrated special-purpose hardware to accelerate common functions—e.g., the cryptography accelerator in Sun's UltraSPARC T1 [268], or integrated GPUs in AMD's Llano APUs [269]. Recently, Intel integrated several accelerators into their Sapphire Rapids CPU [270], targeting datacenter functions. This CPU introduces enhancements to the system architecture for the accelerators. The accelerators are not just PCIe devices programmed via memory-mapped I/O operations. Instead, their ISA has instructions for dispatching work and for signaling; the accelerators operate with virtual addresses exploiting the IOMMU for address translation; and the accelerators support virtualization to make them usable in a cloud environment.

6.3 CHARACTERIZING ACCELERATORS FOR MICROSERVICES

To understand the opportunities and challenges of using an ensemble of accelerators in microservice environments, we measure the performance of a 36-core Intel Xeon Platinum server [271] at 2.4GHz, and estimate the potential impact that nine hardware accelerators proposed by prior work could have. These accelerators speed-up TCP [246], (De)Encryption (Decr and Encr) [247], RPC [44], (De)Serialization (Dser and Ser) [249], (De)Compression (Dcmp and Cmp) [253], and load balancing (LdB) [257]. We run over 80 open-source services from DeathStarBench [213], Train Ticket [78], and μSuite [50]. The infrastructure is presented in Section 6.6.1.

***Q1:* Is a Processor with Many Accelerators a Good Environment for Microservices?** Figure 6.1 breaks down the execution time of invocations of the SocialNetwork services from DeathStarBench on the Intel Xeon server. We identify the sections of the code that can be assigned to one of the accelerators considered. The remaining time is called *AppLogic*. A given accelerator category corresponds to multiple code sections. Bars are normalized and the numbers on top of them are the absolute execution times.

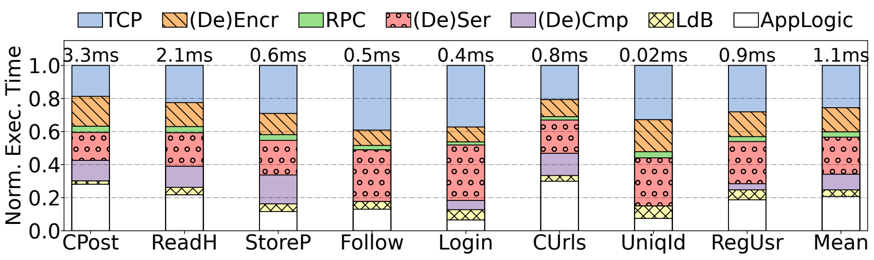


Figure 6.1: Execution time breakdown of SocialNetwork service invocations on the Intel Xeon server. The numbers on top of the bars are the absolute execution times of the invocations.

Despite the high diversity of the microservices' core logic, all services experience all the datacenter tax sources. In fact, microservice invocations spend most of their execution time on tax. On average, an invocation spends only 20.7% of its end-to-end execution time on the core application logic. It spends 25.6%, 14.6%, 3.2%, 22.4%, 9.5% and 3.9% of its time on the TCP, (De)Encr, RPC, (De)Ser, (De)Cmp and LdB operations, respectively. The relative weight of tax increases for microservices with (1) short execution times (e.g., UniqId) or (2) many RPC or storage accesses (e.g., Login). Services from other suites show similar properties. Hyperscalers have also seen that tax dominates the execution of their services [15, 243, 245].

Moreover, in many cases, multiple tax operations are executed back to back, in a sequence, without interleaved operations of the core logic. Figure 6.2 shows sequences of tax operations executed when a core sends a function response (Figure 6.2a) or a read request to a database cache (Figure 6.2b). The first sequence involves Ser, RPC, Encr, and TCP; the second one has Ser, Encr, and TCP.

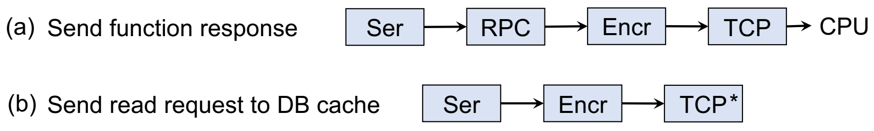


Figure 6.2: Examples of sequences of datacenter tax operations.

If we have accelerators for each of these operations, the question arises as to how to orchestrate them. We consider three approaches. First, in *CPU-Centric* [259], a CPU core invokes one accelerator at a time. When an accelerator completes, it interrupts the core, which can then invoke the next accelerator. While an accelerator is executing, the core can execute another request to avoid idle time. Second, in *HW-Manager* [260, 261, 272], where we model RELIEF [260], the CPU offloads the scheduling of the accelerator requests to a centralized hardware manager. The CPU submits to the manager the sequence of accelerators to invoke. When an accelerator completes its job, it interrupts the manager, which then calls the next accelerator in the chain. On completion of a chain, the manager interrupts the CPU. Finally, in *Direct*, scheduling is performed neither by a CPU core nor by a manager. Instead, a CPU core passes a list of accelerators that need to execute in sequence to the first accelerator. After an individual accelerator in the sequence executes, it directly calls the next accelerator in the sequence without involving the CPU.

We simulate the execution of these three environments, modeling a server like the one measured, and modeling the nine accelerators [44, 246, 247, 249, 253, 257]. We describe the simulation infrastructure in Section 6.6.1. We record the time taken by the orchestration overhead—i.e., sending and receiving interrupts and communicating between CPU core, hardware manager, and accelerators. Figure 6.3 shows the average orchestration overhead as a fraction of the total execution time of the service. The figure shows data for the three approaches above, as the load of the 36-core processor changes. We see that *Direct* has less overhead than *HW-Manager*, which has less overhead than *CPU-Centric*. The overhead of the last two approaches increases rapidly with the load. For common loads around 15 kRPS, the overhead is 25% and 15% in *CPU-Centric* and *HW-Manager*, respectively. This is a major inefficiency.

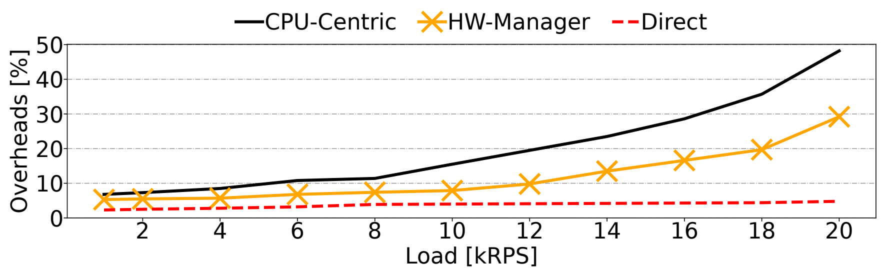


Figure 6.3: Orchestration overheads of different approaches averaged across all services with varying service load.

***Q2:* What is the Control Flow in Accelerator Sequences?** It is important to know if the sequences of accelerator operations triggered by a request are fixed (i.e., an operation in Accel.A is always followed by an operation in Accel.B) and need no intervening CPU involvement. To answer this, we analyze the sequences in our 80 services. For each accelerator in a sequence that requires no intervening CPU involvement before or after, we record which accelerator provides its input (the source) and which accelerator consumes its output (the destination). As shown in Table 6.1, an accelerator can consume and produce data from/for multiple accelerators. Thus, the inter-accelerator connections need to be flexible.

Table 6.1: Source/destination accelerators for each accelerator.

Accelerator	**Src Accelerators**	**Dst Accelerators**
TCP	Ser, Encr, Cmp	LdB, Decr, Dser, Dcmp
Encr	TCP, RPC, Ser	TCP, RPC
Decr	TCP	RPC, Dser
RPC	Decr, Ser	Encr, Deser, LdB
Ser	Deser, Cmp, CPU	TCP, Encr, RPC
Dser	TCP, Decr, RPC	Ser, Dcmp, LdB
Cmp	Deser, CPU	Ser, LdB, CPU, TCP
Dcmp	Deser, TCP, CPU	(De)Ser, LdB, CPU, TCP
LdB	TCP, Dser, Dcmp	CPU

We now consider the control flow inside accelerator sequences that contain no intervening CPU core. We find that a sequence of accelerators often includes dynamic control flow. Figure 6.4 shows two examples. Figure 6.4a is the sequence when a processor receives a function request. The payload may be compressed and require decompression. However, this is unknown until the deserialization step. At that point, we must check a field in the message to decide whether to invoke the Dcmp accelerator or send the data to LdB.

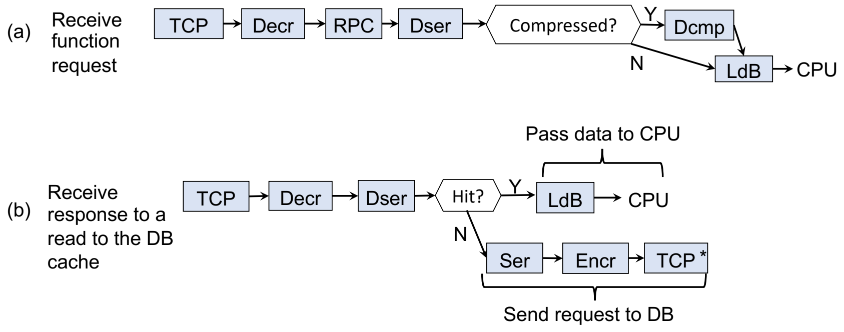


Figure 6.4: Dynamic control flow in sequences of tax operation.

Figure 6.4b shows the sequence when a processor receives the response of a read request to the database cache. Depending on whether the request hit in the cache, different actions are needed. If it hit, the response contains the data and is forwarded to a CPU core; otherwise, the response does not contain the data, and a new request is issued to the actual database. Whether a hit occurred is only known after the Dser accelerator has completed.

Our data shows that 69.2%, 62.5%, 82.5%, and 53.8% of the sequences of accelerators in the *SocialNet*, *HotelReservation*, and *MediaServices* microservices from DeathStarBench, and

TrainTicket, respectively, have at least one conditional statement. Some sequences have up to four conditional statements. Given this, interrupting the CPU every time that a sequence of accelerators encounters a conditional statement would induce substantial overhead. These conditionals are simple, as they involve checking a few bits or flags embedded in the data payload. For example, determining whether to invoke the decompression accelerator merely requires checking a field in the message after deserialization. These decisions are binary and involve simple comparisons and computation logic.

***Q3:* How Should Data be Forwarded Between Accelerators?** The data forwarded from one accelerator to the next exhibits two types of variation: data size and data format. The data size varies across accelerators and, for a given accelerator, across services and across requests in the same service. Figure 6.5 shows the size of the input and output data for each accelerator across all 80 services. In each case, we show the minimum, median, and maximum size in a stacked matter. The median data size is small, i.e., a few KBs (similar to Google [225]). There is, however, a long tail of large data transfers with a few tens of KB. As an example, in our services, the median size of output messages in the Encr accelerator is 1.9KB; however, some messages are larger than 50KB. Hence, while the paths and queues between accelerators can be optimized for small data sizes, they should also support infrequent large messages.

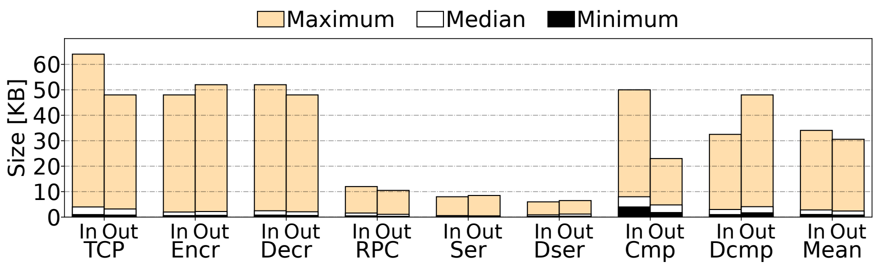


Figure 6.5: Sizes of the input/output data of each accelerator.

In some cases, the data transferred between two accelerators is generated in one format by the source and needs to be consumed in a different format by the destination. As an example, shown in Figure 6.4b, when the Dser accelerator receives a response from a database cache, it outputs the data in string format. This format is used by the CPU (via the LdB accelerator). However, if we missed in the cache, we need to send the request to the database. Hence, this request needs to be serialized again (Figure 6.4b), but the input to the Ser accelerator needs to be in BSON format [273]. This transformation from string to BSON and similar ones are relatively simple. They can be handled by the accelerators with simple hardware logic [250] that performs basic operations like parsing and reformatting. They do not need to involve a CPU core.

There are cases when an accelerator can consume data in different formats. In this case, no special action is needed. For example, the Dser accelerator receives data in Protobuf [244] format when executing after the RPC accelerator (Figure 6.4a), and in string format when executing after the Decr accelerator on data coming from a database cache (Figure 6.4b). Dser can interpret each format and transform the byte sequences into its internal *schema*.

Summary. Microservices can benefit from ensembles of accelerators that target different sources of datacenter tax. These accelerators can be invoked in *sequences without intervening CPU*. For highest performance, the orchestration of such ensembles should avoid invoking a CPU or a centralized hardware manager. Instead, within a sequence, accelerators should handle dynamic control flow, data transformations, and data transfers of different sizes.

6.4 ACCELFLOW: ORCHESTRATING ACCELERATORS

Based on our characterization, we propose *AccelFlow*, an architecture that orchestrates an ensemble of accelerators for microservices in a decentralized manner for high performance.

6.4.1 AccelFlow Hardware Organization

We envision *AccelFlow* to be implemented in a large, multi-chiplet processor with many cores and one or more instances of all the accelerators of Section 6.3. The processor can be organized in different ways: cores and accelerators may share the same chiplet like in the Intel Sapphire Rapids (SPR) CPU [199], or accelerators may be in their own chiplet—e.g., the I/O die of some AMD and Intel servers. The latter design is the one we assume and is shown in Figure 6.6: one type of chiplet contains an instance of all the accelerators except for LdB, and another type of chiplet contains cores, private caches, a distributed shared LLC and LdB. The LdB accelerator is in the core chiplet since it is tightly coupled with the cores. Irrespective of how the accelerators are placed, they execute in sequences, communicating with each other without CPU involvement.

Our design assumes that all the accelerators have the same standard interface. This is a futuristic goal that is motivated for simplicity and by recent industry trends, such as the Intel Accelerator Interfacing Architecture (AiA) [274]. Moreover, accelerators and cores share the same virtual address space (like Intel's Shared Virtual Memory (SVM) [275] in SPR). Like in SPR, the accelerators directly read from and write to the processor's LLC in a cache coherent manner—i.e., on a read, they get the correct data in the system, while on a write, they invalidate the copies in all private caches.

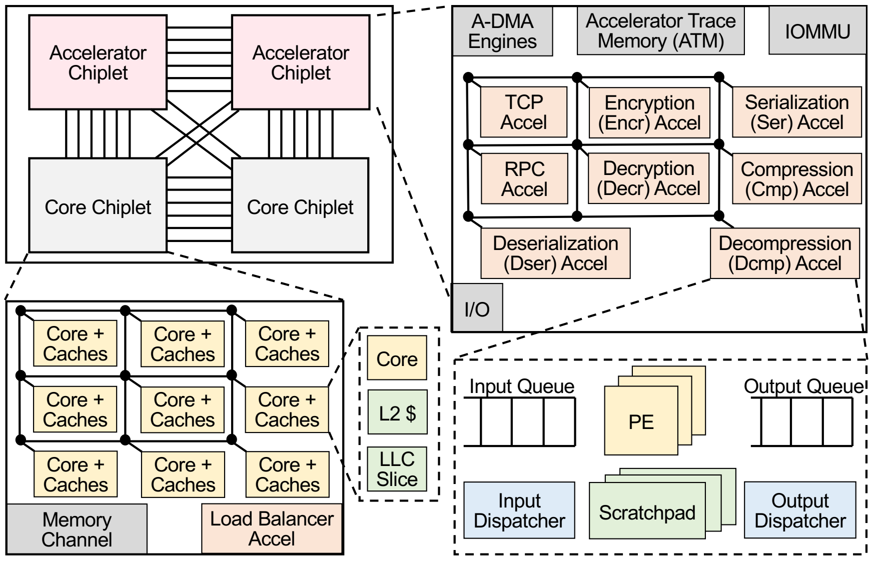


Figure 6.6: Possible organization of a processor with AccelFlow.

Sequence of Accelerators or *Trace*. In AccelFlow, CPU cores build software structures called *Traces* that contain a sequence of accelerator IDs, and store them in a special on-chip memory called *Accelerator Trace Memory* (ATM) (Figure 6.6). A trace can include branch conditions, whose outcomes determine the control flow inside the trace. Branches are encoded as simple logic operations on a set of fields or flags within the payload of the message—e.g., ”if (field1 & field3) goto Ser; else goto Cmp”. A trace may also contain data transformation fields and, in its tail, the address in the ATM that contains the next trace to execute after this one completes.

A CPU core can trigger accelerator execution by passing a trace to an accelerator. From then on, the accelerators in the trace will execute in sequence without CPU involvement, passing data from one accelerator to the next, potentially executing branch conditions, and potentially accessing the ATM for additional traces. Once the trace(s) have executed, control returns to the initiating CPU core.

Architecture of an Accelerator. An accelerator has an SRAM input queue and an SRAM output queue, an input and an output controller (called *Dispatcher*), and multiple processing elements (PEs), each one with a scratchpad, that perform the same acceleration operations (Figure 6.6). A scratchpad contains accelerator PE state that may be private to a request (e.g., the History Window [253] for the Cmp and Dcmp), or shared by all service's requests (e.g., the Accelerator Descriptor Table (ADT) [249] in the Ser accelerator).

Each queue entry contains multiple fields: (1) space for a trace with a moving *Position Mark* that indicates which accelerator is to execute next, (2) the ID of the tenant that is using the entry (since an accelerator can be used by multiple tenants), and (3) up to 2KB of data to be operated upon, to capture the common-case data sizes. The entry may also contain a virtual address *Memory Pointer* that points to a software buffer in memory where more data is stored. Large inputs or outputs (more than 2KB) are stored in such locations. Such a buffer is cacheable in the LLC and cores' private caches, and is kept cache coherent. The input and output hardware queues are not visible to the cache coherence protocol.

The input dispatcher manages the input queue, deciding which entry should be executed by which PE next. If an input queue entry needs data from multiple source accelerators, the dispatcher marks the entry as ready only when all the data has arrived.

The output dispatcher performs multiple operations. First, if the next field in the trace is a branch condition, it determines the next accelerator to invoke. Second, if the next trace field includes a data transformation, it transforms the output data. Third, it uses one of a set of on-chip DMA engines (*A-DMA* in Figure 6.6) to send the output data to the input queue of the next accelerator in the sequence. Finally, after the last accelerator in the trace has executed, the output dispatcher examines the tail of the trace. If it finds the address of an ATM location, it accesses the address and gets the next trace to run. Otherwise, it notifies a CPU core.

Operation of an Accelerator. Each PE in an accelerator consumes input queue entries as mandated by the input dispatcher. The PE consumes the data from the input queue entry plus any additional data in the CPU memory hierarchy that is accessible through the Memory Pointer in the input queue entry. A PE is non-preemptible: it runs tasks to completion. After the PE executes the accelerated operation, it deposits the data results and all the metadata (including the trace) in an entry of the output queue. The output dispatcher handles the entry from then on.

Accelerators exploit PCIe's Address Translation Service (ATS) [276], which enables a device to submit address translation requests to the IOMMU (Figure 6.6). An ATS request converts a process ID and virtual address from that process to a physical address. Each accelerator caches the results in an address translation cache, i.e., a TLB, that is accessed by the input and output dispatchers.

On any exception, the accelerator operation stops, a CPU core is interrupted, and the OS handles the exception.

Anatomy of the Execution of Traces. When the software wants the accelerator ensemble to execute a sequence of operations, it creates a trace. Then, a CPU core executes an *Enqueue* instruction in *user mode*. Enqueue takes as arguments an accelerator ID (the first accelerator in the trace) and the trace. The instruction triggers the input dispatcher of the corresponding accelerator to allocate an entry in the input queue and store the trace in the entry. Enqueue returns the entry's index in the input queue. Then, the core invokes an A-DMA engine, passing the accelerator ID, the input queue index, and a pointer to the data needed to perform the accelerator operation. The A-DMA engine uses the pointer to coherently collect the data and deposit it in the corresponding input queue entry. If the Enqueue instruction returned an error (e.g., because the input queue did not have space for another request), the CPU core retries the call to another accelerator of the same type.

Execution proceeds from accelerator to accelerator, transforming the data, and moving the data and trace from the output queue of one to the input queue of another. When the last accelerator of the trace completes its execution, its output dispatcher uses an A-DMA engine to move the resulting data to a memory location. The dispatcher then sends a *user-level* notification to the CPU core that initiated the ensemble execution, passing the virtual address of the memory location that contains the result. To reduce overhead, such notification is not an interrupt. The CPU core can wait for the notification by polling a flag, or by executing an MWAIT-like instruction [277] and automatically wake up on notification.

Sometimes, it is desirable to partition a trace into multiple subtraces. This is done for three reasons: to avoid having to transfer a very long trace, to enable the reuse of individual subtraces across multiple operations, or when, at a point in the trace, a decision needs to be made that can cause a major divergence of what set of accelerators to invoke next. Then, the software creates multiple traces to execute in sequence. Before the CPU core invokes the first trace, it stores all the traces but the first one in the ATM. Moreover, it includes in the tail of each trace but the last one, the address in the ATM of the subsequent trace.

With this design, when an output dispatcher processing a trace finds that the trace concludes with an ATM address, the output dispatcher reads the next trace from the ATM and deposits it in the input queue of the next accelerator. The new trace then executes.

Preventing Starvation and Deadlock. When a core invokes Enqueue, it may need more than one try to find a free input entry in an accelerator. If, after multiple attempts, the core cannot find a free entry, trace execution falls back to the core. Deadlock is also avoided: when an output dispatcher does not find space in the input queue of the next accelerator in the sequence to deposit its data, it times-out and execution falls back to the core.

Handling Limited Input Queue Space. It is possible that when an output dispatcher wants to enqueue an entry in an input queue, the latter is full. Unlike the case for Enqueue, where the CPU would get an error and retry, the output dispatcher cannot keep retrying. Hence, each input queue has an *Overflow* pointer that points to an overflow area in memory. When an output dispatcher needs to enqueue in a full input queue, it stores it in the queue's overflow memory. The overflowed entries are progressively moved into actual queue entries as the latter become free. This design is relatively easy to support as the output dispatcher uses virtual addresses. We size the overflow area based on the expected peak load of the system, such that all requests can be served within the loosest SLOs. If the area is full, execution falls back to the core. There is no deadlock.

6.4.2 AccelFlow Execution Model

To understand AccelFlow's operation, we examine its execution model, covering what triggers trace execution, what occurs on trace completion, and all traces within our services.

Terminating a Trace Execution. When a trace execution terminates, it can either store the results in memory and notify the CPU core that initiated it (Section 6.4.1), or it can instead start the execution of another trace stored in the ATM. The latter happens, for example, when the last operation of the trace involves the TCP accelerator sending a message that requires a response. An example is shown in Figure 6.2b, which is the trace executed to send a read request to a database cache. The trace invokes the Ser, Encr, and TCP accelerators. The asterisk in the TCP box means that the last entry in the trace includes an ATM address. After TCP sends the request, the TCP output dispatcher accesses the ATM address, which contains the trace to execute when the message response will be received. This trace is immediately loaded into the input queue of the same TCP accelerator. Later, when the response is received, it is routed to the same TCP (since the initial request included the TCP ID) and triggers the execution of the stored trace.

These input queue entries in a TCP accelerator are not held indefinitely waiting for a response. After a certain time, a timeout occurs and execution falls back to the core. The core can terminate the request, re-issue the network message, or inform the user.

Triggering the Execution of a Trace. A trace execution may be triggered by a CPU core with an Enqueue (Section 6.4.1) or by the arrival of a message. An example of the latter is shown in Figure 6.4b, which is executed when the response to the read to the database cache is received. As indicated above, the TCP that sent the request already has the new trace ready.

On message arrival, the trace invokes the TCP, Decr, and Dser accelerators. The data generated by Dser contains a field that indicates whether the request hit in the database cache and returned the data. The output dispatcher of Dser checks the field. If it is set, the data is passed to the LdB accelerator and then the CPU core that started the request is notified. The trace always contains the ID of the CPU core that started it. That CPU core contains a record of what thread issued the original request and needs to be notified.

If, instead, the field is clear, the request had missed in the cache and a read message must be sent to the DB. Hence, the Ser, Encr, and TCP accelerators are invoked. Then, as denoted by the asterisk in the figure, the ATM address at the tail of the sequence is used to access the ATM. The trace in that location is loaded in the input queue of the same TCP accelerator for when the response is received.

Noteworthy Cases. Two noteworthy cases occur when a core receives a new request for a function and when the core sends the final response of the function to the client. To handle the former case, all TCP accelerators already store a trace like the one in Figure 6.4a. The trace follows the execution of TCP with that of Decr, RPC, Dser, Dcmp (if the data is compressed), and LdB. LdB saves the results and notifies a CPU core. The data saved includes the ID of the TCP.

When the core finally sends the response of the function, the ensemble executes the trace in Figure 6.2a—after reading it from the ATM or as initiated by a CPU core. The trace triggers the execution of Ser, RPC, Encr, and the same TCP that received the message—since it contains the TCB (transmission control block) of the request in its internal state. After the message is sent, the CPU is notified.

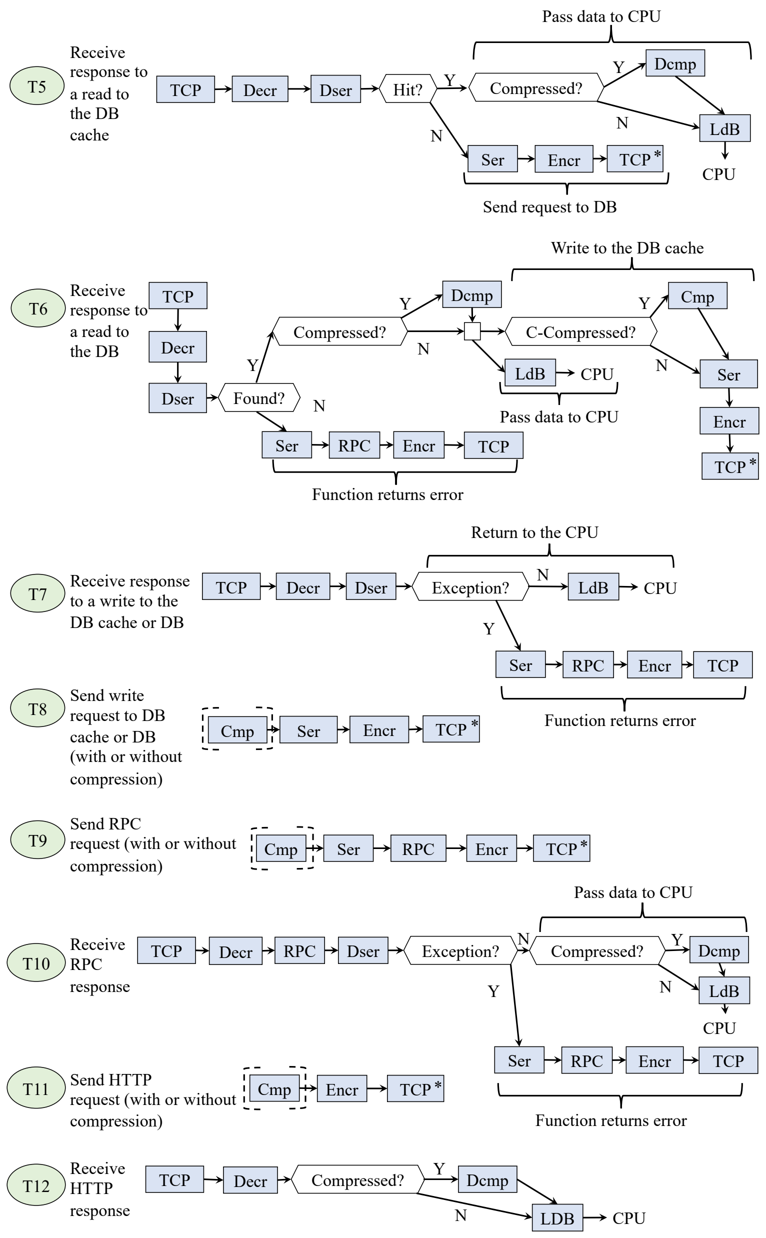
Complete List of Traces in our Services. Table 6.2 shows the complete list of traces that we have identified for our services. Recall there is no intervening CPU core action in a trace. Other services may benefit from additional traces.

Table 6.2: Traces that our services can use. DB means database.

Trace	**Explanation**
T1	Receive function request (with or without Dcmp).
T2	Send function response without Cmp.
T3	Send function response with Cmp.
T4	Send read request to DB cache.
T5	Receive response to a read to the DB cache (with or without Dcmp).
T6	Receive response to a read to the DB (with or without Dcmp or Cmp).
T7	Receive response to a write to the DB cache or DB.
T8	Send write request to DB cache or to DB (with or without Cmp).
T9	Send RPC request (with or without Cmp).
T10	Receive RPC response.
T11	Send HTTP request (with or without Cmp).
T12	Receive HTTP response.

T1 and T2 are shown in Figure 6.4a and 6.2a. T3 is like Figure 6.2a except that a Cmp is invoked before Ser; there is no branch because the CPU core knows that it needs to compress the data. T4 is shown in Figure 6.2b. T5 is shown in Figure 6.7, as Figure 6.4b was a simplified version of it without a check for compressed data and a Dcmp.

T6 is shown in Figure 6.7. If the data was not found in the DB, the function returns an error. Otherwise, the data is first potentially decompressed and then, both passed to the CPU and written to the DB cache in parallel. To write to the cache, the data may need to be compressed again if the cache uses compressed data (*C-Compressed* test).



T5
Receive response to a read to the DB cache
TCP
Decr
Dser
Hit?
Y
N
Compressed?
Dcmp
LdB
CPU
Pass data to CPU
Ser
Encr
TCP*
Send request to DB
T6
Receive response to a read to the DB
Found?
RPC
Cmp
C-Compressed?
Write to the DB cache
Function returns error
T7
Receive response to a write to the DB cache or DB
Exception?
Return to the CPU
T8
Send write request to DB cache or DB (with or without compression)
T9
Send RPC request (with or without compression)
T10
Receive RPC response
T11
Send HTTP request (with or without compression)
T12
Receive HTTP response
LDB

Figure 6.7: Additional traces beyond those in Figures 6.2 and 6.4.

T7 in Figure 6.7 is executed when the response from the write to the database cache returns. The same trace is also executed when the response to a write to the database is received. The response may include an exception; in this case, the function error is directly reported to the user by the accelerator ensemble. T8-T12 are also shown in Figure 6.7 and are largely self-explanatory. The response from an RPC (T10) may include an exception that is handled as in T7. In HTTP responses (T12), errors are taken care by the CPU.

6.4.3 Soft Service Level Objectives (SLOs)

In the common case, the requests in the input queue of an accelerator are processed in FIFO order or, if the software has tagged them with priority levels, they are processed in priority order. If the system supports requests under SLOs, AccelFlow is augmented as follows. When a core receives the invocation of a function that has an SLO, as the core generates the DAG of accelerators to invoke, it assigns an absolute deadline to each of the acceleration steps. The deadlines are set by software, and are treated as *soft deadlines.* These deadlines may be a function of the function's inputs—e.g., large-sized data takes longer to compress. These deadlines are then stored in the *first* trace that the core passes with the Enqueue instruction. They will be passed, as part of the trace, from accelerator to accelerator, and then copied from trace to trace, as the ensemble reads new traces from the ATM. These deadlines are absolute times: if one accelerator finishes early, it passes the slack to subsequent accelerators. The traces in the ATM do not contain deadlines: they are read by multiple requests with potentially different deadlines.

When a request gets queued-up in the input queue of an accelerator, the input dispatcher reads its deadline and estimates whether it will meet its deadline. If it will not, the dispatcher checks earlier requests. If any has a large slack, the dispatcher can change the processing order so that both requests meet deadlines. If this is not possible, the dispatcher may allow the request to proceed and record if it ends up violating its SLO.

6.4.4 Accelerator Virtualization

The input/output queues of an accelerator can have entries from different tenants. As the PEs of an accelerator process input entries, the hardware clears the state of a PE and its scratchpad in between execution of inputs from different tenants. This enables multiple tenants to securely use the accelerator ensemble concurrently.

This approach is different from past work, which has proposed coarse-grain accelerator virtualization. For instance, in AuRORA [278], a CPU requests the reservation of an accelerator and, when it is granted, uses the accelerator by itself until it completes. In contrast, AccelFlow uses a fine-grain approach to virtualization. When a CPU core invokes an accelerator with Enqueue, it includes the tenant ID (assigned by the VMM) in the trace. The tenant ID is passed with the trace across accelerators. Each entry in the input and output queues of the accelerators is tagged with the tenant ID that owns the data. With this design, the accelerators are shared in a fine-grained manner, i.e., per request, by multiple tenants.

To prevent tenants from hoarding accelerators, AccelFlow sets a limit to the number of traces that can be executing at a time from an individual tenant. Every time that a CPU core executes an Enqueue instruction on behalf of a given tenant, it increments a count for that tenant; when execution returns to the CPU core, the counter is decremented. If a counter for a tenant reaches a threshold N, no new trace for the tenant can be initiated by a CPU core. Since, in the large majority of cases, a trace can only be using at most one accelerator at a time, this approach ensures that a tenant cannot use more than N accelerators at a time.

This design can be combined with a technique that limits memory bandwidth use by a tenant in the memory controller, such as Intel's MBA (Memory Bandwidth Allocation) [279].

6.5 DETAILED ACCELFLOW IMPLEMENTATION

1. Input Dispatcher. The input dispatcher of an accelerator is a Finite State Machine (FSM) that continuously monitors the *Free?* flags of the PEs in the accelerator and the *Ready?* flags of the entries in the input queue of the accelerator (Figure 6.8). If a *Ready?* and a *Free?* flag are set, the dispatcher may move the corresponding entry of the queue into the corresponding PE and clear the entry. If the entry's data is larger than 2KB, the dispatcher obtains the rest of the data by following the *Memory Pointer* field in the entry (Section 6.4.1). Table 6.3 shows the latency and maximum bandwidth of transferring data from the input queue to the scratchpad of a PE. The transfer is pipelined to improve throughput. In addition, transfers to different PEs use different ports, which allows multiple queue entries to be transferred concurrently.

While the base AccelFlow design processes the input entries in FIFO order, more advanced policies could process the entries based on their *Priority* field (if there are priorities) or *Deadline* field (if the system uses SLOs) (Section 6.4.3). In this case, the *Priority* and *Deadline* fields are set by the software and are carried by the incoming message. If the dispatcher's *Overflow Pointer* is set, as soon as a queue entry is moved into a PE, the dispatcher follows the Overflow pointer and moves an entry from there into the input queue and, if appropriate, clears the Overflow pointer.

2. Output Dispatcher. The output dispatcher of an accelerator is an FSM that continuously monitors the *Ready?* flags of the entries in the output queue of the accelerator

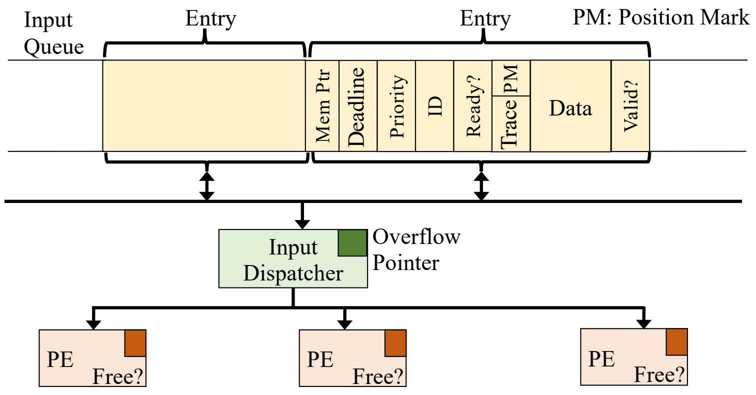


Figure 6.8: Input dispatcher and contents of an input queue entry in AccelFlow.

(Figure 6.9). When an entry with the *Ready?* flag set is found, the dispatcher executes the flowchart of Figure 6.10. First, the dispatcher reads the trace and its Position Mark (PM) into registers (Figure 6.9). It then advances the PM and checks if the next datum is a *Branch* field (which encodes which logic operation to perform on which fields of the payload data). If so, the dispatcher performs these operations on the data still stored in the output queue entry. Based on the result, the dispatcher advances the PM to the correct place of the trace (Figure 6.10). It then saves the PM to the output queue entry.

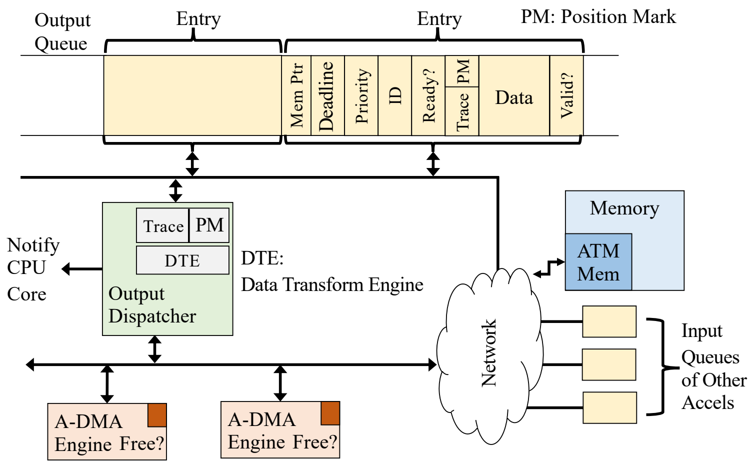


Figure 6.9: Output dispatcher and contents of an output queue entry in AccelFlow.

Next, the dispatcher checks if it has reached the end of the trace. If so, it checks if the tail of the trace has an address. If it does, the dispatcher reads the ATM at this address and loads its contents (i.e., the new trace to execute and PM) into both the output queue entry and the dispatcher registers.

If, instead, the end of trace was reached but there was no address, the dispatcher finds an A-DMA engine with a *Free?* flag set and programs it to move the payload data from the

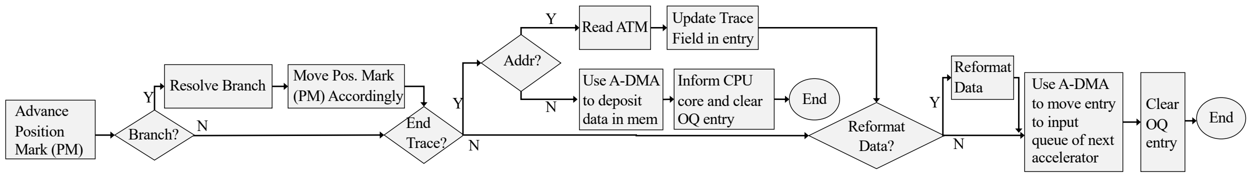

Advance Position Mark (PM)
Branch?
Y
N
Resolve Branch
Move Pos. Mark (PM) Accordingly
End Trace?
Y
N
Addr?
Y
N
Read ATM
Update Trace Field in entry
Use A-DMA to deposit data in mem
Inform CPU core and clear OQ entry
End
Reformat Data?
Y
N
Reformat Data
Use A-DMA to move entry to input queue of next accelerator
Clear OQ entry
End

Figure 6.10: Flowchart of the output dispatcher operation. *OQ* stands for output queue.

Data field in the output queue entry to a memory location (Figure 6.10). Once the engine confirms the end of the transfer, the dispatcher informs the CPU core and clears the output queue entry. The operation ends.

When we are not at the end of the trace or a new trace has been loaded, the dispatcher moves the contents of the output queue entry to the input queue of the accelerator pointed to by the PM in the trace. To perform this task, the dispatcher first checks if the trace contains a *Data Transformation* field. If so, the dispatcher brings the payload data into its Data Transform Engine (Figure 6.9), performs the transformation, and stores the data back to the output queue entry (Figure 6.10). Recall from Section 6.3 that these transformations are very simple. They involve changing between string, BSON, JSON, and similar formats. The engine needed is a simplified form of a (De)Ser accelerator [249], without the support for nested messages or custom data types. If other application domains or accelerators require other transformations, the engine will need to be revisited (e.g., as in DRX [280]).

Then, the dispatcher finds a free A-DMA and programs it to move the output queue entry to the input queue of the next accelerator, setting the *Ready?* bit there. Once the engine confirms the transfer end, the dispatcher clears the output queue entry and terminates the operation. If needed for the transfer, the system correctly handles the logic of the *Memory Pointer* field in the queues and the *Overflow Pointer* in the destination input dispatcher.

3. Interconnection Network, Memory Hierarchy, & Addressing. In the same way as the cores in a chiplet are connected in a mesh, the accelerators in a chiplet are also connected in a mesh (Figure 6.6). The different chiplets are connected with a high-bandwidth highly-connected network. Like the cores, the accelerators access the cache-coherent, distributed LLC of the cores and, if they miss there, they access memory. Accelerator scratchpads and queues are directly addressed and, thus, not cacheable.

Cores and accelerators share the same virtual address space. Each accelerator has a TLB that is accessed by the input and output dispatchers. On a TLB miss, the IOMMU shared by the co-located accelerators loads the correct translation. On a page fault or other exceptions, the accelerator operation stops, CPU is interrupted, and OS handles the exception.

4. Programming AccelFlow. AccelFlow proposes a new programming model and API.

Programmers annotate the code of a service with which sections should be executed on which accelerator. They also create one or more traces, which are graphs of accelerator invocations. A trace does not have glue logic between the accelerators invoked in sequence—except for branch conditions and data format transformations. Branches are encoded as simple bitwise operations on one or multiple fields of an accelerator's output. Data transformations encode source and destination data format.

Currently, programmers construct traces either by using predefined templates (e.g., those in Table 6.2) or by explicitly defining new traces through the API. The API specifies three aspects: accelerator calling graph, branch conditions, and data format transformations. In future work, we will explore ways to automate trace generation via compiler and runtime infrastructures. Our simulator currently takes the instrumented code and the traces, and models the execution in the simulated AccelFlow architecture.

Traces operate within a single microservice and cannot span multiple services, ensuring independent scheduling and avoiding cross-service dependencies. Within a microservice, AccelFlow allows traces to be triggered by network messages such as an RPC request in an event-driven manner, or by a core. Traces remain confined to user-space code and the kernel cannot directly invoke an accelerator on behalf of a microservice. Accelerator execution remains within the control of the microservice runtime. This design ensures simplicity and enhances performance.

6.6 EVALUATION

6.6.1 Evaluation Methodology

Modeled Architectures. We model a server-class processor with 36 cores and 128GB of main memory. Cores and caches are modeled after Intel's Sunny Cove microarchitecture [167, 168, 169] present in the IceLake server processors [97]. The processor has one chiplet with 36 cores and the LdB accelerator, and one chiplet with our remaining 8 accelerators. Table 6.3 shows the architectural parameters.

We model nine accelerators proposed in the literature: F4T [246] for TCP, QTLS [248] for (De)Encryption, Cerebros [44] for RPC, ProtoAcc [249] for (De)Serialization, CDPU [253] (De)Compression, and Intel DLB [257] for load balancing. Each accelerator has 8 PEs.

We evaluate five types of servers. *Non-acc* has no accelerators. The other four servers orchestrate the same set of nine accelerators differently. They are: (1) *CPU-Centric* (as in Section 6.3, accelerators are orchestrated by cores); (2) the state-of-the-art *RELIEF* [260] design (as in Section 6.3, accelerators are orchestrated by a hardware manager); (3) the state-

Table 6.3: Architectural parameters used in evaluation.

Processor Parameters	
Processor	36 6-issue cores at 2.4GHz
OoO Core	352-entry ROB, 200-entry LSQ
L1 D-Cache	48KB, 12-way, 5 cyc. RT, 64B line,16 MSHRs
L1 I-Cache	32KB, 8-way, 5 cyc. RT, 64B line, 16 MSHRs
L2 Cache	512KB, 8-way, 13 cycles RT, 32 MSHRs
LLC Slice	2MB, 16-way, 36 cyc. RT, 32 MSHRs
L1 TLB	128 entries, 4-way, 2 cycles RT
L2 TLB	2048 entries, 8-way, 12 cycles RT
AccelFlow Parameters	
Accel. Queues	64 entries in input queue and 64 in output queue
A-DMA Engines	10
PEs per Accelerator	8
Scratchpad	64 KB per PE in each accelerator
Queue - Scratchpad	10 ns latency and 100 GB/s band. (Max) for 1KB messages
Notification	Average 80 cycles for an accelerator to notify a CPU core
Intra-Chiplet Net	2D mesh, 3 cycles/hop, 16B links
Inter-Chiplet Net	Fully connected, 60 cycles [253], 1Gb/s/link
Main-memory	
Size; Rate	128GB; DDR
Controllers	4 mem. controllers; 4 channels per mem. cntr.
Mem. BW	102.4GB/s per memory controller

of-the-art *Cohort* [263] design (which links pairs of accelerators that frequently go together, but otherwise relies on the cores to orchestrate the accelerators); and (4) *AccelFlow*.

How We Model the Accelerators. An accelerator operates in three steps: it loads its inputs, performs its computation C, and saves the results. Since the RTL-level design of the accelerators is unavailable to us, we can only estimate the time taken by C indirectly. Specifically, the literature provides, for each accelerator and input data set, the speedup S that the accelerator delivers in its computation of C relative to a CPU. Then, in our simulations, we assume that the speedup of the simulated accelerator over a simulated CPU will be the same S. Hence, in our simulations, we model a CPU and measure how many cycles it takes to execute computation C. Then, we assume the accelerator takes $\frac{C}{S}$ time to perform its computation. As a reference, on average across all data sizes observed, the speedups in the literature are 3.5 for F4T, 6.6 for QTLS, 20.5 for Cerebros, 3.8 for ProtoAcc, 4.1 and 15.2 for CDPU (de)compression and 8.1 for Intel LdB.

Simulation Infrastructure. We evaluate the architectures with full-system simulations using QEMU [156] and SST [98]. QEMU captures both user-space and kernel-space instructions, memory accesses, and system calls. QEMU passes all the events to the Ariel core [281] in SST modified for high accuracy. The resulting system models the architectures and performs cycle-accurate simulations. The simulation environment models the whole software stack: OS (Ubuntu 20.04), container runtime (Docker-compose [18]), and the application logic. Main memory is modeled with DRAM-Sim2 [99].

Applications. We run 8 SocialNetwork microservices from DeathStarBench [213]. To model a realistic microservice environment, we take Alibaba's production-level open source traces [5] and pick 8 representative services from them that have similar size and call structure as the 8 SocialNetwork microservices. Then, we use the real-world invocation rates of those Alibaba services in our evaluation. The average load per service is 13.4K requests per second (RPS). We also run experiments with different loads. In this case, we use Poisson distributions for the request inter-arrival time. We use average loads of 5K, 10K, and 15K RPS. These experiments also include the HotelReservation and MediaServices from DeathStarBench, to ensure that our results generalize across applications.

Area Overhead of AccelFlow. We compute the processor area via McPAT [102]. We use the 32nm technology available with the tool, and then scale to 7nm [103]. The total area of our baseline processor without accelerators is 122.3mm^2: 83.1mm^2 are the cores and their private caches, 38.2mm^2 the LLC, and 1.0mm^2 the network.

The literature provides the areas of the (De)Serialization [249] and (De)Compression [253] ASIC accelerators. The area of the Ser, Dser, Cmp, and Dcmp accelerators with 8 PEs and 8 scratchpads each is 0.6mm^2, 0.9mm^2, 9.1mm^2, and 5.2mm^2, respectively. Based on their similar functionality, we estimate the TCP and (De)Encr to have similar area as Dcmp, and the RPC and LdB accelerators to have similar area as the Dser accelerator. These nine accelerators, with 8 PEs and 8 scratchpads each, take 44.9mm^2.

Each accelerator has 64-entry input and output queues, and input/output dispatchers. Each entry in the queues is 2.1KB. For the input/output dispatchers, we consider the worst-case where each dispatcher has the same area as the Deserialization accelerator. These queues and dispatchers for all accelerators take 3.4mm^2, while the 10 A-DMA engines take 1.3mm^2 [282] and the accelerator network takes 0.4mm^2. We do not model the I/O and IOMMU area. The rest of the hardware in the accelerator chiplet takes negligible area.

Of the total processor area, the combination of accelerators, queues, dispatchers, and accelerator network takes 29.0%, while the accelerators themselves take 26.1% of the total area. Hence, the total overhead of AccelFlow is less than 2.9% of the SoC.

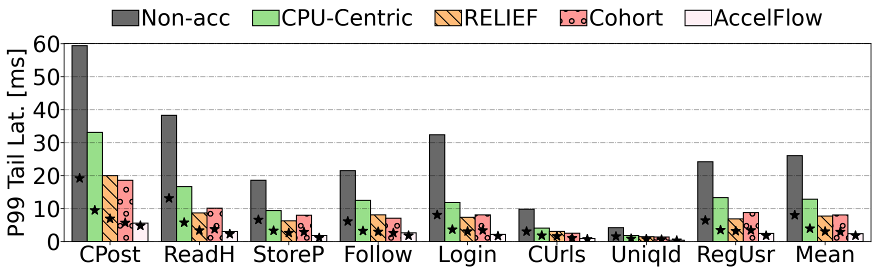


Figure 6.11: P99 tail latency of microservices in different architectures. Black stars indicate the *average* latency of services.

6.6.2 Latency Reduction with AccelFlow

1. End-to-End Tail and Average Latency. The bars in Figure 6.11 show the P99 tail latency of microservices in the five architectures considered. The stars are the average latency. In all services, *AccelFlow* has the shortest tail, followed by *RELIEF* or *Cohort*, then *CPU-Centric*, and then *Non-acc*. On average, *AccelFlow* reduces the tail latency over *Non-acc*, *CPU-Centric*, *RELIEF*, and *Cohort* by 90.7%, 81.2%, 68.8% and 70.1%, respectively. *AccelFlow* attains greater reductions for services that most frequently invoke accelerators (*CPost*), or that have frequent branches in the accelerator trace (*Login*). Then, fast accelerator communication and branch resolution help.

The longer tail latency of RELIEF over AccelFlow does not come from long communication latencies between the accelerators and the RELIEF orchestrator (the latter is in the same chiplet as the accelerators). RELIEF's limitation is that the orchestrator becomes the bottleneck. Ignore the execution time of an accelerator and the communication time between accelerator and orchestrator. Every time that an accelerator finishes, the time for the orchestrator to get interrupted plus to process the information is ≈1.5 μs [260]. A Cpost request uses 87 accelerators (some in parallel) as we will see in Section 6.6.4. Assume a medium load of 10K requests/sec. The time the RELIEF orchestrator is busy for the 10K requests that arrive in 1 second is 1.3 seconds. This causes the longer latency of RELIEF.

Cohort and RELIEF have similar tail latency. Cohort helps reduce its tail latency by allowing some form of static chaining between two or more accelerators (Section 6.6.1). This chaining can be exploited without involving the CPU and, by reducing centralized contention, helps reduce the tail latency in Cohort.

The average latency follows the same trends as the tail, although the impact of AccelFlow is smaller. *AccelFlow* reduces the average latency over *Non-acc*, *CPU-Centric*, *RELIEF*, and *Cohort* by 77.2%, 53.9%, 40.7%, and 37.9%, respectively.

Figure 6.12 shows the P99 tail latency of various groups of DeathStarBench applications

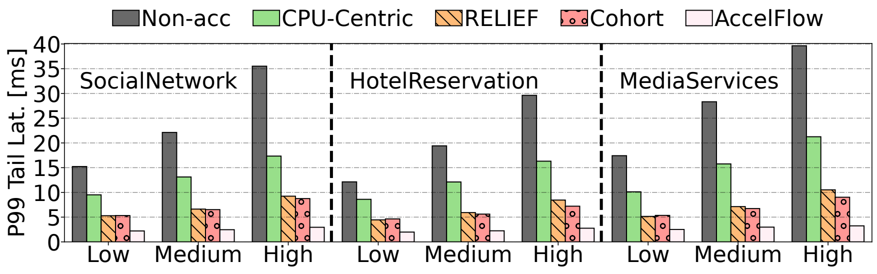


Figure 6.12: P99 tail latency of microservices under different system loads.

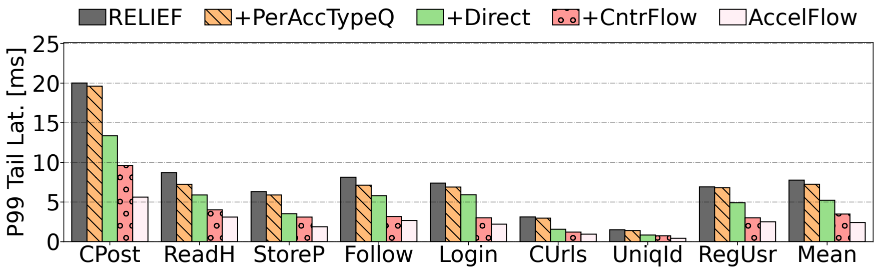


Figure 6.13: P99 tail latency of services with the successive addition of *AccelFlow* techniques.

with various system loads—rather than according to the real-world traces. We compare the architectures for 3 load levels: Low (5K RPS), Medium (10K RPS), and High (15K RPS). We see that *AccelFlow* significantly reduces the tail latency across all loads. However, it is relatively more effective at higher loads because it relieves the contention on the centralized manager. For example, on average, it reduces the tail latency over *RELIEF* by 55.1%, 60.9%, and 68.3% for 5K, 10K, and 15K RPS, respectively.

2. End-to-End Tail Latency Breakdown. Figure 6.13 shows the contributions of the main techniques in *AccelFlow* that reduce tail latency with real-world production traces. We apply these techniques one by one. In *RELIEF*, all 8 PEs of all 9 accelerator types share a single centralized queue. With *PerAccTypeQ*, we distribute the queue so that there is a queue for each accelerator type. *Direct* additionally uses traces and supports direct data transfer between sequences of accelerators, eliminating the need for hardware manager intervention. *CntrFlow* additionally upgrades output dispatchers to resolve branches in the trace, eliminating CPU fallbacks. Finally, *AccelFlow* additionally upgrades dispatchers to perform data format transformations and handle large input data that does not fit in an input queue entry without involving the CPU.

All these techniques are effective. A separate queue per accelerator type (PerAccTypeQ) reduces contention and improves load balance. Direct accelerator-to-accelerator communication is effective, especially for CPost, which has long accelerator sequences. Having dispatchers that resolve branches in traces is beneficial, especially in services with frequent

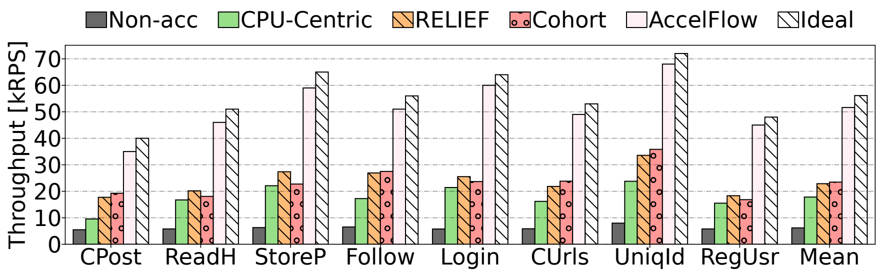


Figure 6.14: Maximum throughput of services in the five architectures plus an *Ideal* one.

dynamic control flow such as Login. Finally, processing large payloads and performing data transformations is also effective. Overall, applying these techniques reduces the average tail latency by 6.8%, 32.7%, 55.1%, and 68.7%.

3. Throughput Improvement with AccelFlow. Figure 6.14 shows the maximum throughput that the architectures attain for the different services. This is the maximum load that the architectures can sustain without violating the service SLO. We define the SLO to be 5× the service execution time on an unloaded system [283, 284]. The figure also shows an *Ideal* system that allows the accelerators to communicate directly without incurring the overheads of branch resolution or data transformations. Accelerators improve request throughput, but their impact is sensitive to how they are orchestrated. On average, *AccelFlow* improves throughput over *Non-acc* by 8.3×, while *RELIEF* only manages 3.7×. Further, *AccelFlow* is within 8.0% of the throughput of *Ideal*.

To further improve the performance, AccelFlow can use advanced scheduling policies instead of FIFO. We re-evaluate AccelFlow with a policy that prioritizes those requests that are closer to their deadline (Section 6.4.3). It can be shown that, with such a policy, AccelFlow improves the throughput by an additional 1.6×.

6.6.3 Sensitivity Analyses

1. Processor Organization into Chiplets. Our processor has a core chiplet and an accelerator chiplet. One could design the processor with one chiplet to reduce communication overheads, or with more chiplets to add flexibility. We evaluate P99 tail latency of microservices for different numbers of chiplets: *1-chiplet*; *2-chiplets* (the base design); *3-chiplets* (TCP and (De)Encr in one chiplet; RPC, (De)Ser, and (De)Cmp in another); *4-chiplets* (TCP and (De)Encr in one chiplet; RPC and (De)Ser in another; (De)Cmp in another); and *6-chiplets* (TCP, (De)Encr, RPC, (De)Ser, and (De)Cmp in separate chiplets). As we separate the accelerators into more chiplets, the inter-accelerator communication is more expensive. As a result, the tail latency of requests increases. The impact is significant:

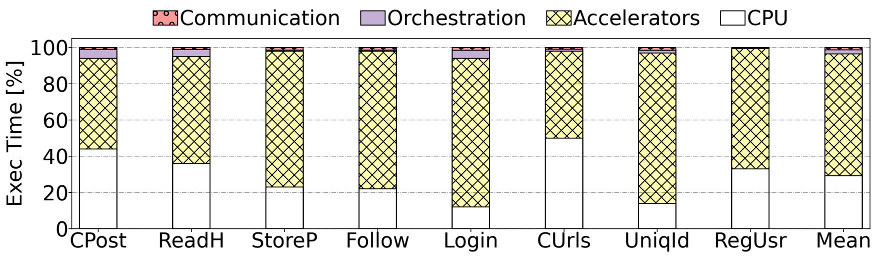


Figure 6.15: Breakdown of the service execution time.

going from 2 to 6 chiplets increases the tail latency by 14% on average.

2. Inter-chiplet Latency. Our default inter-chiplet latency is 60 cycles [253]. We also evaluate the tail latency of microservices for *AccelFlow* with different numbers of chiplets as we vary the inter-chiplet communication latency from 20 to 100 cycles. Inter-chiplet latency becomes more important as we increase the number of chiplets. It can be shown that, as we go from 60 to 100 cycles in 6-chiplet systems, the average tail latency increases by 45%.

3. Number of Accelerator PEs. Our base design has 8 PEs of each accelerator type. We also evaluate tail latency of microservices under real-world invocation traces with different number of PEs per accelerator in *AccelFlow*. It can be shown that on average, with 2 or 4 accelerator PEs, the tail latency of our base system increases by 35.8% and 20.1%, respectively. On average, with 16 or 36 accelerator PEs, the tail latency of our base system reduces by 6.2% and 12.3%.

6.6.4 Characterizing AccelFlow Operation

1. Components of the Execution Time. Figure 6.15 breaks down the execution time of a microservice request in AccelFlow. It shows the time spent running on the CPU, running on the accelerators, executing the orchestration logic (dispatchers), and communicating between engines (i.e., accelerators, CPUs). The experiment runs on an unloaded system, i.e., with one request at a time, to avoid the effects of contention. For the majority of services, the time spent executing on accelerators dominates the total execution time, while the time spent on the orchestration logic is on average only 2.2%. This is in contrast to existing schemes. For example, it can be shown that RELIEF incurs an orchestration overheads of ≈10%.

2. Glue Instructions between Accelerators. When a sequence of accelerators executes, the output dispatcher of each accelerator in the sequence follows the flowchart of Figure 6.10. In most cases, an output dispatcher finds no branch, no end of trace, and no data transformation. In this case, the operation of the output dispatcher takes the equivalent of about 15 RISC instructions.

Service	**Execution Path**	**# Accels**
CPost	T1-CPU-4x(T9-T10)-CPU-3x(T9-T10)-CPU-T2	87
ReadH	T1-CPU-T4-T5-CPU-T9-T10-CPU-T3	28
StoreP	T1-CPU-T8-T7-CPU-T2	18
Follow	T1-CPU-3x(T8-T7)-CPU-T2	30
Login	T1-CPU-T4-T5-T6-T7-CPU-T2	29
CUrls	T1-CPU-T8-T7-CPU-T3	19
UniqId	T1-CPU-T2	9
RegUsr	T1-CPU-T8-T7-CPU-T9-T10-CPU-T2	25

Table 6.4: Per-service execution path and total number of accelerators used per invocation.

When the output dispatcher finds a branch, it resolves the branch and moves the Position Mark accordingly (Figure 6.10). In our services, the possible branch conditions are: Compressed?, Exception?, Hit?, and Found?. Resolving them involves checking a single-bit field in the output queue entry. On average, processing a branch adds the equivalent of about 7 additional RISC instructions.

If an end of trace is found, the dispatcher either reads an ATM location and moves the trace in it to an input queue entry in an accelerator, or it invokes a DMA to deposit the output data to memory, informs a CPU core, and clears the output queue entry. These operations take the equivalent of 12 to 20 RISC instructions.

If a data format transformation field is found, the dispatcher loads the source data in bulk, invokes the data transformation engine (Figure 6.9), and stores the data back to the output queue entry in bulk. The number of RISC instructions executed by the dispatcher is 12 for 2KB payloads. Overall, in the worst case, an output dispatcher executes ≈50 RISC instructions. In our services, the average number of instructions per output dispatcher operation is ≈18.

3. Characterizing Traces. We analyze the execution of each service in our benchmark suite and determine which traces of Table 6.2 it uses, in what sequence, and how many accelerators it invokes. Table 6.4 shows the per-service execution path and the total number of accelerators used per service invocation. We refer to traces as T_i, as defined in Table 6.2. For example, the CPost service executes trace T1, then goes to the CPU, then executes 4 parallel invocations of the sequence of traces T9 and T10, returns to the CPU, then executes 3 parallel invocations of traces T9 and T10, returns to the CPU, and finally executes trace T2. Overall, services use between 2 and 16 traces, and between 9 and 87 accelerators per service invocation.

6.7 RELATED WORK

Datacenter Tax Profiling. Hyperscalers studied how cycles are spent in datacenters [14, 15, 225, 243, 245, 285, 286, 287]. They observed that the tax consumes a large fraction of cycles, which can be mitigated with hardware accelerators, potentially chained into a *sea of accelerators* [245]. The profiling was done at a relatively high level, aggregating the results across many systems. There is not enough information to get insights into how to orchestrate multiple-accelerator execution. They focus on data analysis and not on system design.

Accelerators for Microservices. Many works explored the design of accelerators for individual sources of datacenter tax [43, 44, 45, 46, 114, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258]. They efficiently reduce individual overheads, such as TCP [246] or RPC [44, 46, 114] processing. In contrast, AccelFlow is a scheme to efficiently orchestrate multiple accelerators. Any accelerator with the standard interface can be integrated. Researchers also proposed accelerators for application logic [288, 289, 290, 291, 292], such as key-value stores [292] or databases [289]. AccelFlow can seamlessly integrate such accelerators. Finally, researchers proposed specialized CPU architectures for microservices [31, 115, 185, 293]. AccelFlow can be implemented on top of different processor architectures.

Accelerator Orchestration. Researchers proposed scheduling schemes for processors equipped with accelerators [259, 260, 261, 262, 263, 278, 294, 295, 296, 297]. Many rely on a CPU core [259] or centralized hardware manager [260, 261] to orchestrate the accelerators. Cohort [263] uses a software-oriented acceleration framework based on shared-memory software queues to link accelerators. In the chapter, we compared AccelFlow to Cohort [263] and RELIEF [260]. VIP [262] proposes a scheme that chains several accelerators so they appear to the software as a single device. VIP is applied to coarse-grained video-processing applications that always use the same set of accelerators in a chain; it cannot support the fine-grained, dynamic behavior of microservices, which include dynamic control flows. AuRORA [278] proposes a coarse-grained scheme to virtualize accelerators for multi-tenant DNN workloads. AuRORA is decentralized, but it does not orchestrate accelerator chains.

6.8 CONCLUSION

This chapter presented *AccelFlow*, the first accelerator orchestration framework for microservices. In AccelFlow, CPU cores build *Traces* that contain sequences of accelerators to call and, potentially, branch conditions. The accelerators in the trace execute in sequence without involving the CPU. Compared to the state-of-the-art, AccelFlow reduces P99 latency by 70% and increases throughput by 120%.

CHAPTER 7: Efficient Resource Utilization in Cloud-Native Services

7.1 INTRODUCTION

Despite the attractive benefits of serverless computing, current serverless workloads suffer multiple overheads. The most important ones include lengthy startup latency [53, 63, 86, 187, 188, 298], large memory footprints that limit how many containers can be executing concurrently [64], and frequent execution stalls due to RPC/HTTP invocations to call functions or to access remote storage [189, 190, 191, 192, 193, 194, 195]. Existing techniques address some of these shortcomings, but a lot of improvement is still needed. For example, to reduce startup overhead, some systems keep the state of an idle container in memory for a long time [63, 64, 65], aggravating memory limitations.

Recent measurements have revealed that serverless workloads frequently exhibit bursts of invocations of the same function [63, 299], either from different end-users or from a single one. Different end-users can create bursts of invocations to popular functions, triggered by certain events. A single end-user may issue thousands of invocations of the same function in seconds—e.g., in serverless video processing systems like ExCamera and Sprocket, to parallelize real-time video encoding [25, 26]. In response to either case, current serverless platforms spawn and execute many containers concurrently.

An analysis of state-of-the-art platforms shows how inefficiently this pattern is supported. First, the execution of a function is most of the time blocked on synchronous wait operations, and cores either remain idle or frequently context switch between containers of different functions. Hence, response times degrade. Second, concurrent execution of multiple invocations of the same function causes repeated I/O accesses to the same or similar data, and calls to the same remote functions. The result is inefficient I/O bandwidth use. Finally, the different invocations of the same function largely bring the same state to memory. If the system does not allow memory sharing between invocations, the replicated state consumes substantial memory, inhibiting the execution of other containers and increasing their startup overhead.

To improve performance under this typical behavior, this chapter introduces a new serverless platform design named *MXFaaS* or *Multiplexed FaaS*. MXFaaS improves performance by efficiently multiplexing (i.e., sharing) processor cycles, I/O bandwidth, and memory/processor state between concurrently-executing invocations of the same function.

MXFaaS introduces *MXContainer*, a new container abstraction that can concurrently execute multiple invocations of a single function and owns a set of cores. An MXContainer has a *Dispatcher* process and multiple *Handler* processes. The dispatcher initializes the container in the first function invocation. At every function invocation, the dispatcher forks a handler to serve the request. MXFaaS introduces three techniques, which address each of the three aforementioned inefficiencies.

First, to enable efficient use of processor cycles, the dispatcher in an MXContainer carefully suspends and resumes its handlers. Its aim is to ensure that, at any time, the OS can schedule the *oldest N ready-to-execute invocations* of the function—where N is the number of cores owned by the MXContainer. The resulting execution minimizes function response time.

Second, to enable efficient use of I/O bandwidth, the dispatcher coalesces remote storage accesses and remote function calls from multiple invocations of the same function. The coalesced storage requests can be for a single key or for a vector of them; the coalesced function calls are for the same function. The result is lower network demands and reduced pressure on storage and processors.

Finally, to enable efficient use of memory/processor state, the dispatcher first initializes the state of the container and only later, on demand, spawns a handler process per function invocation. With this design, all invocations share the unmodified initialization state (which may comprise MBs of memory), reducing overall memory footprint and allowing more instances to reside in memory at a time. Further, by executing these multiple invocations of the same function on the owned cores, one also reuses the cache and branch predictor state.

There are some prior schemes that enable some reuse of memory state across concurrent invocations of the same function [86, 189, 300, 301]—but not to the extent of our proposal. The details are in Section 7.9. No prior scheme combines the use of CPU cycles or I/O bandwidth across concurrent invocations of the same function.

We implement MXFaaS in the Apache OpenWhisk [57] and KNative [60] platforms. MXFaaS does not require any hardware or operating system (OS) support, or changes to user functions. We evaluate MXFaaS with a diverse set of serverless benchmarks and show that MXFaaS is very effective. Compared to a state-of-the-art serverless baseline [189], MXFaaS on average speeds-up execution by 5.2×, reduces the P99 tail latency by 7.4×, and improves throughput by 4.8×. In addition, it reduces the average memory usage by 3.4×. Finally, MXFaaS outperforms an ideal scheme that predicts which containers will be needed next and proactively warms them up, by an average of 2.1× (or 2.9× for high load).

This chapter makes the following contributions:

• A characterization of state-of-the-art serverless systems.
• The MXContainer abstraction.
• The MXFaaS serverless platform that enables efficient use of processor cycles, I/O bandwidth, and memory state.
• An implementation and evaluation of MXFaaS.

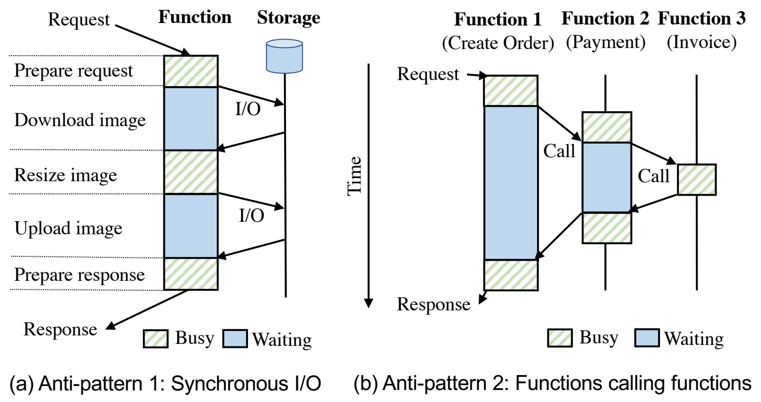


Figure 7.1: Inefficient function patterns: (a) synchronous I/O within a function, and (b) functions calling functions.

7.2 CHARACTERIZING RESOURCE INEFFICIENCIES IN FAAS ENVIRONMENTS

To understand the performance of serverless environments, this section analyzes real-world serverless workloads and open-source serverless benchmarks running on OpenWhisk [57]. For the former, we use production traces from Azure's serverless functions [191, 226] and Alibaba's microservices [5]; for the latter, we use functions from FunctionBench [171] and applications from TrainTicket [78, 302]. We give more details in Section 7.6.

7.2.1 Inefficient Patterns

We observe a few inefficiencies in function implementation, application orchestration, and resource provisioning.

Synchronous I/O within a Function. Serverless functions rely on remote storage to maintain state. Functions are often of millisecond scale [63], and storage I/O can easily dominate function execution time. Therefore, synchronous I/O in function code, as shown in Figure 7.1a, is strongly discouraged as an anti-pattern [303]. Since synchronous I/O is often needed due to data dependencies, the recommendation is to split a function into multiple smaller functions and perform the I/O in between two functions. However, it may not always be feasible to prevent synchronous I/O in functions. Moreover, developers often fail to use disciplined coding practices.

Functions Calling Other Functions. To orchestrate serverless applications, it is common to let a function call other dependent functions—which resembles procedure calls in traditional programming. Such practice is also an anti-pattern because such RPCs result in a compound effect of synchronous waits, as shown in Figure 7.1b. Although this anti-pattern is also strongly discouraged [304], we find that it is prevalent in existing serverless applications. One reason is that many serverless applications originate from traditional microservice applications that use RPCs to orchestrate applications.

Minimizing Startup Time. To minimize the start-up overhead of function invocations, a number of optimizations have been developed. One approach is to keep an idle container in memory for long, so its process and state can be reused when a subsequent invocation of the same function arrives [63, 64, 65]. This optimization tends to increase the memory footprint—potentially preventing other functions from executing concurrently due to lack of memory. Reusing the process in a container also breaks the isolation expected of containers: the state generated by one function invocation is visible to the next invocation [62].

Another approach is to keep container snapshots in disk and pre-load one when a request for the corresponding function arrives [53, 86]. This approach and the previous one speed-up the startup of individual requests, rather than targeting many concurrent requests in a burst. Hence, on a burst, they consume substantial memory or create substantial disk traffic.

Other schemes such as SAND [300] and Faastlane [193] minimize startup time by creating a single container for all the different functions of an application. However, this design makes it harder to efficiently manage the hardware resources per container and scale the number of containers, because the different functions in a container may have very different hardware requirements and software dependencies. For scalability and resource management efficiency, it is best to keep different functions in different containers.

7.2.2 Workload and Execution Characteristics

Invocations of the Same Function are Bursty. Serverless workloads exhibit bursts of invocations of the same function [63, 299]. Figure 7.2 shows the distribution of the number of concurrent invocations of the same function in real-world workloads, based on production FaaS traces from Azure [226] and microservice traces from Alibaba [5]. The figure shows the CDF distribution. In Alibaba, 50% of the invocations of a function are in bursts of 32 or more concurrent invocations of the function. Azure traces are less skewed, but still, 20% of the invocations of a function are in bursts of 8 or more concurrent invocations. This data is a result of the goal of the serverless computing model to promote autoscaling and elasticity.

Idle Time Dominates Function Execution. We take serverless functions from FunctionBench [171] and serverless applications from TrainTicket [302], and measure the idle time during the execution of each function. We find that the two stall patterns of Section 7.2.1 are prevalent. All the 47 functions in the two suites exhibit one of the two patterns. We also inspect other serverless benchmarks [192, 227, 228, 229] and observe the same patterns.

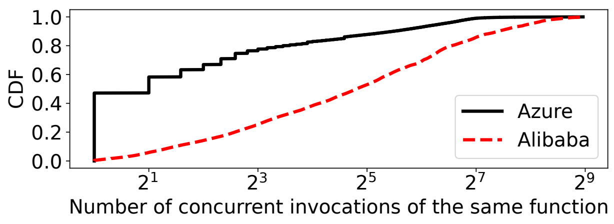
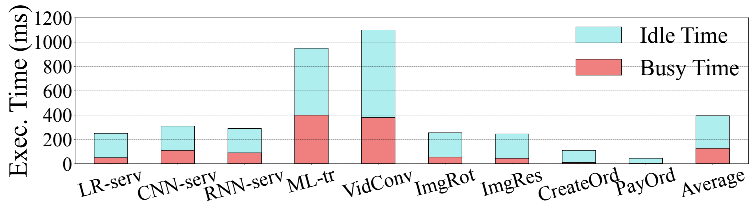
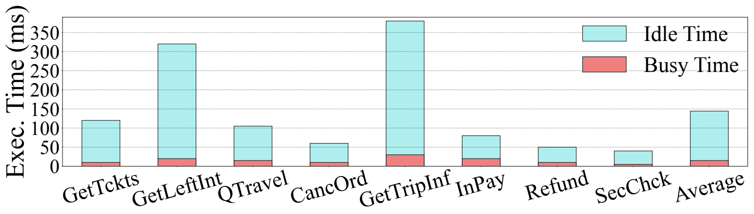


Figure 7.2: Concurrent invocations of the same function in Azure and Alibaba FaaS traces.



(a) Functions that invoke synchronous I/O.



(b) Functions that call other functions.

Figure 7.3: Busy and idle time of functions from FunctionBench and TrainTicket.

Figure 7.3 shows the busy and idle time of a representative set of these functions. Figure 7.3a shows functions that invoke synchronous I/O. On average, 68% of the execution time of these functions is taken by idle time. It can be shown that all the 14 functions in FunctionBench issue synchronous I/O requests, following the procedure of Figure 7.1a, where the code first downloads data from remote storage, then processes it, and finally uploads the results to the storage.

Figure 7.3b shows functions that call other functions. On average, 90% of the execution time of these functions is idle time. In TrainTicket, it is common to have a calling pattern as in Figure 7.1b. Of the 33 functions in TrainTicket, 13 have RPCs and the remaining ones issue synchronous I/O (e.g., CreateOrd and PayOrd in Figure 7.3a). The 13 functions that use RPCs issue on average 4.8 RPCs. As the leaf functions use synchronous I/O, the inefficiency propagates along the call chains (Figure 7.1b).

There Is Substantial State Replication in Memory. When multiple invocations of the same function execute concurrently, they frequently access the same data and instructions. Unless a deliberate effort is made to ensure that the invocations share pages, a lot of data will be replicated in memory. The resulting large memory footprint will limit the number of containers that can reside in memory at a time and hurt throughput.

To understand the extent of the problem, we measure the memory footprint of each individual function in Figure 7.3a and break it into the three categories shown in Figure 7.4: *LibLd* is the footprint of the shared libraries; *Init* is the footprint of read-only data that is function-specific and independent of individual invocations of the function; and *Handler* is the footprint of the per-invocation private data. The bars are normalized to 1 and, on top of each, we show the total footprint in Mbytes. On average, *LibLd*, *Init*, and *Handler* account for 66%, 24%, and 10% of the total footprint, respectively.

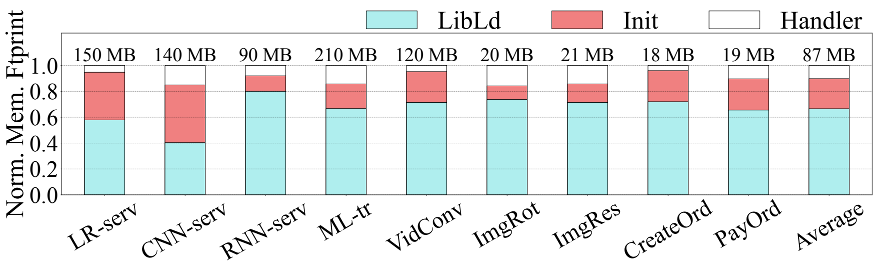


Figure 7.4: Breakdown of the normalized memory footprint.

Under different FaaS schemes, concurrent invocations of the same function can share different parts of the memory footprint. Specifically, schemes that spawn a VM for a function invocation from a previously-generated snapshot (SEUSS [188], REAP [53]) do not enable the sharing of any of these categories. The same is mostly true for schemes that fork the execution of a function invocation from a template (SOCK [301] and Catalyzer's sfork [86]). Schemes that load the shared libraries into the container before forking a process to execute the function invocation (SAND [300] and process-based Nightcore [189]) enable the sharing of the *LibLd* footprint across function invocations. They save substantial memory.

In this chapter, we note that there is still a substantial amount of memory footprint that can be shared across invocations of the same function: the read-only data that is function-specific and independent of individual invocations of the function (*Init*). The average footprint of such data in Figure 7.4 is 20.4 MB. Our proposal with MXContainer will be to delay the forking of a process for an invocation until such data is initialized once by a special process (Dispatcher). As a result, all function invocations will automatically share this data. Based on the numbers in Figure 7.4, this approach will allow us to keep in memory on average 3.4× more concurrent function invocations than process-based Nightcore. The result will be much higher concurrency and throughput.

There Are Concurrent Accesses to the Same Storage Locations. All the concurrent invocations of the same function execute the same code and, intuitively, should access the same storage area for the same or similar data at similar times. If this is true, there is an opportunity to merge the accesses to save I/O bandwidth. An analysis of production FaaS Azure traces [191, 226] shows that 12% of the applications access the same data blob in all of their invocations. Moreover, invocations access relatively few different blobs: 66% of the applications access less than 100 different blobs across all invocations. More importantly, a given blob is accessed in a bursty manner, offering opportunities for access merging. Figure 7.5 shows the CDF of the interarrival time of accesses to the same blob. It can be seen that 18% and 54% of the accesses to a given blob happen within 1ms and 10ms, respectively.

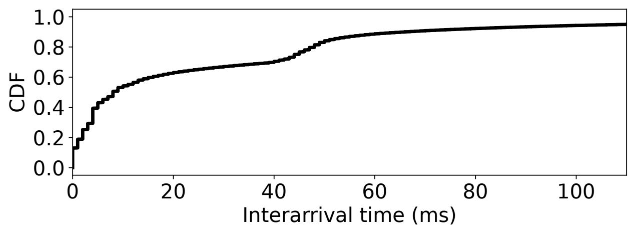


Figure 7.5: Inter-arrival time of accesses to the same blob.

In addition, blobs are typically small: 80% are smaller than 12KB. Hence, accessing many blobs in parallel, for the same or different data, creates a network bottleneck—not due to data volume, but due to connection overheads. Sharing and reusing connections for data transmission can reduce the bottleneck.

7.2.3 Implications

Our analysis has revealed a few key bottlenecks in serverless environments with bursty invocations of functions. First, functions are blocked on synchronous wait operations most of the time. Hence, unless cores are scheduled intelligently, the response time of function invocations can easily degrade substantially. Second, nodes issue similar requests to storage and invoke similar functions. The result is unnecessary I/O bandwidth consumption and pressure on storage and processors. Finally, containers consume substantial memory with replicated state. As a result, serverless systems are often limited by available memory.

7.3 MXFAAS OVERVIEW

To eliminate the bottlenecks uncovered in our characterization section, we now propose a new serverless platform design called *MXFaaS* or *Multiplexed FaaS*. MXFaaS optimizes execution during bursts of invocation requests for the same function—a typical occurrence in

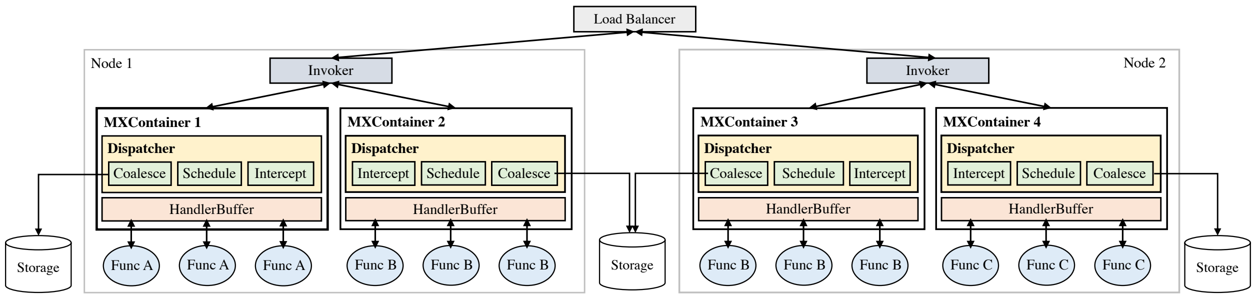


Figure 7.6: Overview of the MXFaaS serverless platform. The blue circles represent cores.

serverless environments. Unlike current platforms, MXFaaS leverages the synergies between these concurrent requests. More specifically, it efficiently multiplexes (i.e., shares) processor cycles, I/O bandwidth, and memory/processor state between concurrent invocations of the same function. The result is a higher throughput and lower latency serverless environment.

MXFaaS introduces a new container abstraction called *MXContainer* or *Multiplexed Container*, which can concurrently execute multiple invocations of the same function and owns a (potentially changing) set of cores. An MXContainer has a *Dispatcher* process and multiple *Handler* processes. The dispatcher initializes the container in the first function invocation. At every function invocation, the dispatcher forks a handler to serve the request. The multiple handlers concurrently execute invocations of the same function on different cores.

MXFaaS introduces three techniques, which improve the utilization of three key resources. First, to enable efficient use of processor cycles, the dispatcher in an MXContainer carefully suspends and resumes its handlers. Recall that a typical function execution is blocked most of the time, due to accesses to remote storage or to calls to other functions. Therefore, cores either remain idle for large periods or frequently context switch between containers of different functions. However, in an MXContainer, since the dispatcher has forked the handlers, the dispatcher can suspend and resume them. The dispatcher's aim is that, at any time, the OS can only schedule the *oldest N ready-to-execute invocations* of the function—where N is the number of cores currently assigned to the MXContainer. The dispatcher buffers the remaining set of invocations of the function, whose handlers may or may not be blocked on I/O or function calls. The result is efficient execution that minimizes average and tail response time.

Second, to enable efficient use of I/O bandwidth, the dispatcher combines remote storage accesses and remote function calls from multiple handlers running invocations of the same function. To support storage request combining, the dispatcher keeps a table with the outstanding storage accesses. When the dispatcher is about to issue a remote request, it checks the table and, if there is a matching request, it combines the two accesses. If there is no matching request, the dispatcher waits for some time before issuing the request—in the hope that an upcoming request can be combined with it. Combined storage requests can refer to a single key or to a vector of them.

In addition, the dispatcher combines function calls to the same function—for the same or different inputs. The overall result of combining storage accesses and remote function calls is lower network bandwidth needs and reduced pressure on storage and processors.

Third, to enable efficient use of memory/processor state, the dispatcher first initializes the MXContainer state and after that, on demand, spawns a handler process per function invocation. With this support, all invocations share the unmodified initialization state (*LibLd* plus *Init* in Section 7.2.2 and Figure 7.4)—while protecting their private data via copy-on-write. The result is a reduced memory footprint, which enables more containers to reside in memory at a time and, therefore, effectively reduces startup overhead.

Further, by executing these multiple invocations of the same function on the owned cores, the MXContainer also enables reuse of the cache and branch predictor state.

7.4 MXFAAS DESIGN

Figure 7.6 shows the MXFaaS serverless platform. It has a Load Balancer and, in each node, an Invoker and one or more MXContainers. An MXContainer manages the concurrent execution of multiple invocations of a function on a node, and owns a (dynamically changing) set of local cores. A node can have MXContainers for different functions, but at most only one for a given function. Different nodes may have MXContainers for the same function.

The dispatcher in an MXContainer admits requests for the function. It buffers those that are: blocked on I/O and unable to run, or ready to execute but lack a core to run on. It only allows the OS to schedule as many ready-to-execute invocations as cores the MXContainer owns. The dispatcher regularly informs the node's Invoker of its buffer's utilization.

When an Invoker observes that a local MXContainer becomes overloaded or underloaded, it dynamically changes the number of local cores assigned to it. When an MXContainer is overloaded and unable to get more cores, the global Load Balancer is informed. Then, the Load Balancer allocates another MXContainer for the same function in another node. From then on, the Load Balancer dynamically shares the load between the two MXContainers.

7.4.1 MXContainers for Sharing Processor Cycles

As shown in Section 7.2.2, a typical serverless function spends most of its time blocked, waiting for data from remote storage or for the return of an RPC. In some current systems, the OS does not preempt the blocked request because the FaaS platform has purposely limited the number of concurrently-running requests. In other systems, the FaaS platform allows over-subscription. Hence, the OS preempts the blocked request and schedules another request for the same or another function. Unfortunately, this operation is inefficient without special support: the OS interleaves the execution of multiple containers without deliberately trying to complete older function requests first. The result is degraded average and tail response time.

The MXContainer approach solves this problem by having the dispatcher help manage the scheduling of the handlers. Recall that the dispatcher has spawned the handler processes and, hence, can suspend/resume them. In an MXContainer, the dispatcher maintains a buffer (*HandlerBuffer*) with the handlers that are ineligible to run. These handlers correspond to function invocations that: (1) are blocked on I/O or RPCs and therefore unable to run, or (2) are ready to run but are not the oldest N ready-to-execute invocations—where N is the number of cores currently owned by the MXContainer. Effectively, the dispatcher only allows the OS to schedule the handlers for the oldest N ready-to-execute invocations of the function; the rest are kept buffered in *HandlerBuffer*. This functionality minimizes average and tail response time.

This functionality is supported as follows. First, when the dispatcher initially forks a handler process for a request, the dispatcher (1) records the handler's sequence order and (2) if all the owned cores are busy, it suspends and buffers the handler in *HandlerBuffer*, marking it as *Ready*. Second, when a running handler reaches a blocking call, the call is redirected to the dispatcher, which buffers the handler in *HandlerBuffer*, marking it as *Blocked*. Finally, when the response for the remote storage access or RPC call is received, the dispatcher intercepts it, passes the value to the corresponding handler and, depending on the handler's sequence order, it either (1) keeps the handler suspended in *HandlerBuffer*, now marked as *Ready*, or (2) resumes this handler and suspends a younger, running handler of the same function. Again, the dispatcher can do this because it has spawned both handlers.

Figure 7.7 shows an example of this mechanism. Figure 7.7a shows a possible timeline of a function execution; the function spends some time waiting for I/O. Figure 7.7b shows the execution of six invocations of the same function in an MXContainer that owns two cores. The invocations are ordered based on arrival time from left to right. An invocation can be either using the CPU (*Busy*) or buffered in *HandlerBuffer* marked as *Blocked* or *Ready*. At time t_0, the dispatcher picks the two oldest invocations: *Invoc1* and *Invoc2*. At t_1, it picks *Invoc3* and *Invoc4* over *Invoc5* and *Invoc6*. At t_2 and t_3, it again picks older invocations over *Invoc5* and *Invoc6*.

Using the Shortest Remaining Processing Time first (SRPT) algorithm can further reduce

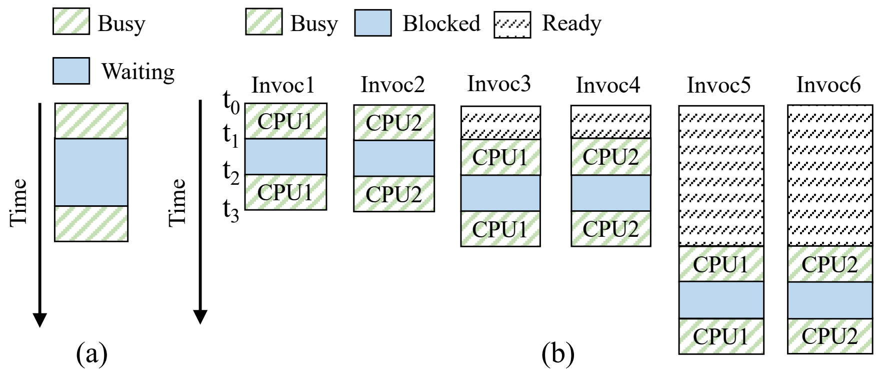


Figure 7.7: Interleaving of function invocations in two CPUs.

the response times when the execution time of the requests has a high variation. In practice, the requests of a given function in FaaS environments are of similar size and duration even when using different inputs [197, 305]. Moreover, SRPT requires estimating the remaining execution time. Consequently, we do not use SRPT.

Overall, in MXContainers, function invocations share processor cycles in a way that minimizes average and tail response time.

7.4.2 MXContainers for Sharing I/O Bandwidth

As multiple handlers in an MXContainer concurrently execute multiple invocations of the same function, these handlers are likely to issue requests for the same storage area (and potentially even the same keys). They are also likely to issue RPCs for the same functions, possibly even using the same argument values. Recall that the dispatcher intercepts all these blocking requests. This fact offers the ability to combine storage accesses or RPCs from multiple handlers—minimizing the network load and the pressure on storage and CPUs. We consider the two types of combining.

Remote Storage Access Combining. To combine remote storage accesses, the dispatcher keeps a software *Miss Status Holding Table* (MSHT). The MSHT has an entry for each outstanding storage access from this MXContainer. It is analogous to the hardware structure that records outstanding cache misses in cores.

When the dispatcher is about to issue a read to remote storage, it checks the MSHT. If there is already an outstanding read to the same key, the dispatcher issues no request. Instead, it combines the two read requests by augmenting the existing MSHT entry with additional information. When the key is received by the node, it will be passed to both the initial and the new requesting handlers.

If the dispatcher wants to issue a read and there is an outstanding write to the same key, or wants to issue a write and there is an outstanding access to the same key, the dispatcher delays its request until the previous access completes.

If the dispatcher is about to issue a remote storage access and does not find an existing entry in the MSHT for the key, it waits a certain time period (T_{merge}) before issuing the request. The goal is to coalesce the request with any subsequent requests to the same or different key that may come within a small time period—and therefore issue a single request instead of several. When the dispatcher combines requests for different keys, it issues one vectorized request to the remote storage. The MSHT records which handler accessed which key. When the dispatcher receives the response, it unfolds the vector and forwards the correct values to the appropriate reading handlers.

Figure 7.8 is an example of accesses to different keys without (a) and with (b) coalescing.

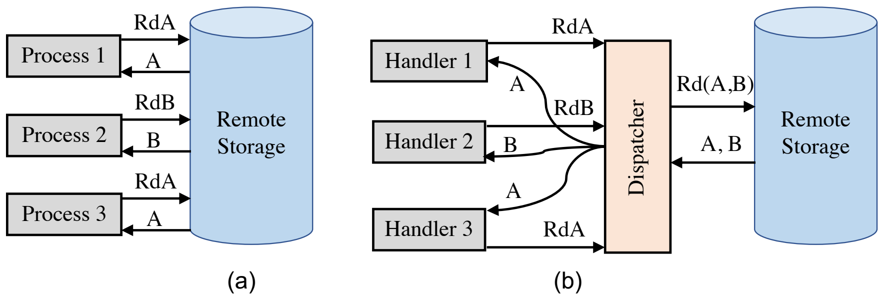


Figure 7.8: Storage access coalescing in MXFaaS.

When the load is low, delaying a request may not result in a merging event and, instead, can cause an increase in average and tail latency. Thus, the dispatcher monitors the load and dynamically decides whether to enable merging.

Function Call Combining A similar strategy could be used to combine RPCs to functions. However, functions may have side effects, which means that the outcome of two calls with the same argument values may be different than the outcome of one single call. Hence, the safe approach to combining involves delaying the RPC for T_{merge} cycles and, if other RPCs to the same function are detected in the meantime (with or without the same argument values), bundle them all in a single I/O transaction that requires executing all the function calls at the destination node.

A special case is functions that, when invoked with the same inputs, produce the same outputs and have no side effects. If the programmer knows that a function behaves in this way, she can annotate the function as *pure*. For pure functions, the dispatcher maintains a table recording the set of {inputs, outputs} tuples observed in the past. When the dispatcher is about to call a pure function with certain input values, it checks the table. If it finds an entry with the same inputs, it reads the outputs and skips the RPC. Pure functions are common: in the SeBS [229], TrainTicket [78], and FunctionBench [171] benchmark suites, 50.0%, 57.6%, and 60.0% of the functions, respectively, are pure.

7.4.3 MXContainers for Sharing Memory and Processor State

The MXContainer for a function instance has a dispatcher process and multiple handler processes. In the first invocation of the function, the dispatcher first executes the function initialization. In every invocation of the function, including the first one, the dispatcher forks a handler process that executes the function. With this design, the different handlers automatically share the unmodified initialization state (*LibLd* and *Init* in Section 7.2.2) and, on a write, create private page copies via copy-on-write. This is in contrast to previous FaaS schemes, where different invocations of the same function share at most *LibLd*. As shown in Figure 7.4, *Init* is large. When many handlers are running concurrently, sharing *Init* pages rather than replicating them reduces the memory footprint significantly. As a result, the MXContainer approach substantially reduces the total memory footprint of the multiple concurrent invocations relative to previous FaaS schemes. The smaller footprint frees-up space for containers of other functions.

With MXContainers, the startup overhead of the multiple concurrent invocations is reduced, as it is paid only once for the first invocation of the burst. Given the short-lived execution of functions, reducing the startup overhead speeds-up execution significantly. Note that, in an MXContainer, each function invocation is executed in the address space of a new process. No process is reused to execute multiple function invocations. Thus, MXContainers avoid the security and correctness issues of reusing a process for multiple invocations.

Serverless functions do not typically write to files because their updates are not persistent. However, they could read from read-only files or write to temporary files. Hence, if we allow multiple processes to run concurrently inside the container, we need to ensure there are no data races in file updates. To achieve this, we develop a scheme similar to copy-on-write memory pages. As long as a handler process only reads from a file, it can use the shared initial file. However, once the handler tries to perform an update to the file, it creates its own temporary file, with a unique name. From this moment on, all reads and writes by the handler are done on the new temporary file. When the handler completes its execution, all of its temporary files are discarded.

An alternative would be to use existing container primitives such as `namespace` and `chroot`. However, these primitives are inefficient because they require copying all the files before the handler starts execution.

Finally, since all these multiple invocations of the same function run on the cores owned by the MXContainer, their processes reuse the cache and branch predictor state with each other. Individual functions typically have low divergence in the set and order of executed instructions across different invocations (even with different inputs) [197, 305]. Thus, the MXContainer design significantly reduces the misses in caches and branch predictors.

7.5 MXFAAS IMPLEMENTATION

We build MXFaaS in both OpenWhisk [57] and KNative [60], two serverless cloud platforms. In this section, we discuss a few important implementation aspects.

7.5.1 Function Runtime

We implement the MXFaaS runtime with 1.2K lines of Python code. Users can write functions in any language that supports the forking mechanism. The initialization of a function is performed by importing a module (for Python functions) or by loading a shared library (for C/C++/Rust functions). We discuss support for additional languages in Section7.8.

As indicated in Section7.4.1, the dispatcher in an MXContainer intercepts the blocking calls in handlers. A function can employ various library APIs to invoke other functions or to perform I/O. In Python, most communication libraries such as `requests`, `redis`, `minio`, `pymongo`, and `boto3` call APIs from the `recv` family of Python's `socket` module (e.g., `recv`, `recv_from`, `recv_into`) to block. Hence, we overload all these `socket` APIs with wrappers that inform the dispatcher when a handler (identified by its PID) calls a block API. When the dispatcher is notified, it calls the socket API on behalf of the handler (after suspending the handler). Later, the dispatcher receives the response and informs the handler.

To be able to support other languages, the dispatcher needs to intercept blocking calls beyond Python's `socket` module.

We inspect blocking I/O and RPC libraries from different languages and find that they eventually invoke the `recvfrom` system call. MXFaaS uses `LD_PRELOAD` [306] to intercept target system calls in user mode. When a `recvfrom` system call is captured, the wrapper forwards the handler PID and call arguments to the dispatcher. The rest of the algorithm remains unmodified. In over 110 open-source functions analyzed, we did not observe any other blocking calls that are long enough to be exploited. There are some local OS blocking calls, but these operations are too short to be exploited for scheduling within an MXContainer.

We implement copy-on-write for files by intercepting system calls for filesystem operations, such as `open`, `read`, and `write`. While the handler does not update the file, it can read from the original shared file. Once it tries to update the file via the `write` call, we copy the initial file to a temporary file and save the translation from the initial file name to the newly-created file name. All later operations to the file are redirected to the temporary file.

We implement the I/O access combining support for Redis. Specifically, the dispatcher intercepts `get` and `put` requests by handlers. For a `get`, if there is an outstanding `get` for the same key, the dispatcher augments the existing MSHT entry with new information. Then, when the dispatcher receives the response, it forwards it to all handlers that requested it. Otherwise, the dispatcher coalesces multiple requests to different keys issued within T_{merge} into one collective request. It sends one `mget`/`mput` request instead of many `get`/`put`. Other storage services can be supported in similar ways.

7.5.2 Serverless Platform

MXFaaS requires platform modifications to set the number of MXContainers in the system and the number of cores for each MXContainer. Initially, the load balancer picks a node for each MXContainer. In a given node, MXFaaS divides the cores among different MXContainers based on their relative needs. To estimate the core needs of MXContainers, we dynamically measure: (1) the fraction of requests for each type of function and (2) the time that handlers spent buffered in state *Ready* in the HandlerBuffer of each MXContainer. Based on the measurements, MXFaaS sets (and adjusts) the number of containers in the whole platform for each function and the number of cores assigned to containers.

Consider the MXContainer of a function in a node. Let C be the number of cores in the node, R the overall number of function requests per second (RPS) received by the node, and F the RPS for the function supported by the MXContainer. Then, the MXContainer is assigned $max(C \times \frac{F}{R}, 1)$ cores in the node. At the same time, MXFaaS monitors the average amount of Ready time per function invocation in each MXContainer. It checks such time against two thresholds: a low one (*LowReady*) and a high one (*HighReady*). If the average Ready time in an MXContainer is higher than *HighReady*, MXFaaS first tries to get more cores for the MXContainer by stealing local cores from another MXContainer whose average Ready time is less than *LowReady*. If the MXContainer is unable to get the necessary local cores, MXFaaS creates a new MXContainer for the same function in another node.

To deal with transient loads, MXFaaS sets aside a pool of idle cores on every node. When an MXContainer experiences a load spike, the invoker first takes cores from the pool before stealing cores from other containers in the node. When the load for the container drops, it returns cores back to the pool. The dispatcher observes if the load changes quickly and, if so, it can further reduce *LowReady* to prevent a container from returning cores too soon.

We changed the implementations of OpenWhisk's invoker and load balancer [307, 308]. The invoker works with MXContainers in addition to traditional containers: it is informed of the MXContainer load and allocates CPU cores accordingly. The load balancer is informed of any MXContainer overload. The modifications required about 400 lines of Scala code.

For KNative, we modified the autoscaler and activator [309, 310], which play a similar role as the load balancer and invoker in OpenWhisk. The modification is identical to that of OpenWhisk but written in about 300 lines of Go code.

7.5.3 Multitenancy and Security Implications

MXFaaS follows the multitenancy security model of existing serverless platforms [311]: a container belongs to a tenant, and different end-users can issue service requests that can be executed in the same container without special security protections.

Requests executed in different MXContainers do not share any state and run on different cores. In this chapter, we assume that it is safe to collocate multiple MXContainers from different tenants in the same server. Requests executed in the same MXContainer use process-level isolation: they share initialization state but cannot access each others' private data. Moreover, they execute on the same cores. Therefore, it is potentially easier for them to use shared hardware resources such as MSHRs, branch predictors, and caches as side channels. Most of these side channels already exist in current systems. An analysis of the resulting security implications is beyond this work's scope.

7.6 METHODOLOGY

Evaluation Environment. We evaluate MXFaaS on OpenWhisk and KNative in a 15-server cluster. Each server has an AMD EPYC 1-socket 7402P processor with 24 cores (2-way multi-threaded), 128GB DRAM and a 128MB LLC. Each server runs Ubuntu 20.04.2 LTS. In this chapter, we only discuss the results from OpenWhisk. The KNative results are similar because MXFaaS is not specific to the underlying system.

Baseline System. To serve as baseline, we have emulated the state-of-the-art Nightcore [189] on top of OpenWhisk and KNative. Each container can support up to a maximum number of process-based invocations of the same function. The processes are forked when the libraries are loaded—i.e., before the function initialization. Therefore, unlike MXFaaS, processes can only share the *LibLd* state in Figure 7.4. If there are more concurrent invocations than the maximum allowed, the additional requests are buffered and run later when

Table 7.1: Serverless benchmarks used in the evaluation.

Benchmark	Description
Standalone Functions	
LR-serv	ML model serving: Logistic regression
CNN-serv	ML model serving: CNN-based image classification
RNN-serv	ML model serving: RNN-based word generation
ML-tr	ML model training: Logistic regression
VidConv	Video processing: Apply gray-scale effect
ImgRot	Image processing: Rotate image
ImgRes	Image processing: Resize image
CreateOrd	Web service: Write created order to database
PayOrd	Web service: Withdraw money from account
Serverless Applications	
TcktApp	Get all tickets for a given trip (15 functions)
TripInfApp	Get information about the trip (24 functions)
GetLeftApp	Get unsold tickets for a given time frame (5 functions)
CancelApp	Cancel an order (4 functions)

some of the previous invocations complete.

Evaluated Functions and Applications. We use functions from FunctionBench [171], a suite that includes ML training, ML model serving, and image/video processing. We choose FunctionBench because it is widely used in prior serverless research [53, 64, 136, 312, 313, 314]. Since FunctionBench does not include functions from the popular web services, we include two standalone web functions from TrainTicket [302], a large serverless application suite (CreateOrd and PayOrd). Web-service functions are more lightweight than those in FunctionBench (Figure 7.3a). The complete set of functions evaluated is in the upper part of Table 7.1. We also use serverless applications composed of several functions that call each other. We select four representative applications from TrainTicket [302] (lower part of Table 7.1). We use Redis [315] as the storage service for all the evaluated functions. To be conservative, we annotate no function as pure (Section 7.4.2).

Workloads. We evaluate MXFaaS under *low*, *medium*, and *high* load levels, corresponding to an average of 450 requests per second (RPS), 1000 RPS, and 1800 RPS, respectively. We choose these low, medium, and high load levels as they drive the CPU utilization in our MXFaaS environment to $\approx$ 25%, 50%, and 70%, respectively, which is representative [152, 316, 317, 318]. Also, like in prior research on serverless systems [136, 196, 300, 319, 320, 321, 322], we use the Poisson distribution to model request inter-arrival time.

Parameter Setting. We perform sensitivity analyses to determine the values of MXFaaS parameters. For T_{merge}, we set 1ms. We set the SLO of a request to 2$\times$ the execution time of the same request on an unloaded system. This is more conservative than the prior art [319, 320]. When the average response time gets close to 1.5× the unloaded execution time, MXFaaS considers the corresponding MXContainer to be getting overloaded. Thus, we set $HighReady$ to 40% of the function execution time. When the average response time is close to the unloaded execution time, MXFaaS considers the corresponding MXContainer to be underloaded. Thus, we set $LowReady$ to 10% of the function execution time.

An MXContainer only accepts a certain number of concurrent function invocations. Such number depends on the number of cores it owns (N), the average busy (B) and idle (I) time of an invocation, and the $HighReady$ threshold. Specifically, the execution time of a request is I+B. Within the I period, we can squeeze in $\frac{I}{B}$ additional requests. Therefore, the total number of requests executing in N cores is $N \times (1 + \frac{I}{B})$. If we are willing to add $HighReady$ delay to each I+B execution without violating the SLO, the response time becomes I+B+$HighReady$. If the number of requests to get the response time I+B is $N \times (1 + \frac{I}{B})$, then using a simple proportion we can derive that the number of requests to satisfy the response time I+B+$HighReady$ is $N \times (1 + \frac{I}{B}) \times (1 + \frac{HighReady}{I+B})$. Of these, N are running and the rest are queued in the HandlerBuffer. An MXContainer dynamically targets this number of queued requests. The dispatcher informs the invoker about the number of queued requests every 200ms. If the queue goes over this number, the dispatcher requests the invoker to provide extra cores or offload some of the future requests to another server.

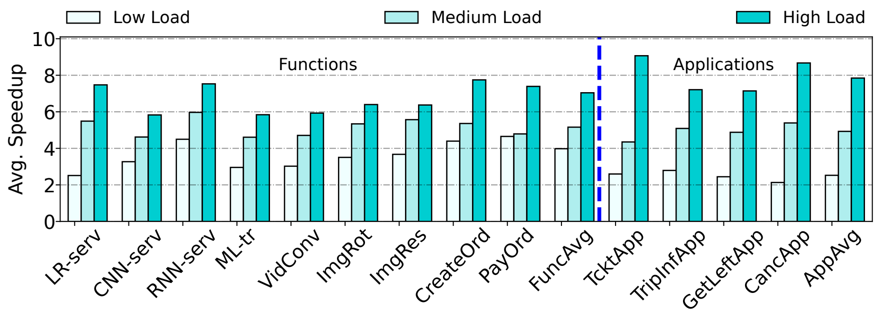
7.7 EVALUATION

In this section, we evaluate MXFaaS' end-to-end latency reduction, its resource efficiency, its scalability, and a comparison to proactive container creation techniques.

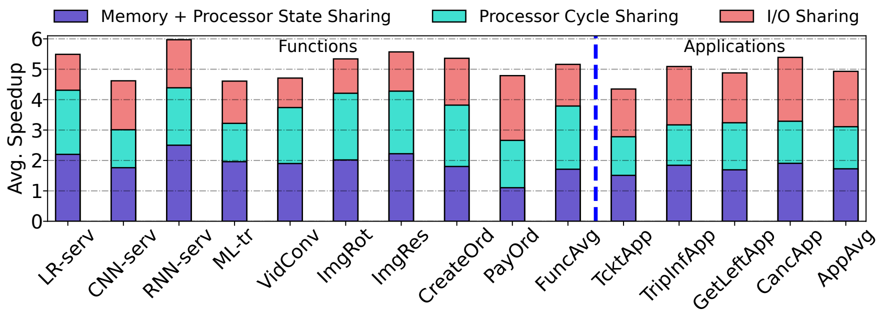
7.7.1 End-to-end Latency Reduction

We measure MXFaaS' ability to reduce the end-to-end latency of requests that invoke serverless functions or applications. The end-to-end latency of a function or application invocation is the time from when the client sends a request until when it receives the result. We normalize the MXFaaS latency to the latency with Nightcore [189], which is our state-of-the-art baseline.

Average Speedups. Figure 7.9a shows the speedups of MXFaaS over the baseline, for low, medium, and high system loads. From left to right, the figure shows bars for the functions, their average, the applications, and their average. On average across all benchmarks and load levels, MXFaaS delivers a speed-up of 5.2×.



(a) Speedups for various loads.



(b) Speedup breakdown aggregated across all three load levels.

Figure 7.9: Speedups of MXFaaS over Nightcore.

MXFaaS achieves higher speedups with higher loads because more requests benefit from the multiplexing. Under high load, the latency of baseline increases substantially, as the baseline does not exploit the idle times per request and, therefore, supports limited parallelism. The result is an inefficient use of processor cycles and queuing of ready requests even though CPUs are idle. This queuing effect is amplified in short-lived functions. For example, the web-service functions (CreateOrd and PayOrd) have a high baseline overhead (as they have the shortest execution) and thus they benefit substantially from MXFaaS.

Figure 7.9b shows the contributions of each of the three MXFaaS components to the average speedup. The numbers are aggregated across all three load levels. We apply the three components one by one: sharing memory and processor state (Section 7.4.3), sharing processor cycles (Section 7.4.1), and sharing I/O (Section 7.4.2). All the techniques are effective. They deliver average speedups of 1.9×, 1.9×, and 1.4×, respectively. Processor cycle sharing especially helps functions with relatively longer idle time due to blocking. I/O sharing has higher contributions in serverless applications, where functions have smaller data volumes and more communication. Finally, memory and processor state sharing especially helps ML functions that share a large model and a large input dataset.

Tail Latency. MXFaaS significantly reduces the tail latency of the function/application requests. Figure 7.10 shows the P99 tail latency in MXFaaS for different loads normalized to that in the baseline. On average across all benchmarks and loads, MXFaaS reduces the P99 tail latency by 7.4×. As the load increases, the reduction also increases. In the baseline, the tail latency is high due to queuing effects when requests are waiting for resources. MXFaaS reduces the tail latency in two ways. First, it only lets the OS schedule *the oldest N ready-to-execute invocations* of a function, where N is the number of cores owned by the function's MXContainer (Section 7.4.1). Second, it uses memory and processor state more efficiently.

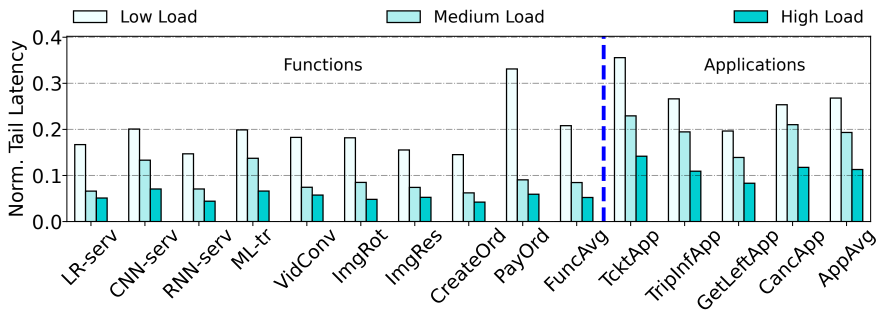


Figure 7.10: Normalized P99 tail latency.

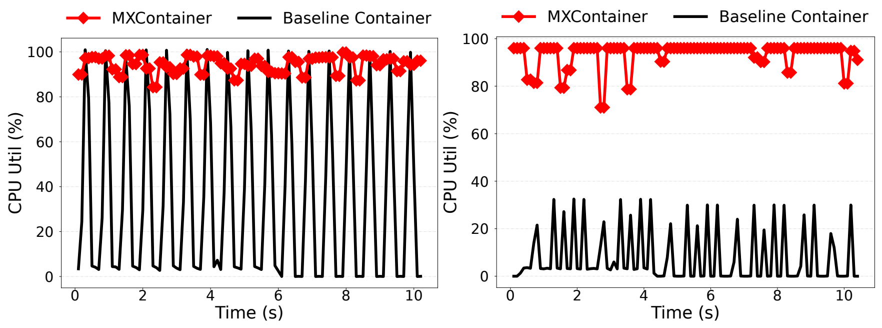
7.7.2 Resource Efficiency

MXFaaS significantly improves resource efficiency, which results in higher throughput. In this section we consider several aspects.

Container CPU Utilization. We compare the CPU utilization in MXContainers and in the baseline containers. Figure 7.11 shows the container CPU utilization over time in MXContainers and in the baseline, while executing CNN-serv (the least idle workload) and CreateOrd (the most idle workload). We see that the CPU utilization of the MXContainer is around 90-100% most of the time, thanks to efficiently multiplexing many function invocations in the container. The CPU utilization of the baseline container is highly fluctuating and often very low.

Since the MXContainer already drives the system to near 100% container CPU utilization, accepting more ready function invocations to compete for cores would only lead to CPU contention. Such contention would degrade performance. To validate this point, we conducted a sensitivity analysis by allowing the OS to schedule more ready-to-run function invocations than the amount of cores owned by the MXContainer (Section 7.4.1). It can be shown that, allowing 20% and 50% more ready requests to contend for scheduling, increases the tail latency by 1.6X and 4X, respectively.

System Throughput The higher CPU and memory efficiency results in improved system throughput. We define System Throughput as the number of concurrent requests that the



(a) CNN-serv (least idle workload) (b) CreateOrd (most idle workload)

Figure 7.11: Container CPU utilization over time.

system can process before the average response time becomes twice that of an unloaded system. We show the value in Table 7.2, based on workload classes. MXFaaS increases the average throughput by 4.8× over the baseline.

Table 7.2: System throughput in MXFaaS and in the baseline.

Workloads	**Baseline** (Req/s)	**MXFaaS** (Req/s)	**Improvement** (Times)
ML-functions	900	4100	4.6
Img/Video Processing	1250	6000	4.8
WebServices	1800	9000	5.0
TrainTicket Apps	200	900	4.5
Average	1037.5	5000.0	4.8

I/O Bandwidth Savings We measure the effect of MXFaaS' I/O sharing technique. Figure 7.12 shows, for Baseline and MXFaaS, a histogram of the latency to fetch the data from global storage. The figure corresponds to the ImgRot function under the high system load. The two designs that we compare include the recently-proposed caching scheme in [191]. From the figure, it can be shown that MXFaaS reduces the median latency from 0.49s to 0.34s. The effect on the tail latency is even more substantial: the latency decreases from 5.81s to 1.76s. The reason is that I/O combining relaxes the pressure on the network and on remote storage. We also see that a large number of MXFaaS requests have very low latency, as a result of hitting in the MSHT (Section 7.4.2).

To pick the value of T_{merge}, we performed a sensitivity study. As we increase T_{merge}, the fraction of merged I/Os also increases. However, the tail latency of the data fetches also increases. We pick a T_{merge} value equal to 1ms, which merges substantial requests without affecting the tail latency much. Figure 7.13 shows the fraction of merged I/Os and the tail latency of data accesses for the ImgRot function and the high load, as we vary T_{merge}. We

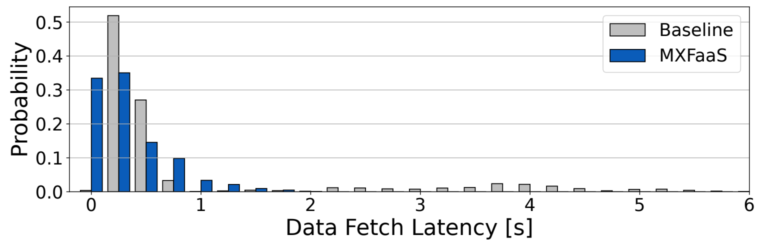


Figure 7.12: Histogram of data fetch latency for the ImgRot function.

see that, for the chosen T_{merge} value, MXFaaS merges 46% of I/Os.

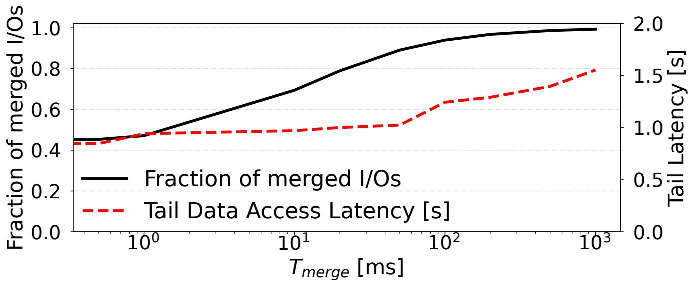


Figure 7.13: Sensitivity study of T_{merge} for the ImgRot function.

The percentage of merged I/Os depends on the data fetch latency and the system load. Across all applications and loads, MXFaaS reduces the number of I/Os by 24%-83%, with an average of 52%. On average, a pending request in the MSHT combines with 6.1 other requests. Further, a request stalling for T_{merge} combines with 3.2 subsequent requests.

Memory/Processor State Reuse The MXContainer design enables substantial sharing of memory and processor state across invocations of the same function. Figure 7.14 shows the average memory footprint in baseline and in MXFaaS across all the three loads. In the figure, the bars are normalized to the footprint in MXFaaS. The numbers on top of the bars are the absolute values of the footprint in Baseline and MXFaaS in GBytes. From the figure, it can be shown that MXFaaS reduces the average memory footprint of the functions and applications by 3.4× (from 67.2GB to 19.5GB). Higher loads lead to higher reductions.

To reason about the branch and cache state reuse, we use the hardware performance counters [323] of the servers to measure the number of misses in the branch predictor and in the caches. We consider two cases: MXFaaS deliberately schedules the requests for the same function on the same set of cores (Case 1), and MXFaaS lets the OS schedule the requests on any currently available core (Case 2). Figure 7.15 shows the measured Misses Per KInstruction (MPKI) in L1 caches, L2 caches, and branch predictor, and the average response time of requests in Case 1 normalized to those in Case 2. The figure shows data for each function. On average, MXFaaS reduces the L1, L2 and branch MPKI by 46%, 43%,

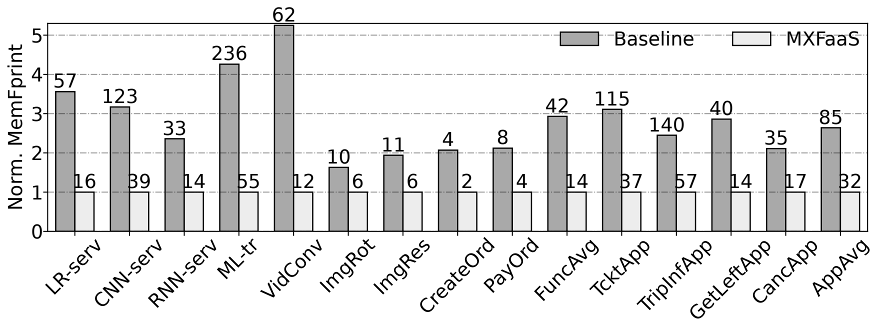


Figure 7.14: Normalized memory footprint in baseline and in MXFaaS averaged across all three loads. The numbers on top of the bars are the absolute values in GB.

and 45%, respectively, which translates into a 30% reduction in the response time.

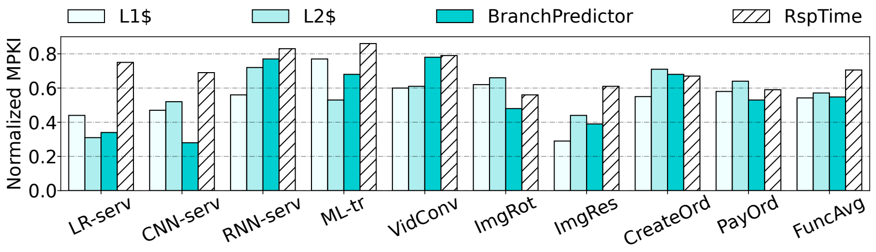


Figure 7.15: Microarchitectural state reuse in MXFaaS and its impact on the response time.

7.7.3 Scalability

MXFaaS is a scalable FaaS platform. We conduct a scalability experiment with three cluster sizes: 10, 15 (default) and 20 servers. Figure 7.16 shows the speedup of MXFaaS over baseline across all benchmarks with medium load for the three cluster sizes. As the cluster size increases, MXFaaS achieves higher relative speedups over the same-size baseline. On average, MXFaaS speeds up the execution by 4.4×, 5.2× and 6.1× with 10, 15 and 20-server clusters.

7.7.4 Comparing to Proactive Container Creation

There are several techniques that reduce FaaS startup-time by predicting which containers will be needed next, and proactively allocating and preparing them [63, 187, 319, 320, 324]. Instead of comparing MXFaaS with each technique individually, we compare MXFaaS with the *best-case scenario*: the prediction technique is 100% correct and there is no cold-start time—if there is enough memory space for the container.

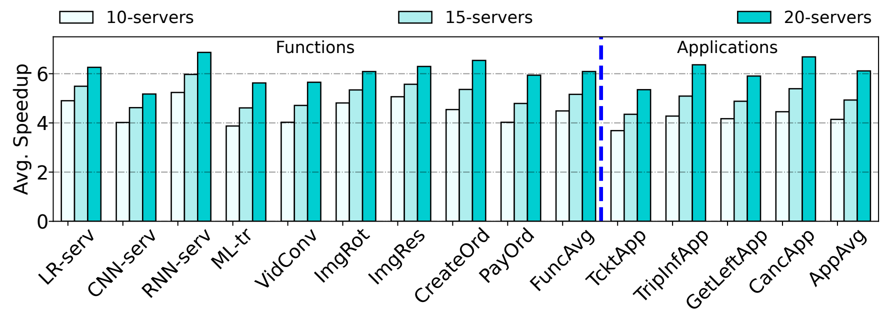


Figure 7.16: Speedup of MXFaaS over the baseline for different cluster sizes.

Figure 7.17 compares MXFaaS with this best-case scenario. It shows the average response time of functions under low, medium and high loads. We show a representative function (CNN-serv) and application (TcktApp). On average across the three loads, MXFaaS reduces the response time over this ideal scheme by 1.6× for CNN-serv and 1.7× for TcktApp. Across all benchmarks and loads, the reduction is 2.1× (or 2.9× if we only consider high load). There are two reasons for the baseline's losses. First, under medium and high loads, available memory becomes scarce. Hence, some requests need to wait to allocate a container till some memory is freed-up. MXFaaS is not as constrained by memory (Section 7.7.2). Second, under any load, MXFaaS benefits from processor cycle and I/O bandwidth sharing.

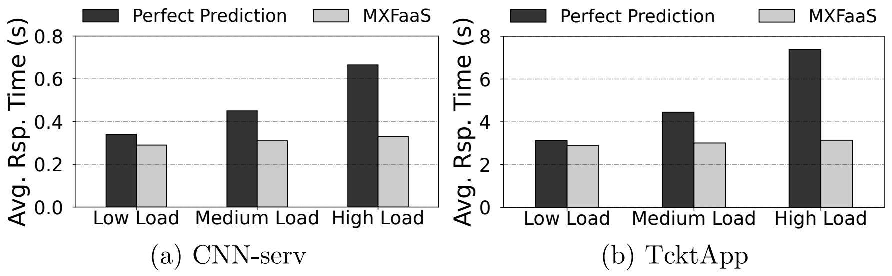


Figure 7.17: Average response time of two benchmarks in MXFaaS and in an ideal environment with perfect pre-warming.

7.8 DISCUSSION

It is straightforward to support the execution of function invocations as threads in an MXContainer instead of processes. We implemented such threading support and measured its performance. Replacing forked processes with threads improves the performance of the benchmarks by 11% on average. However, threads (i) have weaker isolation and (ii) require the function implementations to be thread safe. As indicated by AWS [62], the thread model raises correctness issues in functions that modify global variables.

Languages such as Java and NodeJS do not support the fork semantics. For these languages, the MXContainer dispatcher cannot fork handlers but needs to prepare the initialization of each handler, for example, by loading modules. As future work, we will be working on additional support for these languages. One can choose to use thread-based MXContainers for these languages if not providing strong isolation and thread-safety are not concerns.

7.9 RELATED WORK

7.9.1 Serverless Systems

Work on serverless systems falls into four categories.

Snapshotting. SEUSS [188] and REAP [53] reduce the cold-start overhead by spawning a VM from a previously-generated snapshot. These techniques attain large overhead reductions, although they still have startup times of 70-1000ms. They are not optimized for high concurrency. As the number of concurrent instances increases, the startup overhead becomes larger due to the snapshot-reading contention. They do not exploit the requests' idle time.

Fork from a template. SOCK [301] and Catalyzer's sfork [86] rely on the assumption that containers for different functions have a lot of data in common. Hence, they create a new container to serve a request by forking from a template container shared across all functions. Then, they insert function-specific code in the forked container, execute the function-specific initialization and only then execute the handler. These schemes reduce, but not remove, the cold-start overhead. They hurt performance by executing function-specific initialization code for every concurrent invocation. In addition, they do not share read-only function-specific initialization data, which often has a large footprint. They do not optimize scheduling or manage concurrency. Finally, SOCK requires a special container type, while sfork in Catalyzer requires OS modifications.

Thread/process-level isolation. These schemes use different abstractions for function execution, including container [54, 57, 60], process [193, 300, 301], thread [60, 189], and software-based fault isolation [325, 326]. To reduce cold-start overhead, some schemes relax the isolation boundaries and, in a given container, allow the execution of multiple invocations of the same function (Nightcore [189]) or the execution of the different functions of an application (SAND [300] and Faastlane [193]).

The approach used by SAND and Faastlane does not handle efficiently the common case of a function that is shared among multiple applications. First, the shared function cannot scale independently of other functions in the application. Second, the shared function needs to be copied to all the containers that serve the different applications. Another shortcoming of including all the functions of an application in a container is that the size of the container is very large, as it has to include the support for potentially different runtimes, languages, libraries, and packages. Furthermore, functions can come from different security domains, and executing them together in the same container may leak.

Nightcore executes concurrent invocations of the same function in a container with separate processes or threads. It provides suboptimal concurrency management because (1) a container allows only a predetermined number of requests to be dispatched and (2) the container queues requests above this threshold until prior requests are completed. Nightcore, like the previous two schemes, does not exploit the fact that a function spends significant time idle and thus, wastes available CPU resources. Finally, Nightcore forks processes as soon as the libraries are loaded—hence, processes are unable to share all the read-only function initialization data that is independent of individual invocations of the function.

Predictions. Some startup-time reduction techniques [63, 187, 319, 320, 324] predict which containers will be needed next, and proactively allocate and prepare them. In practice, accurate prediction is hard. Unless the accuracy is high, the response time grows significantly and resources are wasted. Further, when the load is high, available memory becomes scarce. Hence, even with high prediction accuracy, some requests need to wait to obtain memory.

7.9.2 Other Related Work

Microsecond-scale core allocation and scheduling. Recent works have developed μs-scale core allocation and scheduling techniques to improve CPU efficiency [47, 48, 107, 327]. They aggressively reallocate cores to minimize idle time. Unlike MXFaaS, they are agnostic to FaaS workloads and do not consider the synergies between concurrent function invocations as MXContainer does. Many of them require OS changes, while MXFaaS does not.

Optimization of I/O and RPC. Many optimizations have been developed to reduce the overheads of storage I/O and RPC invocations of functions. They include data caching to reduce remote storage accesses [191, 192, 194, 195, 328, 329] and minimizing RPC overhead [189, 190]. MXFaaS' I/O access coalescing is complementary to these efforts, as it reduces I/O bandwidth and communication overhead originating from the containers. Other work [299, 330, 331] reduces the cost of distributing container images under bursty workloads. MXFaaS reduces data volumes as all the invocations of the same function in a given server share the container image.

Startup-time reduction. Some proposals reduce startup latency by proactively preparing function containers to hide latency [63, 187, 319, 320, 324], keeping containers warm [63, 298], or using snapshots and caches [53, 86, 188, 301]. While MXFaaS is complementary to these techniques, it does not benefit as much from them as other systems, since these techniques mostly impact only the first function invocation of the MXContainer.

7.10 CONCLUSION

In this chapter we introduced MXFaaS, a new serverless platform design where concurrently-executing invocations of the same function share processor cycles, I/O bandwidth, and memory/processor state. MXFaaS introduces the new *MXContainer* abstraction, which enables substantial improvements in processor, I/O, and memory efficiency in serverless environments. Our evaluation showed that, with MXFaaS, serverless environments are much more efficient. Compared to a state-of-the-art baseline, MosaicCPU on average sped-up the execution by 5.2×, reduced the P99 tail latency by 7.4×, and improved the throughput by 4.8×. In addition, it reduced the average memory usage by 3.4×.

CHAPTER 8: Software Caches for Low I/O Overheads in Cloud-Native Services

8.1 INTRODUCTION

In serverless systems, for high availability and fast scalability, functions are commonly implemented as *stateless* services [332, 333], which means that all the data of a function is discarded from a node once the function is unloaded from the node. Hence, any durable data must be stored in global storage, such as the Azure Blob Storage service [334]. This results in inefficient data reuse: subsequent function invocations must reload their data from global storage. In addition, for security reasons, cloud providers do not allow direct communication between functions. As a result, data items must be passed through the global storage [192, 335, 336, 337].

We measure that applications spend 35-93% of their end-to-end response time on storage reads/writes. Such operations are typically implemented as Remote Procedure Calls (RPCs). To mitigate these costs, data can be cached locally in the memory of the nodes where functions execute. However, distributed software caches add a new challenge to the FaaS infrastructure: how to keep these caches coherent while avoiding the high cost of frequent inter-node communication. In this chapter, we show that prior proposals [194, 195, 328, 329, 338, 339] address this challenge in sub-optimal ways for FaaS environments.

Most schemes [194, 195, 329, 338] cache a data item in the memory of only a single node, called the data item's *home node*. Function invocations running on nodes that are not the data item's home always access the item from the home. These schemes eliminate any need for coherence, as there is at most one cached copy of the data item. However, they work well only when the function invocation runs on the node that is the home of the data items that the function accesses; otherwise, remote accesses are needed. In practice, multiple function instances running concurrently on different nodes need to access the same data item. Hence, these caching schemes are not effective. We measure that data movement due to remote reads/writes still accounts for up to 82% of the total application response time.

Faa$T [339] allows a data item to be cached in multiple nodes and keeps caches coherent via a software protocol. It uses a *versioning protocol* that associates a version number with each data item. Data items have a home node, which caches the latest data value and version number. When a non-home node reads the data item, it first fetches the item's version number from the home, even if it caches the data item locally. Then, it compares the version number in the home with the locally-cached version number. If the two numbers match, the invocation accesses the data item directly from the local cache. Otherwise, it fetches it from the home. Further, when a non-home node writes the data item, the update is propagated to the home, where it updates both data and version number.

This protocol works well for large data items, where fetching only the small version number is substantially cheaper than fetching the entire data item. Moreover, it is designed to scale to many sharing nodes and frequent updates, as it avoids invalidation messages. However, it is not optimized for FaaS access patterns: we measure that accessing and checking versions in our applications can cost up to 78% of the application response time.

The reason is that the majority of storage accesses in FaaS are *reads to small data items*. Production-level Azure functions [339] reveal that 77% of the storage accesses are reads, and 80% of the data items are no larger than 12KB. These facts make versioning protocols suboptimal, since: (1) the time to fetch the version number is comparable to the time to fetch the data item, and (2) the majority of version comparisons are unnecessary, since there are no writes between reads.

This motivates us to re-visit *invalidation-based* distributed coherence for FaaS. Invalidation-based protocols, though commonly used for hardware cache coherence [340, 341, 342, 343], have been disregarded in distributed software environments [344], mainly because: (1) coherence directories introduce fault tolerance concerns and (2) invalidation messages may scale poorly with increasing numbers of nodes. However, we argue that invalidation-based protocols can be a good match for FaaS. The reasons are: (1) functions are designed to be stateless [332] and are thus more robust to failures, and (2) the total number of nodes sharing the same data item is typically less than a few 10s [35, 63, 136], which limits the coherence traffic due to invalidation operations.

Given these insights, this chapter proposes *Concord*, a novel distributed caching system for FaaS environments. Concord maintains a software data cache per FaaS application and distributes the cache across the multiple nodes where function instances of that application execute. Concord leverages the extensively-studied area of hardware cache coherence and proposes an invalidation-based distributed coherence protocol in software.

Concord tailors the protocol to a distributed environment in three ways. First, it organizes the distributed caches and coherence support on a *per-application* basis. Second, it designs the coherence protocol to be resilient to failures. Third, it minimizes coherence traffic by modifying the scheduling of function invocations to route them to nodes that likely cache the needed data. To make the protocol resilient to failures, Concord employs write-through software caches and a distributed coordination service that monitors nodes' health. In case of node crashes, the coordination service redistributes the data items homed in the crashed node. Overall, Concord achieves high performance while ensuring data safety.

We also use Concord to provide *transparent support* for transactional storage accesses and communication-aware function placement. Specifically, for transactional accesses, Concord relies on its coherence protocol to detect and recover from transaction races—instead of requiring application re-writing [345] or extensive storage logging [346]. It buffers the speculative state in local caches before committing it to global storage, and relies on coherence messages to detect conflicts. Furthermore, for function placement, Concord exploits coherence messages to transparently learn over time which functions frequently communicate with each other. Later, it uses the collected data to intelligently co-locate such functions on the same nodes, reducing network overheads.

We implement Concord in the OpenWhisk [57] serverless platform. We use a 16-node cluster running a diverse set of serverless benchmarks. Compared to state-of-the-art baselines [338, 339], Concord speeds-up execution by 2.4× and improves throughput by 1.7×, while using only 6.2MB of otherwise idle application memory (*i.e.*, 4.8% of the total application memory). Overall, this chapter makes the following contributions:

• We analyze distributed FaaS software cache designs and the design space of coherence protocols for them.

• We introduce the Concord caching system, which includes a high-performance and fault-tolerant directory-based distributed coherence protocol.

• We use Concord to provide transactional storage accesses and communication-aware function placement.

• We evaluate Concord and its features.

8.2 MOTIVATION

8.2.1 Need and Opportunity for Data Cache Designs

Figure 8.1 breaks down the response time of popular FaaS applications into time spent reading/writing data from/to storage and time processing the data. We will describe the applications in Section 8.5. Our experimental results corroborate past studies [194, 338, 339] showing that global storage access time dominates FaaS response time. The graph shows that such time accounts for 35.1-93.0% of the end-to-end function response time, with an average of 63.1%.

A long line of research [194, 328, 338, 339] uses in-memory caching to mitigate these overheads. Most works propose per-node caches shared by all the applications co-located on the same node (Figure 8.2a) [194, 195, 328, 329, 338]. However, real-world data from Azure production [339] shows that strong data re-use is mainly observed among invocations of the same application. For example, 30% of Azure applications access the exact same data

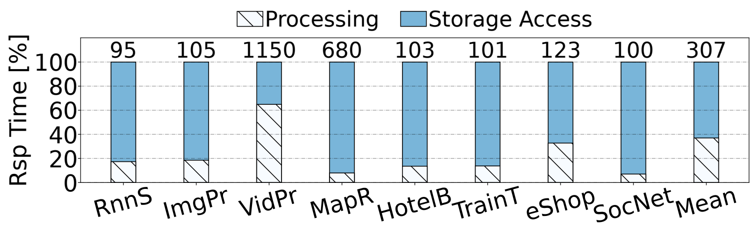
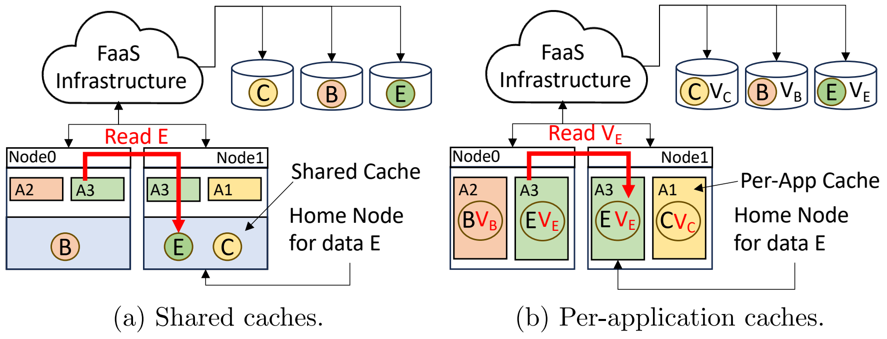


Figure 8.1: Breakdown of applications' response time into processing and storage access. The numbers on top of the bars indicate the absolute response time in ms.



(a) Shared caches. (b) Per-application caches.

Figure 8.2: State-of-the-art in-memory FaaS caching schemes.

objects across all of their invocations, and 99.7% of data objects stored in global storage are not shared across applications at all. Thus, caches should be managed and maintained per application (Figure 8.2b), rather than being shared across applications. This design also enables discarding a cache instance from a node's memory when the corresponding local application instances are shut down.

Using main memory to cache data objects creates the challenge of properly reserving the necessary physical resources and charging for them. Faa$T [339] allocates memory in a node for an application's cache and charges users extra per cached byte access. In reality, we find that it is unnecessary to allocate such extra memory. The reason is that the majority of the memory already allocated for FaaS functions remains unused. Indeed, a recent trace from Huawei [347] suggests that users request 5× more memory than needed for 50% of the functions, which adds-up to 100s of MBs per node. The unused memory of one application can be transparently and dynamically re-purposed into a cache for that same application, improving performance for free.

Insight #1. Accesses to global storage limit the performance of FaaS functions. Per-application data caches can mitigate these costs transparently and, if designed properly, for free—by utilizing applications' allocated but unused memory.

Table 8.1: Average/Max number of node sharers measured when run on a 16-node cluster.

	Low Request Load	**Medium Request Load**	**High Request Load**
HotelB [40]	1.7/6	2.2/9	3.4/12
TrainT [302]	1.3/5	1.6/6	2.2/10
eShop [201]	1.1/4	1.2/5	1.4/6
SocNet [40]	2.7/11	3.6/14	5.0/15
Average	1.7/6.5	2.2/8.5	3.0/10.8

8.2.2 Data Coherence and FaaS Trends

Reads on small data items dominate FaaS accesses. An analysis of Azure public traces [226] performed by Faa$T [339] showed that 80% of data items are no larger than 12KB, 77.3% of accesses are reads, and a large fraction of accesses to data items are bursty (even burstier than Poisson). We observe similar trends with IBM traces[348]. The high fraction of reads and high burstiness can result in high local cache hit rates in distributed software cache designs for FaaS.

Data is shared across nodes. We use Azure production traces for storage accesses [339] to benchmark our FaaS applications running on a 16-node cluster. We will describe the cluster in Section 8.5. We measure the number of nodes accessing the same data, *i.e.*, the data *sharers.* Table 8.1 reports the average number of sharers under a low, medium, and high load of requests across all data items. We find that even under low load, there are multiple sharers per data item. For high performance, distributed software caches should allow caching the same object in multiple nodes. Of course, for correctness, the caches of sharers must be kept coherent.

Insight #2. FaaS distributed software caches require coherence, and the protocol should be optimized for read operations on small data items that commonly hit in local caches.

The maximum number of sharers is often modest. Table 8.1 also reports the maximum number of sharers measured, and shows that such number never reaches the total number of nodes, even under high request load. This is attributed to the modest data sharing across function instances of the same application, and the high degree of function instance co-location in a node. This allows us to revisit protocols for distributed systems that were traditionally discarded because of the potential need to support many sharers.

Functions are robust to failures. On today's serverless platforms, when a function fails, the platform re-executes the function and tolerates the failure [349, 350, 351]. To use this retry-based approach, functions must be idempotent, *i.e.*, functions must exhibit the same behavior when they are re-executed. To make functions idempotent, storage APIs for serverless functions are also designed to be idempotent, typically using a form of key-value interface.

Insight #3. The observed number of sharers per data object and the inherent robustness of serverless functions on failures allow us to revisit coherence protocols originally tailored for hardware directory protocols.

8.2.3 Prior Art on Software Caching Schemes

We detail the operation of two closely related works: Faa$T [339] and OFC [338]. In Section 8.6, we quantitatively compare them to our proposal.

In **OFC** [338], each node provides a cache shared by all locally-running applications. Each data item has a home node and can be cached only in the cache of its home. All caches in the system can be accessed by all applications. Figure 8.2a shows an example of this system. Applications *A2* and *A3* are co-located on Node 0, while *A1* and a second instance of *A3* are co-located on Node 1. Node 1 is the home for data *E*, as determined by the hash of *E*'s address, and all reads/writes to E go to the cache of Node 1. This design has no need for cache coherence because there is no data replication. However, it results in frequent remote reads/writes because a data item can only be cached in its home node.

Faa$T [339] introduces per-application caches and allows multiple cached copies of the same data across nodes. A cached copy consists of the data value and its version number. Each data item has a home node. The cache in the home contains the data value that is consistent with the global storage and the latest version number of the data. When a non-home node reads a data item, it first fetches the item's version number from the home, even if it caches the data locally. Then, it compares the number with the locally-cached version number. If these numbers match, the function accesses the data directly from the local cache. Otherwise, the data is fetched from the home. When a non-home node writes the data item, the write is propagated to the home, where it updates both data and version number, and then to the global storage. The home returns the new version number, and the writing node updates both data and version number locally. Writes by the home node update both version and value on the data item, both locally and in global storage. No invalidation is sent.

Figure 8.2b shows an example of this system with the same application and data layout as in Figure 8.2a. Note that Applications *A1*, *A2*, and *A3* have separate caches. However, *A3* has caches in the two nodes. Each data item has a version number, shown with the letter *V*. In the figure, Node 1 is the home for *E* and *E* is also cached in Node 0. When *A3* running in Node 0 reads *E*, Faa$T first fetches V_E from Node 1 and compares it with the local V_E. If version numbers are the same, *A3* consumes the local data. Otherwise, *A3* fetches *E*'s data and version number from Node 1, and stores them locally.

This design suffers from traffic induced by fetching version numbers. Figure 8.3 compares

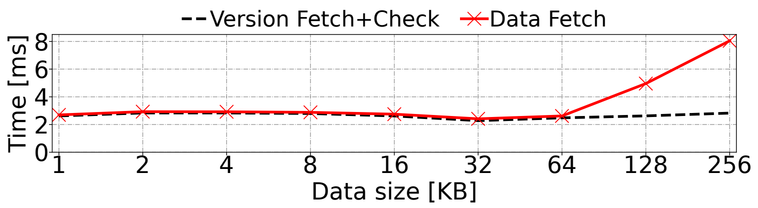


Figure 8.3: Time to fetch and check a version number vs time to fetch a data item.

the time it takes to fetch and check a version number from the home node to the time it takes to fetch the actual data from the home, as we increase the data size in our cluster. We use dual-port Intel X520-DA2 10Gb NICs (PCIe v3.0, 8 lanes) and gRPC for reading/writing remote data. The nodes are connected with 10 Gbps full-duplex ethernet. The total time for fetching the data and its sequence number also includes gRPC overheads, such as serialization and RPC-encoding. We see that the cost of version fetch and check is comparable to the cost of data fetch for objects of 64KB or less; it is lower only for larger objects. In FaaS environments, the data size is typically no larger than 12KB. As a result, Faa$T adds coherence messages that could potentially be avoided.

Insight #4. Prior cache designs are suboptimal for FaaS, as they induce remote accesses to either data or metadata.

8.3 CONCORD: HIGH-PERFORMANCE CACHING FOR FAAS

Driven by our insights, we propose *Concord*, a distributed software caching system for FaaS environments. Concord achieves high performance with support for fault tolerance. It exploits unused memory resources that users are already charged for. Hence, it does not introduce any extra monetary costs. Finally, it requires no changes to the applications.

8.3.1 Concord Overview

Figure 8.4 overviews the Concord system. Concord takes a fraction of the memory allocated to local functions that is temporarily unused, and re-purposes it to act as *local cache instances*. It then binds a cache instance to each application that includes a local function. In a cluster, a function may have multiple function instances located on the same or different nodes. Function instances from the same application that are co-located on a node share a cache instance. For example, in Figure 8.4, instances of *Func1* and *Func2* from *App1* in Node 0 share the same cache instance. A given application typically has function instances in multiple nodes of the cluster, as different invocations of the application may be executing

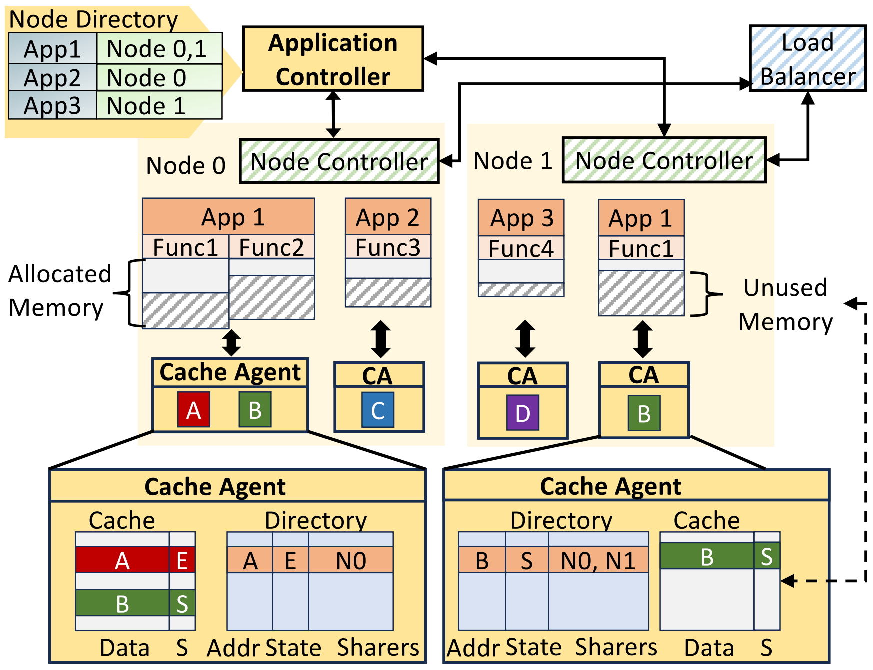


Figure 8.4: Architecture overview of the Concord FaaS platform.

on different nodes. Consequently, multiple nodes may have cache instances of the same application. All these cache instances together form the distributed cache of the application. In the figure, the cache instances of *App1* on Nodes 0 and 1 form *App1*'s cache.

Since multiple applications may be co-located on a node, individual nodes typically host multiple cache instances. However, as data sharing occurs only within an application, caches of different applications are isolated from each other. Cache instances, like function instances, are ephemeral: once all function instances sharing a cache instance are removed from a node, the cache instance is discarded.

Concord allows copies of the same data item to reside in cache instances in multiple nodes, and keeps these copies coherent—*e.g.*, in Figure 8.4 data item *B* is cached in cache instances in Nodes 0 and 1. As Concord binds caches to applications, writes from one application do not affect the caches of other applications.

Concord is tailored to the needs of a distributed FaaS environment. Compared to systems that are kept coherent with conventional hardware schemes, FaaS environments have larger scale, suffer longer-latency communication and higher network contention, and are more prone to failures. Thus, Concord (1) minimizes contention, (2) reduces the number of coherence messages, and, (3) provides fault tolerance.

First, to minimize contention, Concord has per-application caches, and the directory for an application is sharded only across the nodes that contain caches of that application.

Each data item can be cached in multiple nodes, but it is assigned one home node. The home provides the data item to other nodes and, through the directory, maintains the cache instances coherent for that data. The home of a data item is decided via *consistent hashing* [352]. For a given application, when a new cache instance is created or an old one is removed, the home of some data items of the application may dynamically change.

Second, Concord schedules function invocations via a coherence-aware algorithm, which tries to place invocations that operate on the same data on the same node. In this way, such invocations often use the same cache instance, minimizing the need for coherence messages.

Third, Concord designs the protocol for fault tolerance. Caches are write-through and, thus, global storage is always up-to-date. Concord uses a distributed coordination service to detect a failed cache instance and then embeds a recovery mechanism in the protocol.

8.3.2 Concord Architecture

Concord has an organization similar to existing FaaS platforms [57, 353], with enhanced load balancer and node controllers. Concord adds an *Application Controller* and, in each node, a *Cache Agent* (CA) for each cache instance. Figure 8.4 overviews the architecture.

Cache Agent. Each cache instance is managed by a cache agent, which has three roles. First, storage requests issued by a function are transparently intercepted by the function's runtime and forwarded to the corresponding cache agent. The cache agent can either satisfy the request locally, forward it to a remote cache agent, or forward it to the global storage. Second, to maintain cache coherence, the cache agent manages a *Data Directory* for the data items homed locally. The directory entry for a given data item stores the list of remote cache agents that have the data item in their local caches (*sharers*). When a node modifies the data item, the cache agent in the data item's home invalidates all other sharers. Finally, the cache agent re-purposes the allocated unused memory of all the co-located instances of functions that belong to an application into that application's local cache instance. The agent monitors the memory use of the functions and dynamically adjusts the size of the cache instance based on the total unused memory. When the cache instance needs to shrink, the agent evicts some data.

Node Controller. On every node, the node controller connects function instances with the appropriate cache agent. When a function instance is created, the node controller checks if there is a corresponding cache instance. If there is no such instance, the node controller creates it, together with its cache agent.

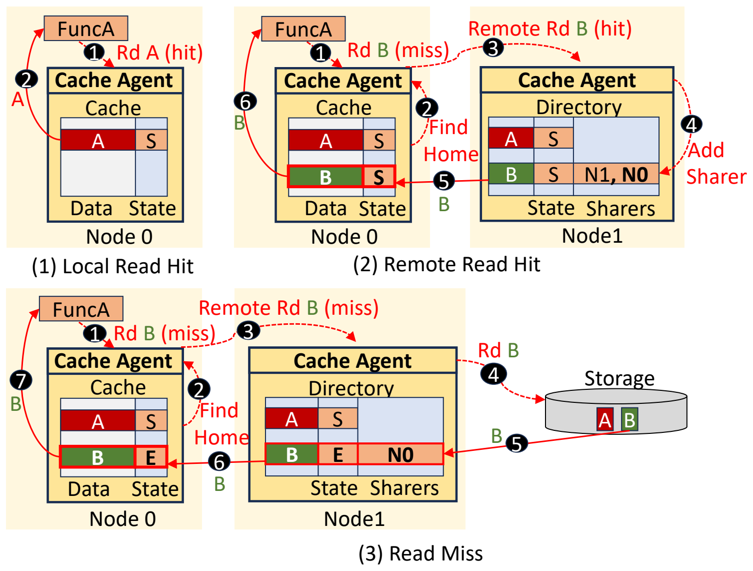
Application Controller. It stores a list of the nodes that host a cache instance of each application in a *Node Directory.* When a new cache instance is created or an old instance is removed, it informs other cache instances of the same application.

8.3.3 Distributed Directory-Based Coherence Protocol

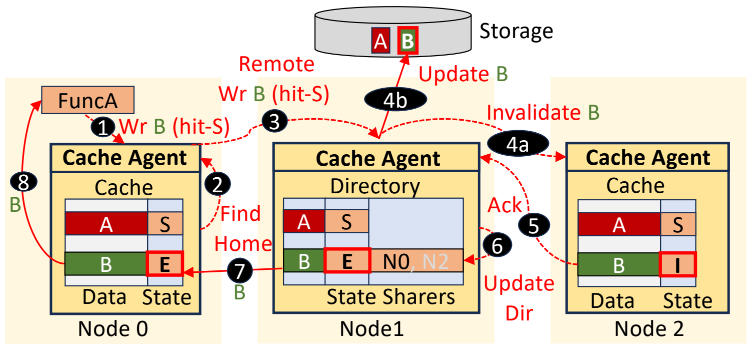
Data Directory. The main software structure for maintaining cache coherence in Concord is the *Data Directory.* The data directory of an application is distributed and managed by the application's cache agents. Each entry in the directory corresponds to a data item, which is a blob of potentially different sizes accessed by its key. A directory entry stores the list of cache instances that currently cache the data item (*i.e.*, the data sharers), and whether the data item in the cache instances is in state Shared (S) or Exclusive (E) (in which case, there is a single sharer). The data item in a cache instance can be in one of three possible states: Exclusive (E), Shared (S) or Invalid (I). E and S states indicate that the data item is cached in a single cache instance or in potentially multiple cache instances, respectively, and that the data is coherent with storage. The cache instance caching the data in state E is the *data owner.* We use a MESI protocol without the M state; the latter is removed to enhance reliability.

The directory of an application is distributed across all the nodes that have cache instances of that application. A given data item has its directory entry in its home node, where its *home cache agent* manages its entry. For example, in Figure 8.4, the homes and directory entries of data items A and B of application $App1$ are in Node 0 and Node 1, respectively.

The home cache agent of a data item is determined via consistent hashing [352]. Consistent hashing is a common technique used in distributed systems to shard the keys uniformly across a cluster of nodes. The goal is to minimize the number of keys that need to be moved when nodes are added or removed from the cluster, thus reducing the impact of these changes on the overall system. In Concord, we use it for when the cache of a give application expands into more nodes or shrinks into fewer nodes. Specifically, in Concord, all the cache agents of an application form a consistent hashing ring. The hash of a cache agent ID determines the position of the cache agent in the ring. The hash of the address of a data item determines the position of the data item in the ring. The home of a given data item is the first cache agent that appears in the ring while traversing the ring clockwise starting from the data item's position. Thus, when cache agents are added to or removed from the ring, some data items may change homes. Note that this is a departure from conventional hardware schemes, where the location of the directory entry for a data item is fixed over time, as long as the number of nodes remains constant.



(a) Read operations.



(b) Local write hit on a data item in S state.

Figure 8.5: Coherence operations in Concord. Dashed lines indicate control messages, while full lines indicate data transfers.

Coherence Operations. There are six coherence operations in Concord's cache coherence protocol. As we describe them, note that, to minimize traffic, when a cache instance evicts a data item, it does not inform the home.

Local read hit. It occurs when a read finds the data item in the local cache instance (Figure 8.5a-1). The local cache instance provides the data item.

Remote read hit. When a read does not find the data item in the local cache instance (Figure 8.5a-2), the cache agent forwards the read to the home cache agent of the data item (1-3). If the home has a directory entry for the data item, this is a remote read hit. The directory entry can be in state S or E. Assume state S. In this case, the home cache agent gets the data item either from the local cache instance (if the home cache instance has it) or from the global storage (otherwise). The directory then marks the requesting cache agent as a sharer (4), and forwards the data item to it (5). The requesting cache instance loads the data item in state S and passes it to the function process (6).

Assume, instead, that the data item is in state E in the directory. The home cache agent gets the data item from the owner cache instance, which downgrades its state to S. Both the requesting cache agent and the previous owner are marked in the directory as sharers in state S. The data item is then forwarded to the requesting cache instance, which loads it in state S. Note that, because of a cache eviction, the owner cache agent may respond that it does not have the data item anymore. In this case, the home cache agent gets the data item from storage, marks the requesting cache agent as sharer in state E and forwards the data item to the requester, which loads it in state E.

Read miss. When a read misses in the local cache and, on reaching the home, finds that the directory has no entry for the data item (Figure 8.5a-3), this is a read miss (1-3). The home cache agent fetches the data item from global storage (4), creates a directory entry for the data item, sets the requesting cache agent as sharer in state E (5), and sends the data item to the requesting cache instance (6), which loads the data item in state E and passes it to the function process (7).

Local write hit. When a write finds the data item in the local cache, this is a local write hit. The data item can be in state E or S. If it is in E state, the write updates the local cache and propagates to global storage, bypassing the home. The local cache agent does not accept external requests for the data item until the storage acknowledges the update.

If the data item is in S state (Figure 8.5b), the write updates the local cache and is propagated to the home cache agent (1-3). The home cache agent checks the corresponding directory entry, sends invalidations to any sharer cache instance (4a), and propagates the update to global storage (4b). All sharer cache instances invalidate their copies and then send acknowledgments to the home (5a). When the home cache agent receives all acknowledgments (including one from the storage), it updates the directory to mark only the requesting cache agent as owner in state E (6). Then, the home responds to the requesting cache agent (7), which marks the state of the local copy as E and informs the function process (8).

Remote write hit. When a write does not find the data item in the local cache instance, the cache agent forwards the update to the home cache agent of the data item. If the home has a directory entry for the data item, this is a remote write hit. Then, the transaction consists of the home cache agent sending invalidations to the current sharers and propagating the update to global storage. The action is slightly different depending on whether the directory entry has an owner in state E or multiple sharers in state S. In the first case, the home cache agent needs to send the invalidation to the owner and receive the acknowledgment before sending the update to global storage; in the second case, the home cache agent can send the invalidations to all the sharers and the update to global storage in parallel. In either case, when the home cache agent has received all the acknowledgments (including the one from the storage), the transaction follows steps 6, 7, and 8 of the local write hit.

Write miss. When a write misses in the local cache and, on forwarding the update to the home, finds that the directory has no entry for the data item, this is a write miss. The home cache agent propagates the update to global storage. On receiving the acknowledgment from the storage, the home cache agent creates a directory entry for the data item, sets the requesting cache agent as owner in state E, and sends an acknowledgment to the requesting cache, which marks the entry in the local cache in state E and informs the function process.

We have seen that, when a node writes to a data item that is in state E in its local cache instance, the update propagates to storage directly while bypassing the home. This design is faster than having to go through the home. It exploits the common case when a node keeps updating the same data item while no other node is accessing the data item. It is also race free because any future read or write to the data item by another node must access the cache instance of the owner node before reading the data item or updating storage, respectively.

In all other writes, the home cache agent is informed of the update and acts as the point of serialization for writes to that data item. If multiple nodes want to update the data item concurrently, the home processes one write operation at a time, and waits until all nodes acknowledge the invalidation and the global storage is updated, before signaling that the write operation is completed. This eliminates any data races.

In theory, sending invalidations can slow down write transactions in Concord. However, in a write-through protocol (used in both Concord and the state-of-the-art [339]), the overall write latency is typically dominated by the update to the global storage and its acknowledgment. Indeed, except in the case when there is a single sharer in state E, storage update and its acknowledgment happen in parallel with sending invalidations and receiving acknowledgments from the sharers—therefore hiding the cache invalidation latency. If the number of sharers is so high that the invalidations to S nodes and their acknowledgments are on the critical path, Concord could potentially fall back to a versioning coherence protocol [339, 354], but we have not evaluated it. We evaluate the cost of invalidations in Section 8.6.

Support for External Reads/Writes. Concord allows other cloud workloads to share data with serverless functions via global storage, through what we call external reads/writes. When users deploy their FaaS applications to Concord, they specify which storage locations are going to be used by the applications (*e.g.*, folders in Azure Blob Storage [334]). Then, the system registers a *listener* FaaS function that is invoked on every storage update (including external writes) on these locations [355]. When the listener is invoked, it first checks if the write was triggered by a FaaS function or from an external application. If the latter, the listener forwards the update to the application controller. The controller forwards the update to the correct home cache agent, which handles the external write as a local write. Thus, even with external writes, functions never operate on stale data.

8.3.4 Dynamic Coherence Domains

We call the set of cache instances of an application a *coherence domain*. In Concord, the coherence domain of an application changes over time. This is because a cache instance is ephemeral: it is created when the first function instance of the application is loaded into the node, and it is destroyed when all the instances of all the functions of the application are removed from that node. A node controller removes a function instance from a node when its grace period expires (*e.g.*, after 10 minutes without being used [63]) or when there is no space in the node. To ensure correctness, all cache instances must know what is their current domain and how to compute the homes of all the data items in their application.

Consistent hashing enables cache instances to enter and leave a domain with minimal disruption [352]. Consider a cache instance leaving a domain. The instance first synchronizes with all the other cache instances in the domain, informing them about its departure. Then, such instances remove from their local directories any sharer pointer pointing to the node of the departing cache instance. In addition, the instances recompute the new home for all the data items that were homed in the departing cache instance. Note that, with consistent hashing, all nodes can compute the new data item homes in a distributed, decentralized manner. Then, the departing cache instance sends the directory entries of all the data items homed locally to the corresponding new home, and waits for an acknowledgment from the new home. Note that all the data items are re-homed into the same node—i.e., the next node clockwise on the hash ring. Finally, after all the data items have been moved, the remaining cache instances synchronize again. By using this two-phase commit protocol, Concord avoids data races. Concord takes similar steps when the domain expands after a new cache instance is created.

8.3.5 Memory Use in Concord

Concord does not reserve extra memory for the software caches. The size of the cache for an application is dynamically adjusted by leveraging the unused memory from all the co-located

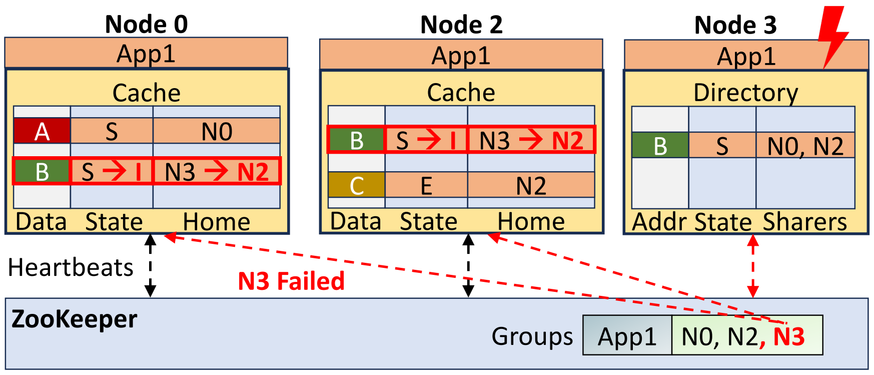


Figure 8.6: Failure detection and recovery in Concord.

containers that belong to the same application. As a result, the cache does not increase the application's memory footprint. Large objects are cached only if sufficient unused memory is available, and they are evicted if their memory is needed for regular application operations. When the cache size reduces dynamically, some data items and directory entries may be evicted. However, no data item changes homes and, thus, there is no rehashing of the data.

If caches were to increase an application's memory footprint, we could run the risk of having to evict containers due to lack of memory space. The result could be more container cold starts, which are more expensive than remote storage accesses.

8.3.6 Fault Tolerant Distributed Coherence Protocol

As core failures happen rarely in multicore processors, hardware coherence protocols do not typically include features for fault tolerance [356]. However, the software-based coherence protocol in a distributed environment must be robust to unexpected node failures. Hence, Concord is equipped with mechanisms to detect failed nodes and recover from them. Additionally, Concord ensures that the data remains consistent on a node failure. We consider each of the two aspects in turn. Note also that serverless functions have inherent resilence to node failures due to their idempotent design principle.

First, to detect node failures, Concord uses a distributed coordination service that manages the membership of the nodes. Figure 8.6 shows the mechanism. The coordination service periodically sends heartbeats to all the cache agents of each of the active applications. When a node does not respond to heartbeats, Concord assumes that the node failed. Then, it informs all the cache instances in the coherence domain(s) of the application(s) in the failed node. We use ZooKeeper [357] in our implementation. As ZooKeeper provides hierarchical namespaces, each application in Concord is treated as a separate group, and ZooKeeper manages the membership of cache instances per individual application. For example, in

Figure 8.6, when Node 3 fails, ZooKeeper informs only the other cache instances of App_1, which happen to be in Nodes 0 and 2. If Nodes 0 and 2 also run other applications, the cache instances of those applications are not informed.

When the cache instances of an application receive a notification that one of the cache instances in their domain was in a node that failed, they first check if they locally cache any data homed in the failed instance. As caches are write-through, it is guaranteed that the data item in the global storage is up-to-date, and the only lost information is the directory. Hence, the cache agents in the domain evict from their caches all the data items homed in the failed cache instance. Then, they form a new consistent hash ring as described in Section 8.3.4. In the example of Figure 8.6, Nodes 0 and 2 detect that locally cached data item B was homed in an instance in failed Node 3. They evict B from their caches. After recomputing the consistent hash ring, the figure assumes that B is now homed in Node 2.

Second, Concord ensures that the data remains consistent on a node failure. Node failures during reads are simple to handle, as reads at most change the directory state. During recovery, if the updated directory was lost, all sharers evict the data items in their caches that were homed in the failed node; if the reader node was lost, the directory removes the reader node from the list of sharer nodes for all the data items.

Writes are more complex, as they additionally modify the state of data items. The critical case is when the home node fails while processing a write that could have updated global storage but failed to invalidate all the cached copies. In this case, Concord must prevent the case of some cache instances reading the new value of the data item while others can still read the old value of the data item. This is prevented as follows: no cache instance is allowed to read the global storage for a data item that was homed in the failed node (and therefore potentially read a new value) until the recovery is complete. During the recovery, as nodes discover the failed node, they evict from their caches the data items homed in the failed node. By the time the recovery is over, all the old versions of the data item in cache instances are explicitly invalidated and a new home is declared. The next read will necessarily go through the new home, which will read the latest data item value from storage. All subsequent reads will see the new value while no read will see the old value. No data inconsistency will occur.

8.3.7 Coherence-Aware Invocation Scheduling

In conventional systems, function invocations are typically scheduled on a node that has a warm container of the function to minimize cold start overheads [224]. In high-load environments, a given function may have concurrent instances on several nodes. In this case, existing systems schedule an invocation randomly on any of the nodes that have an instance of the function. These invocations may operate on the same or different data, typically determined by the invocation's inputs, which *are visible* to the provider. With random scheduling, invocations operating on the same data may be scheduled on different nodes. This results in frequent remote accesses, increased number of cache instances sharing data and, consequently, a larger number of coherence invalidations.

Concord proposes *coherence-aware* invocation scheduling. When the load balancer receives a function invocation, it picks the function instance to send it to as follows. It computes the hash of the invocation inputs and uses the result to pick one of the nodes that has an instance of the function. By doing this, the system maximizes local cache hits.

It is possible that the chosen node is overloaded. In this case, the load balancer does not send the invocation to it. Instead, it tries with another hash function and picks the resulting node. If multiple tries picked overloaded nodes, the load balancer picks a random non-overloaded node. Overall, Concord densely packs the invocations of a function operating on the same data, to minimize data transfer overheads and coherence traffic.

8.3.8 Verification of the Software-Based Concord Protocol

During fault-free operation, the software-based Concord protocol follows the well-established ESI protocol from hardware schemes. The corner cases occur on node failure, system recovery, and coherence domain expansion/reduction. The actions taken in these cases are described in Sections 8.3.4 and 8.3.6. To verify all cases, we use the TLA+ formal specification and verification language, and model-check the protocol in TLC [358]. With TLA+, we first specify all the states and possible actions from all the states. Specifically, we use Exclusive, Shared, and Invalid for the cache states, Active and Failed for the node states, Sharers+Ownership for the directory states, and ActiveInstances for the set of nodes that participate in a coherence domain. We model the events {Local/Remote}{Read/Write}Hit, {Read/Write}Miss, DataEvict, NodeFail, RecoverOnFail, and DomainChange.

We check for no deadlock and no livelock concurrency conditions, and prove that they always hold. The checks are performed under fault-free operation and under node failure. We next describe two corner cases and show how Concord ensures forward progress.

First, consider a Node A waiting for an invalidation acknowledgment from a failed or unreachable Node B. In Concord, Node A will not wait forever. If Node B has failed, Concord's coordination service (*i.e.*, ZooKeeper) will detect it through its heartbeats. If, instead, Node B is simply unreachable from A, Node A will timeout and inform the Application Controller to delete the cache instance on Node B. In both cases, the coordination service then notifies all the nodes in the same coherence domain(s) that the cache instances in B left the domain(s). Then, Node A cancels the waiting request and all the nodes perform the actions described in Section 8.3.6

Second, consider that a read from Node A misses in the local cache, is forwarded to the home Node B and, in the meantime, the cache instance in Node B leaves the coherence domain. In this case, Node B cannot respond to the read request. In Concord, Node A does not wait forever. Node B initiates the creation of a new consistent hash ring as explained in Section 8.3.4. In the process, all the nodes check if they have any outstanding operations with Node B. If a node, such as A does, it cancels the read, recomputes the correct new home, and reissues the operation. All these operations are performed in software.

We also check two data consistency invariants. The first one is that the coherence states in all the caches are correct. The second one is that a read to a valid cache location returns the value last written to it.

8.4 UNLOCKING NEW CAPABILITIES WITH CONCORD

We further leverage Concord's coherence protocol to unlock two new capabilities in FaaS environments: transactions and communication-aware function placement.

8.4.1 Support for Transactions

Transactions could be useful in many FaaS applications, such as banking and online shopping. However, current FaaS systems do not inherently support transactions. To execute sections of code atomically, users need to write the code in a manner that guarantees atomicity. For example, with AWS Saga patterns [345], users write additional functions to detect transaction violations and roll back to the correct state. State-of-the-art proposals for transactions log all the storage accesses that happen within a transaction to enable safe rollbacks (*e.g.*, Beldi [359]) or use global storage locks. Both practices can penalize performance due to the logging overheads and the reduced storage availability due to the locks.

The Concord coherence protocol can automatically and user-transparently ensure transaction atomicity, improving programmability while delivering high performance. The user only needs to specify the beginning and end of the transaction. A transaction can be a piece of a function, a whole function, or multiple functions of an application. Then, the Concord caching layer monitors data accesses and determines if accesses from any function conflict with the accesses of the transaction.

In Concord, while a process P_1 is executing a transaction, every data item that it reads or writes is recorded in the local cache instance as Speculatively Read or Speculatively

Written, respectively, and marked with the ID of the process. No other process P_2 executing the same application, either locally (and, therefore, accessing the same local cache instance) or remotely (and, therefore, accessing a remote cache instance that is kept coherent with the Concord protocol) is allowed to conflict with P_1. Specifically, if P_2 attempts to write a data item that has been speculatively read by P_1, or attempts to read or write a data item that has been speculatively written by P_1, the transaction in P_1 is squashed. Conflicts between two local processes are trivially detected on access to the local cache instance; conflicts between a local and a remote process are detected through the Concord cache coherence protocol: a locally-cached speculatively read data item receives an external invalidation or a locally-cached speculative written data item receives an external read or an invalidation.

Speculatively written data items remain buffered in the local cache instance and are not propagated to global storage. If a transaction is squashed, all its speculatively written data items in the local cache instance are discarded, and their Speculative bits and process ID field are cleared. The transaction can then be re-executed.

If a transaction completes, it proceeds to commit. For this, the runtime grabs a global lock to ensure that commits are serialized. In addition, it locks the directory entries for the data items accessed in the transaction. Then, the runtime forwards all the updates of the transaction to the global storage, and clears the corresponding Speculative bits and process ID fields. After this, the directory entries are unlocked and the global lock is released.

To ensure livelock freedom, forward progress, and fairness, Concord uses known techniques from software transactional memory [360]. They include exponential back-off on conflict and priority increases after multiple squashes.

Users must ensure that a transaction execution has no side effects beyond storage accesses and invocations of functions, such as HTTPs or other system calls. One could automatically detect these side effects and prevent them from being globally visible while the transaction is in progress, using techniques similar to those proposed in SpecFaaS [37]. We leave this exploration for future work.

8.4.2 Communication-Aware Function Placement

Conventional FaaS systems place different functions on nodes in the cluster independently from each other. This means that two functions that interact in a producer-consumer manner may well be placed on different nodes. In this case, performance suffers due to communication overheads. A higher-performance solution would co-locate both functions on the same node, to reduce network overhead and even allow them to communicate via shared memory instead of via network RPCs [335, 336, 337]. Unfortunately, providers cannot easily co-locate producer-consumer functions since, for privacy reasons, they do not know which opaque-box functions communicate with each other.

Concord proposes *Communication-Aware Function Placement* to reduce communication overhead while still maintaining function privacy. The idea is to monitor coherence messages and use this coherence traffic to transparently identify functions that frequently interact with each other. These functions are then co-located in the same node. For example, if one function keeps writing to certain locations and a second one keeps reading these locations, we have identified a producer-consumer pattern.

Concord monitors coherence messages and builds a *Producer-Consumer Table (PCT)*, which lists small sets of functions that frequently communicate with each other. We call them *Paired* functions. Concord consults the PCT to decide where to place function instances. Specifically, when a cluster receives a new invocation of function F, Concord checks if there is already an available instance of F to serve it. If so, the instance is re-used, avoiding a cold start. Otherwise, Concord uses the PCT to place the new instance of F in the cluster. For this, it first checks if the cluster has an instance of a function paired with F. If so, Concord places the new instance of F on the same node to reduce communication overhead. If no such instance exists, Concord anticipates the resource needs of a Paired function and places the new instance of F on a node that can accommodate it plus a Paired function instance.

Overall, Concord uses both communication-aware function placement and coherence-aware invocation scheduling (Section 8.3.7) to minimize communication overheads.

8.5 METHODOLOGY

We evaluate Concord on OpenWhisk [57] in a 16-node cluster. Each node is an Intel Xeon Silver server with 20 cores, 192GB DRAM, and a 128MB LLC. It runs Ubuntu 20.04.

Evaluated Systems. We compare Concord to the state-of-the-art *Faa$T* [339] and *OFC* [338] designs (Section 8.2). For all systems, we use write-through caches and an LRU replacement policy. We use an optimized Faa$T implementation that caches the version numbers of data items in the home [339], rather than having to fetch them from storage. All systems run on an optimized OpenWhisk implementation that supports our MXFaaS [35] framework. Hence, the cold start and scheduling overheads are minimized.

Evaluated Applications. We evaluate Concord with the 7 multi-function applications shown in Table 8.2. Each function is deployed with the minimum amount of memory allowed with OpenWhisk (128MB), and all functions use less than this amount of memory throughout their whole execution. We use Azure Blob Storage [334] as the storage service for all the

Table 8.2: Serverless applications used in the evaluation.

Application	**Description**
TrainT [302]	Book, cancel, or get remaining train tickets.
eShop [201]	Web e-commerce to browse and buy items.
ImgProc [228]	An image thumbnail generator pipeline.
VidProc [338]	Distributed video processing benchmark.
HotelBook [40]	A hotel reservation application.
MediaServ [40]	Review, rate, rent, and stream movies.
SocNet [40]	Social network application.

functions. The distribution of storage accesses is 80% reads and 20% writes with 5% read-only objects, which is the same as in Azure [339].

We evaluate Concord under low, medium, and high load levels, corresponding to an average of 500 requests per second (RPS), 1250 RPS, and 2000 RPS, respectively, received by the cluster. These load levels are chosen based on load testing: the low, medium, and high loads drive the CPU utilization of the cluster to about 25%, 50%, and 70%, which is representative [152, 316, 317, 318]. We use the Poisson distribution to model the request inter-arrival time [38, 136, 196, 300, 319, 320, 321, 322].

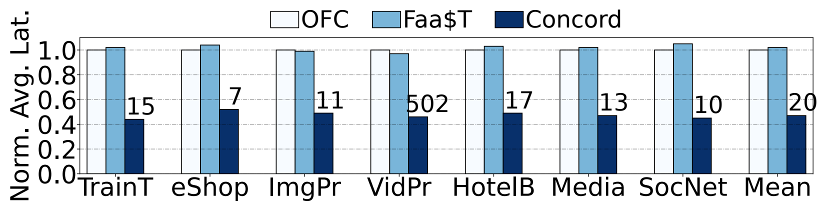
8.6 EVALUATION

In this section, we evaluate Concord's performance, scalability, memory consumption, robustness to coherence domain changes, sensitivity to available cache size, transactional support, and communication-aware function placement.

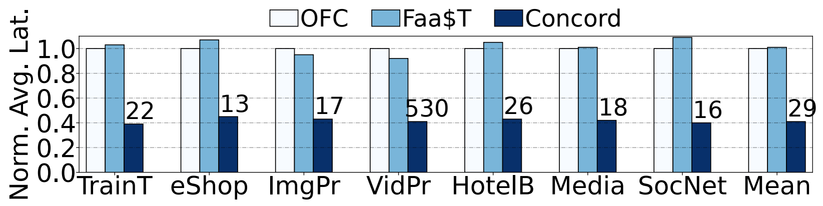
8.6.1 Concord Performance

We measure the applications' average request latency and throughput. The latency is measured end-to-end, from when the client sends a request until the result is received.

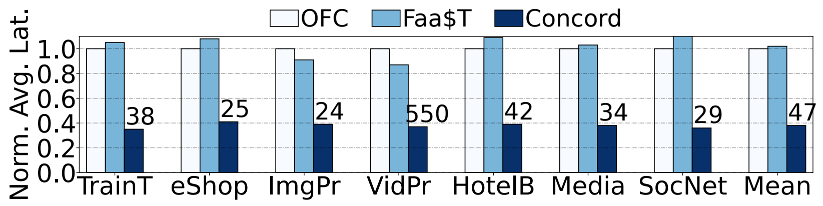
Application Latency. Figure 8.7 shows the average application request latency in OFC, Faa$T, and Concord with different system loads normalized to OFC. On top of the Concord bars, we show the absolute latencies in ms. We see that OFC and Faa$T have similar latencies. While Faa$T allows a data item to be cached in multiple caches, it does not reduce the average latency over OFC because it has significant coherence overheads. On the other hand, Concord's optimized caching protocol minimizes network overheads and results in a much lower latency for all applications and across all loads. On average, Concord reduces the average application latency over OFC by 2.1×, 2.4×, and 2.6× for low, medium,



(a) Low request load.



(b) Medium request load.



(c) High request load.

Figure 8.7: Average application request latency in OFC, Faa$T, and Concord normalized to OFC. The numbers on top of the Concord bars are the absolute request latencies in ms.

and high load, respectively, and over Faa$T by 2.2×, 2.5×, and 2.7× for the same loads. Concord attains higher reductions for applications that frequently read small data, such as TrainT and SocNet; in these cases, the impact of the Concord techniques is more notable. Also, Concord's latency reductions increase with higher system loads.

Cluster Throughput. Concord's reduced request latencies result in improved overall cluster throughput. We define cluster throughput as the rate of requests that the cluster can process before violating the applications' Service Level Objectives (SLO). Similar to prior art [283, 284, 361], we define SLO as five times the application latency on an unloaded cluster. Figure 8.8 shows the cluster throughput of OFC, Faa$T, and Concord. On average, Concord improves the throughput over OFC and Faa$T by 1.7× and 1.8×, respectively.

Characterization of Concord Operations. To understand the performance of Concord, we now characterize some of its operations. Unless otherwise indicated, the data corresponds to the average of low, medium, and high load conditions. First, we note that Concord enables fast reads. Recall that a read request in Concord can result in a local hit, a remote hit, or a remote miss. It can be shown that, on average, a local hit, a remote hit, and a remote miss

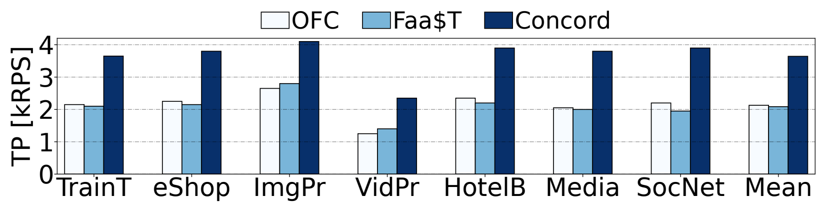


Figure 8.8: Cluster throughput of OFC, Faa$T, and Concord measured in kilo requests per second (kRPS).

Table 8.3: Distribution of read operations in Concord without coherence-aware invocation scheduling (*C-NoCAS*) and in Concord (*C*).

	Local Hit [%] (C-NoCAS — C)	**Remote Hit** [%] (C-NoCAS — C)	**Remote Miss** [%] (C-NoCAS — C)
TrainT	72 — 84	21 — 9	7 — 7
eShop	68 — 75	23 — 16	9 — 9
ImgProc	76 — 85	18 — 9	6 — 6
VidProc	73 — 81	19 — 11	8 — 8
HotelBook	82 — 90	13 — 5	5 — 5
MediaServ	79 — 88	15 — 6	6 — 6
SocNet	73 — 81	21 — 13	6 — 6
Average	75 — 83	18 — 10	7 — 7

for a read in Concord take 1.6ms, 3.1ms, and 32ms, respectively.

Table 8.3 shows the distribution of read accesses per application for (i) Concord without coherence-aware invocation scheduling (Section 8.3.7) and (ii) the complete Concord. In both cases, the techniques of Section 8.4 are not included. We can see that, in Concord, on average 83% of read requests are local hits. Even without the proposed coherence-aware invocation scheduling, on average 75% of read requests are local hits. With so many accesses satisfied with low latency, Concord delivers high performance.

Write requests are less frequent, and can be slightly slower due to the invalidation messages sent to sharers. Figure 8.9 shows the average and maximum number of invalidations sent per write operation throughout the execution of the evaluated applications. Averaged across all applications, an average write operation causes 1.2 invalidations, while the maximum number of invalidations per write is 4.9. Recall that our platform has 16 nodes.

Finally, Figure 8.10 compares the average request latency for Concord without coherence-aware invocation scheduling (Concord No CAS) and Concord for the different applications. The bars are normalized to Concord No CAS and the Concord bars are annotated with the average request latency. Concord No CAS tries to co-locate invocations of the same function on the same subset of nodes. Thus, it already captures some locality. However, it does not consider the specific data that these invocations operate on. Co-locating invocations that

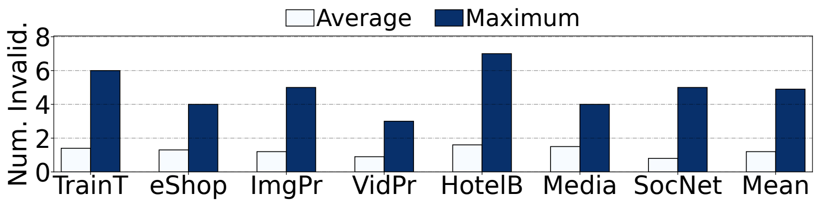


Figure 8.9: Average/max number of invalidation messages sent per write in Concord.

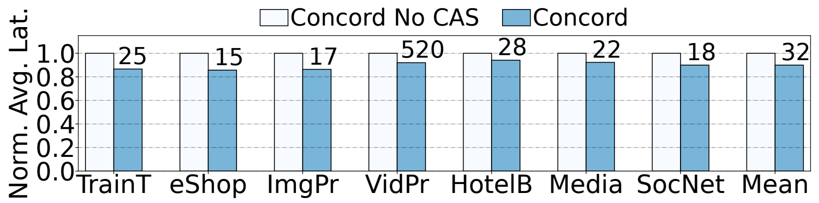


Figure 8.10: Normalized average request latency in Concord without coherence-aware invocation scheduling (Concord No CAS) and in Concord. The numbers on top of the Concord bars are the absolute latencies in ms.

potentially operate on the same data increases data reuse, which results in higher local cache hits. As we see in the figure, coherence-aware invocation scheduling reduces the average request latency by 11%.

8.6.2 Write Operation Scalability

In this experiment, we measure the latency of write operations to shared data in Concord as we change the cluster size from 1 to 30 nodes. All nodes first load the data item in their caches and then one of them writes, invalidating all the other nodes. The home node sends the invalidations and receives the acknowledgments in parallel with propagating the write to global storage and receiving the acknowledgment from global storage. As a reference, a round trip to storage takes around 30ms, while the round trip of an invalidation to another node and its acknowledgment takes around 2ms.

Figure 8.11 shows the average latency of these writes across all the evaluated applications for Concord and Faa\$T. Recall that a write in Faa\$T sends the update to the home and then to the global storage, but does not invalidate the sharers. For comparison, the figure also shows the latency of a read hit to the local cache in both Concord and Faa\$T.

The figure shows that, with few nodes, a write in both Concord and Faa\$T has the same latency of 30ms—which is determined by the access to the global storage. As the number of nodes increases, a write in Faa\$T maintains the same latency, since no node is invalidated. In Concord, the latency increases, reaching 32.4ms for 30 nodes. The reason is that more

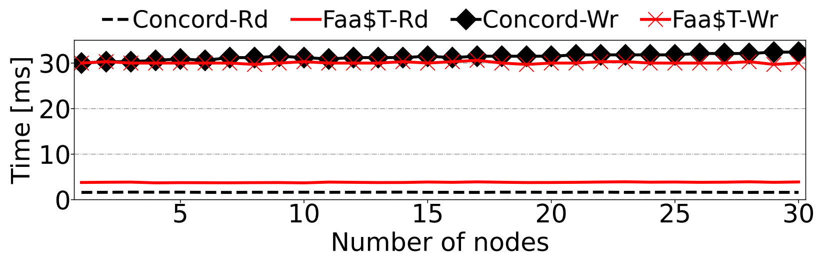


Figure 8.11: Average time to perform a write operation on a data item shared by all nodes and a read operation that hits in the local cache for different numbers of nodes in *Faa$T* and *Concord*.

invalidations are being sent. However, the latency increase is modest because invalidations are sent in parallel with the access to global storage.

The figure also shows the latency of read hits, which does not change with the number of nodes. It is 3.8ms in Faa$T and 1.6ms in Concord. The latency is Faa$T is higher because even a local read hit needs to perform an access to the home to check the version number. Overall, Concord speeds-up the frequent read operations by more than 2× while slowing down the less frequent write operations by at most 8%.

8.6.3 Concord Memory Consumption

Concord re-purposes the unused memory pre-allocated by the functions of an application into the application cache. Hence, users are not charged extra for the caches. However, the amount of cache memory is limited by the amount of unused memory. Fortunately, the unused memory is typically significantly larger than the amount of memory needed by the caches in Concord. Figure 8.12 shows the average and maximum amount of memory consumed by a single cache instance. Across all applications, the average and maximum sizes of a cache instance are 6.2MB and 12.6MB, respectively. On the other hand, the average unused memory per application in a node is 56.8MB. Hence, a cache instance typically uses a bit more than one tenth of the unused application memory.

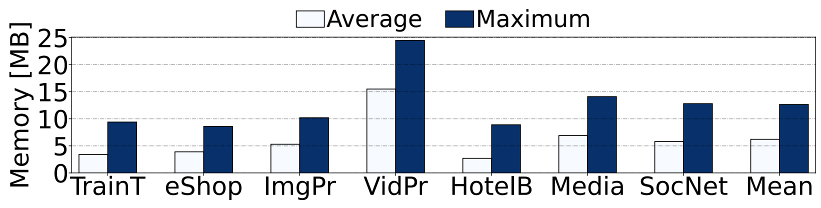


Figure 8.12: Average and maximum memory consumed by a cache instance of an application throughout the application's execution in Concord.

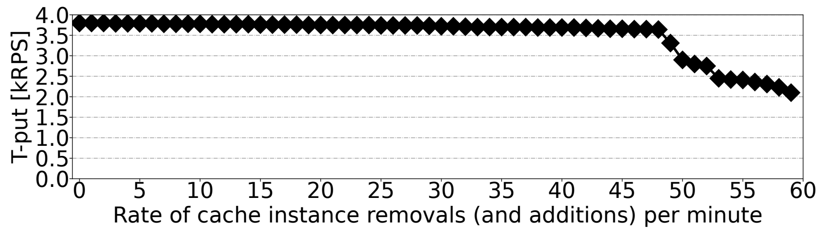


Figure 8.13: Throughput of the SocNet application when varying the rate of cache instance removals (and additions).

8.6.4 Concord Coherence Domain Changes

Concord removes and adds cache instances to a coherence domain transparently to the application. While such operations take some time, they are not blocking. For example, on average, the latency of removing a cache instance from a 16-node coherence domain and adding a new one is about 120ms, but all the nodes beyond the one being removed or added do not stall unless they try to access a data item that moves homes. Figure 8.13 considers the SocNet application and shows the cluster throughput as we change the rate of cache instance removal (and subsequent addition). We use a coherence domain that extends across 16 nodes and randomly select instances to evict and add them back. The figure shows that Concord maintains high throughput until a removal rate as high as 48 removals (and additions) per minute. The other applications show similar performance trends.

8.6.5 Sensitivity Analysis

Available Node Cache Size. Figure 8.14 shows the speedup of Concord over OFC with different node cache sizes at medium load. We define speedup as reduction in average latency, and show the results averaged across all applications. With very small cache sizes (tens of KBs), Concord provides little benefit due to frequent cache evictions. As the cache size increases, the speedup increases, reaching 2.5. Once the cache captures the application's working set at about 6-7MB, further increases in size provide little benefit.

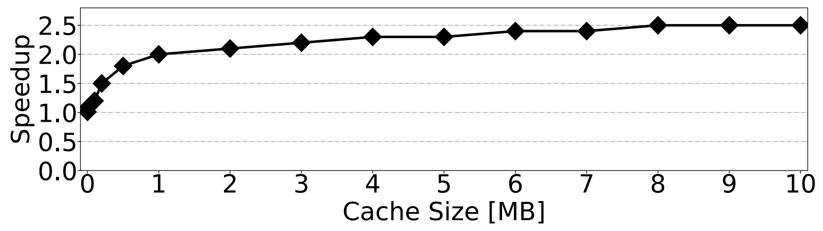


Figure 8.14: Speedup of Concord over OFC with different cache sizes at medium load, averaged across all applications.

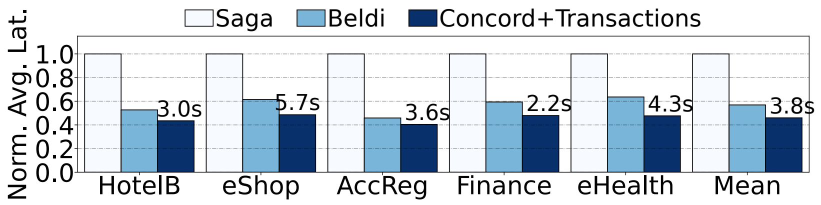


Figure 8.15: Average application latency in Saga, Beldi, and Concord with transactions. The numbers on top of the bars are the absolute latency.

Remote Node Access Latency. The cluster used in our experiments has whole-stack internode round trip latencies of around 2ms. This is a typical latency in current datacenters. If this round-trip latency is reduced to a few μs, the gains of Concord over OFC or Faa$T decrease—since the additional access to the home node required for most read operations in OFC and Faa$T becomes cheaper. Such scenario could be the case with rack-level CXL deployments.

8.6.6 Transaction Support

To evaluate transactions, we use five applications from AWS samples [227]: Hotel Booking, Online Shopping, Account Registration, Online Banking, and Online Health Records. These applications have large transactions: a transaction encloses a sequence of 6-8 functions plus the scheduling and FaaS platform overheads. Figure 8.15 shows the average application latency when using transactions implemented with Saga [345], Beldi [346], and Concord.

Concord outperforms the other schemes for two main reasons. First, it detects transaction conflicts much faster, thanks to using coherence messages; the other schemes detect conflicts by re-reading the data (or logs) from the storage. Second, Concord does not require the execution of additional functions to clean-up an aborted state; it rolls back to the correct state by just flushing its software caches. Overall, Concord with transactions reduces the average application latency by 54% and 20% over Saga and Beldi, respectively.

8.6.7 Communication-Aware Function Placement

To evaluate communication-aware function placement, we cannot use the applications from Table 8.2 because they do not have frequent producer-consumer patterns. Instead, we use a new set of applications with such patterns from open-source projects [171, 192]. These applications are: IoT Sensor Data Collection, ML Sentiment Analysis, Video Processing, Map Reduce, Event Streaming, and Illegal Recognizer. Figure 8.16 shows the normalized average

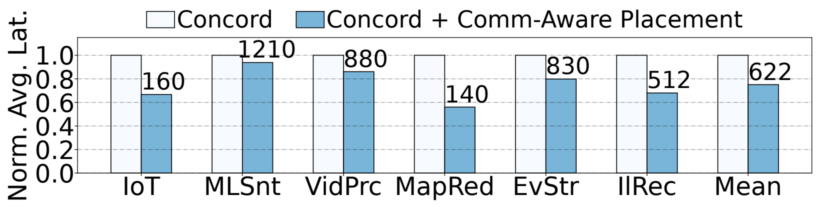


Figure 8.16: Normalized average application latency with Concord and with Concord plus our communication-aware function placement policy. The numbers on top of the bars are the absolute latencies in ms.

latency of these applications with Concord and with Concord plus our communication-aware function placement policy. The numbers on top of the bars are the absolute latencies in ms. We see that co-locating the functions that communicate frequently with each other can significantly reduce application latency. On average, communication-aware function placement reduces the applications' average latency by 25%. The reductions are higher for the applications with shorter execution times, as the network overheads dominate.

8.7 COMPARISON TO A FAULT-TOLERANT PROTOCOL

Apta [362] is a hardware-based cache coherence protocol targeting CXL that uses write-through hardware caches for fault tolerance. In this section, we create a software version of Apta's protocol using software caches, run it in our cluster of Section 8.5, and compare its performance to Concord. Apta uses separate compute and memory nodes, and places the directory in the memory nodes. Apta is described without explicitly considering persistent storage (unlike Concord).

Apta introduces *lazy invalidations*, where the invalidations issued by the directory on a write are moved out of the critical path of the write—allowing the write to complete while some caches may hold stale data items for a short window of time. Apta then enforces *coherence-aware scheduling*, where the memory nodes tell the scheduler not to schedule functions that might use such data items, on the nodes that temporarily have stale values of such data items.

Some additional differences between the extended implementation of Concord and Apta are that Concord introduces: 1) *coherence-aware invocation scheduling* (where a hash of the function inputs determines the node where to schedule the function, hoping to reuse state left by the same function with the same inputs), 2) *transaction execution* (easy to support because all accesses in Concord are recorded in software), and 3) *communication-aware function placement* (where Concord learns producer-consumer function pairs and schedules

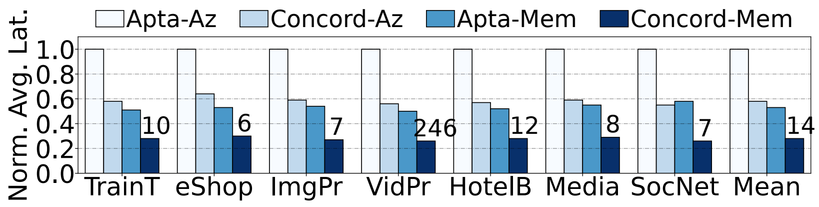


Figure 8.17: Average application latency in Apta and Concord normalized to Apta-Az. The numbers on top of the Concord-Mem bars are the absolute application latency values in ms.

them together). On the other hand, Apta introduces *locality-aware scheduling* (where a function is scheduled on the node where its predecessors executed).

Due to the limited size of our cluster and to consider the worst case for Concord, we compare Concord with 15 compute nodes to the software version of Apta with 15 compute and 15 memory nodes in two cases. First, in *Apta-Az* and *Concord-Az*, both systems need to propagate updates to Azure Blob Storage (like in our Concord design). Second, in *Apta-Mem* and *Concord-Mem*, updates are propagated only to memory nodes (like in the Apta paper); in this case, we add 15 memory nodes in *Concord-Mem* as well. Figure 8.17 shows the average application latency at medium load in all four environments normalized to Apta-Az. The Concord bars include the communication-aware function placement optimization. The numbers on top of the Concord-Mem bars are the absolute latency values in ms.

On average across applications, Concord-Az and Concord-Mem reduce the average application latency over Apta-Az and Apta-Mem by 41.2% and 47.4%, respectively. Focusing on the *Mem* environments, there are three reasons why Concord is faster. First, Concord allows all nodes to continue scheduling new function invocations without interruption, unlike Apta's approach with coherence-aware scheduling. On average, Apta's scheduler has only 8.9 out of 15 compute nodes available for scheduling new invocations at a time.

Second, Apta adds scheduling overheads, as on every function invocation, the scheduler contacts all memory nodes and checks which compute nodes are currently unavailable. Then, it schedules the request on an available compute node. On average, Apta increases the scheduler's response time by 2.8×.

Finally, Concord's coherence-aware scheduling and communication-aware placement optimizations are more effective than Apta's locality-aware scheduling optimization.

In the *Az* case, writes having fewer hops in Concord than in Apta directly affects performance. Indeed, Apta's write operations travel to both the memory nodes (to check the directory) and to the Azure Storage; in Concord, they go directly to the storage if they update a datum homed locally or a datum in E state. This is the reason for the E state in Concord's protocol. On average, Concord reduces the number of hops per write by 28.6%.

8.8 RELATED WORK

Distributed Caching. Researchers have proposed various caching schemes in distributed systems [363, 364, 365, 366, 367, 368, 369]. These proposals target environments where the cache is an independent software system used by long-lived web services. In contrast, Concord targets serverless environments where caches are tightly coupled with the highly-dynamic ephemeral function instances. Some prior works target FaaS [194, 195, 325, 328, 329, 338, 339]. Most proposals do not provide coherent caches [194, 195, 325, 329, 338]. Also, some designs introduce custom APIs and make applications responsible for their data coherence management [325]. We compare Concord to closely related work, namely Faa$T [339] and OFC [338], throughout the chapter. If the developer annotates all read-only objects, Faa$T can avoid version checks for such objects, potentially improving performance. However, in the Azure traces [226], only 5% of the objects are read-only. Thus, Concord still outperforms Faa$T with annotations in this configuration.

Producer-Consumer Optimizations. SAND [300] and Faastlane [193] optimize producer-consumer patterns across the functions of an application workflow. They do not optimize the storage accesses to persistent objects. SAND [300] uses a hierarchy of local and global per-function queues used for local and remote communication, respectively, leading to multiple overheads: (i) communication via queues is serialized, creating bottlenecks in high-load scenarios, (ii) the publish-based messaging is slower than RPCs, and (iii) the hierarchy introduces multiple hops to read/write data when the communicating functions are located on different nodes. Our experiments show that, with conventional scheduling, Concord reduces the average latency of SAND by 8% for workflows with producer-consumer patterns. When Concord adds communication-aware function placement, it reduces the average latency of SAND by 31%.

Faastlane [193] consolidates all functions of an application into a single container, enabling communication through loads and stores with Intel's Memory Protection Keys (MPK). This approach performs similarly to Concord when all accesses hit in the cache. However, it complicates resource management and scaling since the functions within a container can have diverse hardware requirements and software dependencies.

Leases have been extensively used in distributed systems [354, 370, 371, 372]. A lease gives its holder cache the right to read or read-write a cached object for a limited period of time. When the lease expires, the cached object is self-invalidated and the cache must renew the lease to cache the object again. All lease designs induce lease renewal overheads that commonly include communication with the storage server that grants them.

Scheduling. Proposals on locality-aware FaaS scheduling [224, 373] focus on scheduling same-function invocations on the same nodes to minimize cold-starts. Palette [374] co-schedules functions that operate on the same data, providing a new API for users and a load balancer that takes this hint into account. Concord *transparently* prioritizes the co-location of function invocations that have the same input parameters.

8.9 CONCLUSION

This chapter proposed *Concord*, a distributed software caching system for FaaS environments. Concord introduces a directory-based distributed coherence protocol for software caches. The protocol minimizes coherence traffic, reduces contention points, and is robust to failures and frequent creation/removal of application cache instances. Concord on average speeds-up FaaS execution by 2.4× and improves throughput by 1.7×, while using 6.2MB of idle application memory.

CHAPTER 9: Software Speculation for Further Performance Gains

9.1 INTRODUCTION

As indicated above, serverless functions suffer from overheads such as cold start [53, 298, 375, 376], virtualization, RPC or HTTP invocation, and the need to persist outputs to global storage. Unsurprisingly, in applications with many functions, where cross-function control and data dependences are common, such overheads accumulate. The result is long application response times, despite using state-of-the-art frameworks such as AWS Step Functions [377], Azure Durable Functions [378], IBM Cloud Composer [379], or Google Cloud Workflows [380].

In this chapter, we aim to fundamentally accelerate the execution of function-based serverless applications. We want to go beyond current work which, while effective, either focuses on cold-start effects (e.g. [53, 63, 64, 65, 85, 86, 187, 188, 298, 301, 319, 320, 324, 375, 376, 381]) or targets one type of overhead such as cross-function communication [193, 300, 382], data access latency [191, 194, 195, 328, 329, 383], or RPC invocation [43, 44]. What we seek is a novel way to execute applications in this paradigm.

An analysis of serverless applications suggests a way to accelerate this environment. Specifically, we find that the outcomes of the branches that encode cross-function control dependences are fairly predictable. Moreover, in this environment where functions are stateless by definition, cross-function data dependences are often predictable. Indeed, functions often produce the same outputs every time that they are invoked with the same inputs. Hence, the data that a function will generate for the subsequent function is typically predictable.

With these insights, we propose to accelerate serverless applications using software supported *speculation*. Our proposal is termed *Speculative Function-as-a-Service* (*SpecFaaS*). It is inspired by the out-of-order execution of modern processors. In SpecFaaS, functions in an application are executed early, speculatively, before their control and data dependences are resolved. Control dependences are predicted with a software-based branch predictor, while data dependences are speculatively satisfied with memoization. With this support, the execution of downstream functions is overlapped with that of upstream functions, substantially reducing the end-to-end execution time of applications.

While a function execution is speculative, SpecFaaS prevents its buffered outputs from being evicted to global storage. When the dependences are resolved, SpecFaaS proceeds to validate the function. If no dependence violation is detected, the function commits. Otherwise, the buffered speculative data is discarded and the offending functions are squashed and re-executed. SpecFaaS provides policies to configure the degree of speculation based on control/data dependence predictability.

We prototype SpecFaaS on Apache OpenWhisk [57], an open-source serverless computing platform. SpecFaaS runs transparently to the applications. We evaluate SpecFaaS using three application suites, namely *Alibaba* [384], *TrainTicket* [78], and *FaaSChain*. They have a total of 16 FaaS applications of, on average, 12 functions each. In a warmed-up environment, SpecFaaS attains an average speedup of 4.6×. Further, on average, the application throughput increases by 3.9× and the tail latency reduces by 58.7%.

Overall, this chapter makes the following contributions:

• SpecFaaS, a novel approach to accelerate serverless applications with (software-supported) speculative function execution.

• A characterization of dependences and overheads in large serverless applications.

• An implementation and evaluation of SpecFaaS on the OpenWhisk platform.

9.2 BACKGROUND

Serverless Applications. Large, complex serverless applications are organized as workflows of multiple interdependent functions. To construct these workflows, FaaS platforms have composition frameworks, e.g., AWS Step Functions [377], Azure Durable Functions [378], IBM Cloud Composer [379], or Google Cloud Workflows [380]. These frameworks allow developers to specify control- and data-flow dependences between functions, such as producer-consumer relationships, control branches, loops, parallel execution, and error handling.

We implement our infrastructure on top of OpenWhisk [57]. Listing 9.1 shows the code snippet of a smart home application that is described in [385, 386], using OpenWhisk Composer [387]. This application is composed of 7 functions and Figure 9.1 shows the workflow.

Listing 9.1: Code snippet of a smart home FaaS application using OpenWhisk Composer.

```
import composer
def main():
    return composer.when('Login',
        composer.sequence(
            'ReadTemp',
            'Normalize',
            composer.when('CompareTemp', 'TurnAir'),
            'Done'),
        'Fail')
```

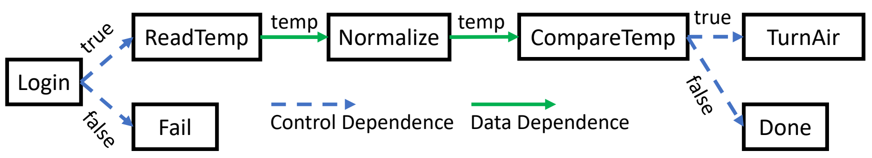


Figure 9.1: The workflow of the FaaS application in Listing 9.1.

The application specifies two types of dependences using the `when` and `sequence` directives. The `when` directive is a branch statement that specifies a control dependence: `when(br_cond, true_target, false_target)` If `br_cond` returns `true`, we execute `true_target`, otherwise, we execute `false_target`. The `sequence` directive is used to express data dependence: `sequence(source, destination)`. The output of the `source` function will be used as the input of the `destination` function. Applications can also use loop structures, which are specified with `while` and `do_while` directives. These directives are compiled to the same code as `when`, and so we will not consider them separately. It is also possible to execute multiple functions in parallel with the `parallel` directive. Such directive is currently not supported by OpenWhisk Python Composer, and thus we have implemented it ourselves.

An application's workflow is exposed to the FaaS platform. However, the code of each function may not be visible. Therefore, we treat every function as a blackbox.

Serverless Workflow Execution. The execution of the functions in an application's workflow needs to satisfy all data and control dependences. For example, in Figure 9.1, Functions `ReadTemp` and `Fail` do not execute until `Login` finishes and returns the condition that determines which function to execute next. Similarly, `Normalize` does not execute until `ReadTemp` finishes and produces its output.

Serverless platforms schedule each function to execute as soon as it reaches the ready state. Typically, function scheduling is done by a *Controller* component, which keeps track of the state of the workflow graph. In OpenWhisk, after a function completes, the controller calls a helper function called *Conductor*. The conductor picks the next function to execute. Then, the controller initializes a *Worker* that first encapsulates the function in an execution environment (a container [388] or a micro VM [85, 389]) and then launches the function.

Implicit Workflows. The workflows described so far are called *Explicit*, in that the developer explicitly specifies the whole graph topology and the controller knows the next function to execute. However, workflows can also be created in an implicit manner. In *Implicit* (or multi-tier) workflows, functions invoke other functions as subroutines. Specifically, a function calls another function, waits on the results, and then continues with its own execution, possibly to call other functions. There are gather functions that invoke multiple simple services and then aggregate the obtained data. An application example from Alibaba [384] is shown in Figure 9.2. Since the FaaS platform may not know the code of functions, it does not know ahead of time which functions a given function can invoke.

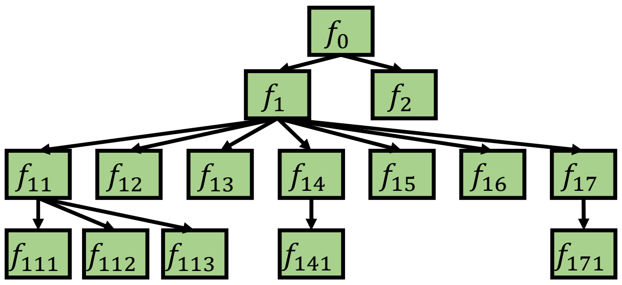


Figure 9.2: Example of an application with implicit workflow.

Studies by Alibaba [384] and Facebook [14] show that the multi-tier paradigm is popular in production-level microservice architectures. The largest open-source serverless applications [40, 78] are also built in this way. Note that the caller function is blocked while waiting for the results from a callee.

9.3 FAAS WORKFLOWS CHARACTERIZATION

Our work is driven by the characteristics of FaaS application workflows. In this section, we analyze three FaaS application suites running on OpenWhisk. These suites are *Alibaba* [384], *TrainTicket* [78], and *FaaSChain*, which we detail in Section 9.7. Alibaba and TrainTicket use implicit workflows, while *FaaSChain* uses explicit workflows. Table 9.1 characterizes these application suites. For each suite, we show the number of applications and, on average per application: the number of functions, the number of cross-function branches, the number of cross-function data dependences, the number of callees per function with calls, the maximum application DAG depth, and the application execution time in a warmed-up environment. For *TrainTicket* and *FaaSChain*, the execution time is obtained by running the applications on AMD EPYC 7402P servers as described in Section 9.7; for *Alibaba*, the execution time is obtained from the traces.

Our analysis reveals the following observations:

Observation 9.1: *Even under warmed-up conditions, function execution per se constitutes less than 1/2 of the time taken to invoke and run an FaaS function.*

Figure 9.3 shows the average response time of a function invocation under cold start conditions in each of the three application suites. Each bar is broken down into five categories, each with a number expressing their duration in ms. The bottom-most category is *Container Creation*, which includes creating the container and network stack, and connecting to the network. This category is the one taking the longest by far (1500ms), and is shown

Table 9.1: FaaS application suites considered.

Characteristic	Alibaba	TrainTicket	FaaSChain
Workflow Type	Implicit	Implicit	Explicit
# of Applications	5	5	6
Per-appl. metrics:			
Avg # Functions	17.6	11.2	7.8
Avg # Branches	N/A	1.8	2.5
Avg # Data Deps.	3.4	4.8	2.7
Avg # Callees/Func.	3.4	4.8	N/A
Max DAG Depth	5	3	10
Avg Exec. Time (ms)	387.2	268.8	160.0

broken into two pieces in the figure. Next is *Runtime Setup*, which involves injecting the function code and starting the docker proxy. This category is also large and, together with the previous one, constitutes the *cold-start overhead*.

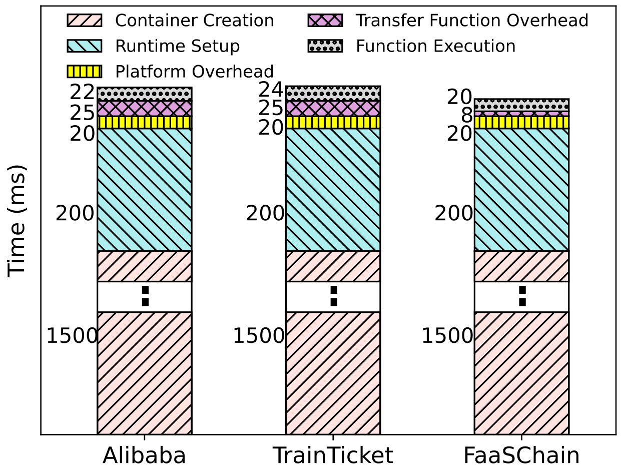


Figure 9.3: Breakdown of a function's response time.

Platform Overhead is the time for the communication between different FaaS platform components such as front-end, controller, and worker when the new request comes. *Transfer Function Overhead* is the time between when a function completes and its successor starts execution. For implicit workflows, this time corresponds to an HTTP or RPC call; for explicit workflows, it corresponds to additional communication between worker and controller, and the execution of the conductor function. *Function Execution* is the actual function execution.

From the bars, we understand the overheads: function execution only accounts for 33-42% of the total function response time in warm-up conditions (or 1% in cold-start conditions). Our goal is to minimize *Platform Overhead* and *Transfer Function Overhead*, and overlap *Function Execution* across functions. In addition, if the system runs under cold-start conditions, we want to overlap *Container Creation* and *Runtime Setup* across functions.

Observation 9.2: *The sequence of functions executed in application is highly deterministic.*

The sequence of functions that an application executes can change across invocations of the application, as some functions conclude with a branch condition that can transfer execution to different functions. In addition, in implicit workflows, some functions call other functions conditionally. In this experiment, we measure, for a given application, the sequence of functions that it executes from beginning to end. We record how many times we observe each of the possible different sequences. We then select the most popular sequence.

On average, the most popular sequence accounts for 90% of the total invocations of the application in Alibaba and 98% in TrainTicket. We do not report the result for FaaSChain because its control dependences use synthetic data (Section 9.7). Based on this result, our goal will be to develop *software branch predictors* to pick functions to execute early, speculatively, before knowing whether they will need to execute.

Observation 9.3: *Most FaaS functions do not read from writable global state; many do not even write to global state.*

While FaaS functions are stateless by definition, in addition to taking inputs and producing outputs, they may read and write global state. In this experiment, we first measure the fraction of functions that either do not read global state or, if they do, they read read-only state. For example, this fraction is 75.8% for TrainTicket and 85.1% for FaaSChain across the runs. These functions are guaranteed to produce the same outputs every time that they are invoked with the same inputs. Given these large fractions, our goal will be to maintain memoization tables that record pairs of {inputs, outputs} observed for functions. These tables will allow us to predict the outputs of functions in advance, hence allowing the early, speculative execution of successor functions.

An interesting subset of these functions are those that, in addition, do not modify the global state. These functions produce the same outputs for the same inputs and have no side effects. The fraction of such functions is 57.6% for TrainTicket and 61.7% for FaaSChain. These functions can not only be memoized, but their execution can be skipped altogether!

Observation 9.4: *Remote storage is not frequently updated.*

We analyze traces of the blob accesses in Microsoft's Azure Functions [191, 226] and observe that write operations are not common. Specifically, out of 40M accesses, only 23% are writes. Moreover, two thirds of blobs are read-only. Out of the writable blobs, 99.9% are written less than 10 times overall. Finally, the time between a write and a subsequent read to the same storage location is more than 1s in 96% of the times, and more than 10s in 27% of the times. This means that reads and writes to the same location are separated from each other, and are rarely issued in the same function invocation.

Observation 9.5: *FaaS functions have few types of side-effects.*

To confirm that the behavior of our application suites is representative, in this experiment, we take another 110 open-source serverless functions from various benchmark suites [171, 192, 227, 229, 390] and analyze their side effects. Our analysis shows that, in agreement with the previous sections, a major portion of the functions (63.4%) does not have any side-effects. The rest have only three types of side-effects: writes to global storage, writes to temporary local files, and HTTP requests.

Observation 9.6: *CPUs are not fully utilized in the cloud.*

Recent studies on resource utilization of cloud and datacenter systems [152, 316, 317, 318, 391] report that computing resources are commonly over-provisioned and not fully utilized. In this experiment, we extract from the Alibaba traces [384] the CPU utilization of the bare-metal nodes in the Alibaba cloud. For each node, we compute the P90 CPU utilization—i.e., the utilization U such that, for 90% of the time, the CPU is utilized U or less. We then generate the P90 distribution for all the nodes in the cluster. Figure 9.4 shows the CDFs for P90, P80, P70, P60, and P50. We can see that, most of the time, the CPU usage is 60-80%. Thus, the environment could support some cycles wasted due to misspeculation.

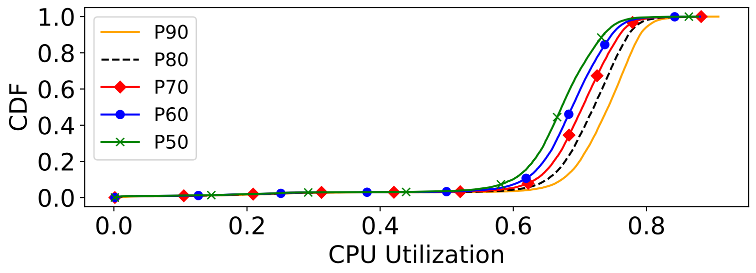


Figure 9.4: P50-P90 CPU utilization of Alibaba's bare-metal nodes.

9.4 SPECFAAS OVERVIEW

With SpecFaaS, we propose a new approach to accelerate serverless applications that are composed of multiple functions. The idea is to optimistically execute functions speculatively, before their control or data dependences are resolved. Later, if a mis-speculation is detected, the optimistically-executed functions are squashed and potentially re-started. SpecFaaS' rationale is our observations that the sequence of functions executed in an application is highly predictable and that individual functions, when passed the same inputs, typically produce the same outputs. As a result, with SpecFaaS, we can substantially reduce the end-to-end execution time of serverless applications.

SpecFaaS is inspired by the out-of-order execution of instructions in modern processors—a function in SpecFaaS is analogous to an instruction in processors. Consider Figure 9.5(a), which shows one possible conventional execution of the smart-home application of Figure 9.1. All functions are executed in sequence. There are two types of cross-function dependences that prevent any concurrent execution of functions: control (e.g., between *Login* and *ReadTemp*) and data (e.g., between *ReadTemp* and *Normalize*). If SpecFaaS correctly predicts the cross-function control dependences, the execution timeline will be Figure 9.5(b); if SpecFaaS correctly predicts both control and data dependences, the timeline will be Figure 9.5(c).

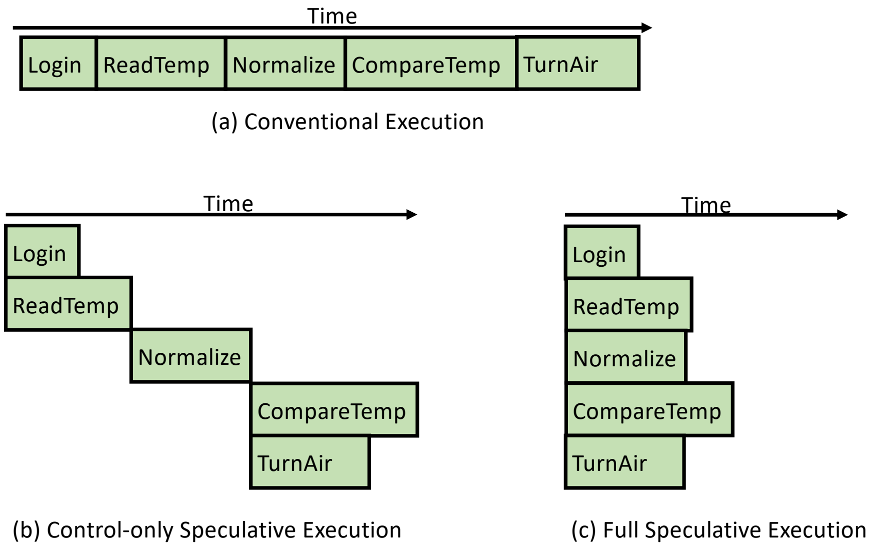


Figure 9.5: Possible execution of the smart-home application of Figure 9.1: conventionally (a) and with speculation (b and c).

To predict control dependences, SpecFaaS adds to the FaaS controller a *Branch Predictor* software table with an entry for each function that may invoke different functions. The entry contains probability information to decide on the next function to invoke. When a function is about to execute, the branch predictor is queried. If an entry is found, the outcome with the highest probability is identified and the corresponding function is also invoked, speculatively.

To predict data dependences, the FaaS controller is augmented with a *Memoization* software table for each function that creates outputs. For a given function, the table contains the pairs of {input, output} values that the function has taken and produced, respectively, in the past. When a function is about to execute with certain input values, its memoization table is queried. If an entry with such input values is found, the corresponding output values are retrieved and the next function of the application in sequence is concurrently invoked, speculatively, taking the retrieved values as inputs.

A function may read data values beyond those that it explicitly takes as inputs. Moreover, it may write other variables beyond those declared as outputs. Consequently, as a function executes speculatively, its global writes are buffered in a *Data Buffer* and not merged with the global state until the function's speculative execution is validated. The Data Buffer is shared by all the concurrently-running functions of a given application invocation. Similarly, global reads first access the Data Buffer, to check if the Data Buffer contains the desired data, as the current function or earlier, less-speculative functions may have generated it. The Data Buffer is also used to detect data dependence violations—i.e., a speculative function that has read data that is later updated by a predecessor function.

On a misspeculation, the FaaS controller squashes the mispredicted function and all its successors, and invalidates the corresponding data in the Data Buffer. In case of a control misspeculation, the controller then launches the correct-path functions (Figure 9.6(a)); in a data misspeculation, the controller re-launches the successor functions, now with the correct input values and correct Data Buffer state (Figure 9.6(b)).

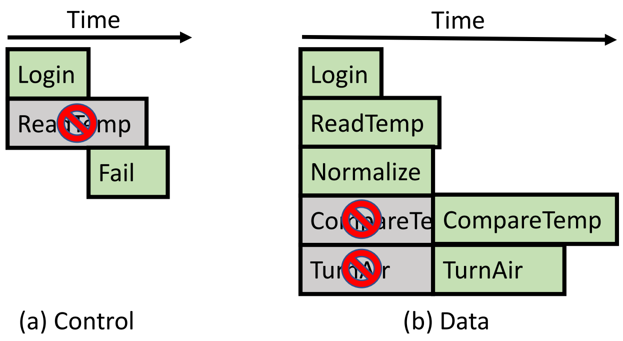


Figure 9.6: Control (a) and data (b) misspeculations.

To reduce execution overheads, SpecFaaS keeps in the FaaS controller node a software table with the sequence of functions that the application needs to execute. This *Sequence Table* is created at the application compile time. For each function, it records the next function to execute. For functions that may invoke one of multiple possible functions, the entry incorporates the branch predictor entry. The Sequence table allows the controller to immediately identify the next function to call without the need to invoke a component like the Conductor in OpenWhisk, hence minimizing the Transfer Function Overhead (Section 9.3).

SpecFaaS is a generic design for modern serverless infrastructures. It works with both implicit and explicit application workflows (Section 9.2). Moreover, while in this chapter it targets warmed-up environments, it is also effective if the FaaS infrastructure is not equipped with any of the previously-proposed optimizations that remove the function cold-start overheads shown in Figure 9.3. Indeed, consider Figures 9.5(a) and 9.5(c). Each of the bricks shown may or may not include the function start-up overheads; in either case, SpecFaaS can reduce the execution time of the application substantially. In this chapter, we will assume by default that all the function start-up overheads have been removed by previously-proposed optimizations. Finally, while we implement SpecFaaS in the OpenWhisk serverless framework [57], it can be implemented in other frameworks as well.

9.5 SPECFAAS DESIGN

Figure 9.7 shows an overview of the SpecFaaS system. For a given application, the controller node maintains a software structure representing the pipeline of not-yet-committed functions of the application (*Function Execution Pipeline*). Functions are ordered in program order and tagged based on whether they are speculative or not, and whether they have completed or not. In addition, the controller keeps the application's Sequence table (which includes the Branch Predictor), the application's Data Buffer and, for each function in the application, the Memoization table. The Sequence and Memoization tables can remain in the controller across invocations of the application and, at every invocation of the application, are augmented with more execution information.

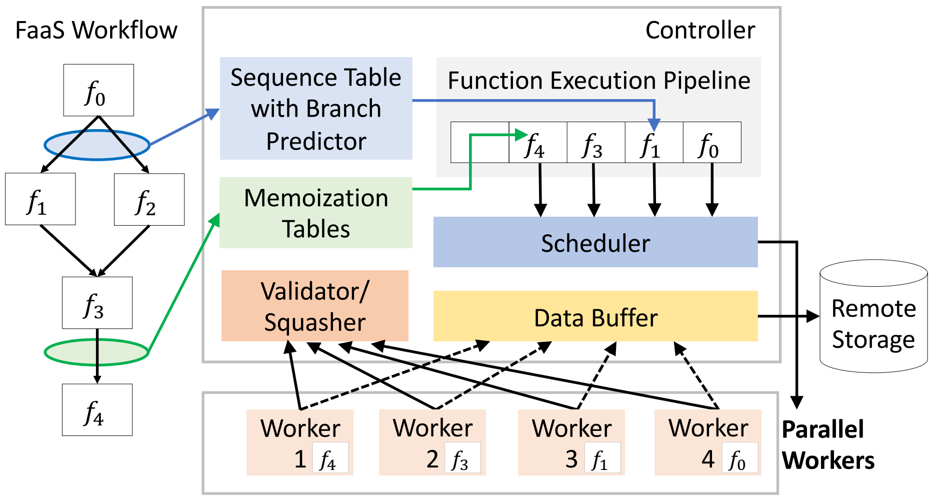


Figure 9.7: Overview of the SpecFaaS system.

During execution, the controller repeatedly picks the next function to execute (speculatively or not) from the Sequence table, launches it in one of the nodes, and detects any misspeculations. If a function misspeculates, the controller squashes and potentially relaunches the function and its successors; otherwise, it eventually commits the function. In the process, the controller updates the application's Function Execution Pipeline, Sequence table, Data Buffer, and Memoization tables. We now describe the main structures.

9.5.1 Sequence Table and Branch Prediction

The Sequence table lists the ordered sequence of functions to be executed—like the sequence of instructions in a program. It enables the controller to pick the next function to launch with minimal overhead. However, like programs, FaaS applications have branches that SpecFaaS wants to predict in advance. Hence, the entries of the Sequence table for functions with branches are augmented with a branch predictor entry.

Figure 9.8(a) shows an application where function f_2 can be followed by either f_3 or f_6. Figure 9.8(b) shows the application's Sequence table. The entry for f_2 includes a pointer to the entry for f_6 and a branch predictor entry. The latter contains state that determines whether the controller should take the branch to f_6 or proceed to the next entry, i.e., f_3's.

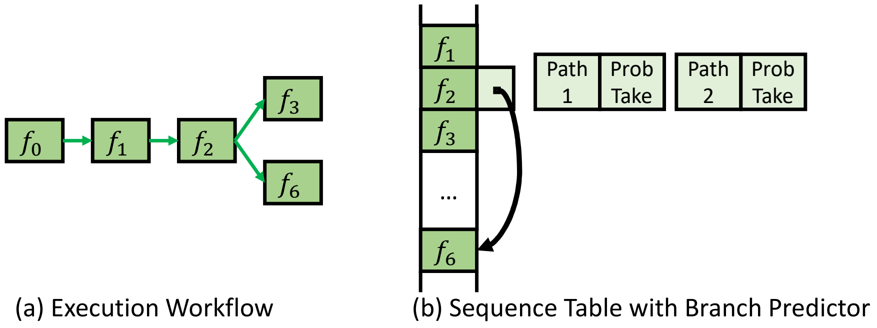


Figure 9.8: Execution workflow and corresponding Sequence table with Branch Predictor.

The branch predictor entry contains the probability to take the branch. Such probability is obtained by recording the outcome of previous invocations of f_2. If the probability is higher than a threshold, the controller will speculatively launch f_6; if it is lower than another, lower threshold, it will speculatively launch f_3. We find that branches in FaaS applications are highly biased. Specifically, we find that the *path* of functions executed from the beginning of the application until the branch typically determines the branch outcome. For example, it may be that if f_2 is reached from f_0 and f_1, the branch is typically taken, but if reached from f_0 and f_{11} (not in the picture), it is typically not taken. Hence, as shown in Figure 9.8(b), a branch prediction entry has a sub-entry for each possible path that reaches f_2.

If the branch at a given function f can have N targets, then the Sequence table entry for f has N-1 pointers and a branch predictor entry with the probabilities of invoking each of the targets (for each path reaching f).

The controller keeps a record of the sequence of functions executed so far. On a branch, it checks the appropriate predictor and decides which speculative path to follow. Once the branch resolves non-speculatively, the controller updates the predictor and, if the prediction was incorrect, squashes the mispredicted functions and executes the correct path.

9.5.2 Function Memoization

A function often produces outputs that are consumed as inputs by its successor function. To speed-up execution, SpecFaaS executes the successor function speculatively without waiting for the predecessor function to complete. To accomplish this, the controller maintains a per-function Memoization table. The table contains input values that a function has taken-in in the past, and the corresponding output values that the function has produced. With this support, when the controller launches a function *f* with a given set of inputs, it checks *f*'s Memoization table. If the specific input values are found, the controller retrieves the output values from the table and speculatively launches the successor function of *f* passing the retrieved values as inputs. When *f* commits, the controller validates *f*'s execution.

Note that, even if *f* is invoked with inputs present in its Memoization table, we still need to execute *f*. This is because many FaaS functions also read and write global (i.e., non-private) data beyond their inputs and outputs. As a result, *f* may read global data that causes *f* to produce unexpected outputs. Further, *f* may issue updates to global state, which cannot be skipped. In addition, *f* may get squashed before completing execution. For example, the Data Buffer may detect a dependence violation (Section9.5.3), where *f* reads a global variable that is later written by a predecessor of *f*. Because of all these cases, *f* must execute and validate at its commit time that it produces the expected outputs. If *f* generates unexpected outputs, when *f* commits, *f*'s Memoization table is updated.

Some FaaS functions are *pure*, meaning that they do not read or write any global state. Therefore, the inputs that they take fully determine the outputs that they will generate, and they do not have side effects. To speed-up execution, SpecFaaS allows programmers to declare pure functions using the `pure-function` annotation (Section9.6). When the controller is about to launch a function and finds from its Memoization table that the function is pure, it skips the execution of the function and launches the successor function with the outputs retrieved from the table.

If a function takes input values that are not yet in its Memoization table, the successor is not launched until the predecessor completes. Later, when the predecessor completes and commits, the pair of input-output values are saved in a new row of the Memoization table.

Our measurements of real-world datasets show that Memoization table sizes are relatively modest. The combined tables for all the functions in one application use 100 to 1K entries, which consume 1.5KB to 30KB.

9.5.3 Data Buffering

In serverless environments, nodes may have software caches that temporarily store remote data accessed by a locally-running function [191, 194, 195, 328, 329]. While different designs are possible, the goal of the caches is for functions to be able to re-access previously-accessed data with low latency.

In SpecFaaS, because some functions are executed speculatively, we need one additional level of data buffering per application invocation. A new buffer, called the Data Buffer, exists in the node running the controller for the application invocation. The Data Buffer can receive requests from all the nodes that are currently running functions of the application invocation. Its goal is to detect and manage data dependences between two concurrently-executing functions of the application: (i) a non-speculative and a speculative function, or (ii) a speculative and a more speculative function. If an in-order RAW dependence is detected, the requested data is forwarded from the Data Buffer to the node running the successor function; if the RAW dependence is out-of-order, the controller sends a squash signal to the successor function (and to the successor's successors, recursively). The Data Buffer is also able to manage in-order and out-of-order WAW and WAR dependences without squashes.

At a high level, the Data Buffer is used as follows. When a function updates a record, the runtime system, in addition to updating the local cache, it sends the update to the Data Buffer. The Data Buffer stores the update and checks if any successor function has prematurely read the record. If so, the successor function (and its successors) are squashed and re-started.

When a function reads a record that the function has not accessed before, the runtime system sends the request to the Data Buffer, which provides the record. The record is also stored in the local cache. Note that the Data Buffer is only accessed if the read is *exposed*—i.e., the function has not yet read or written the record. If the read is not exposed, the read gets the data from the local cache.

In more detail, the Data Buffer is a table with a row for each record accessed by the in-progress functions of the application—i.e., the functions that are being executed or that have just executed but not yet committed. Each row has the address of the record plus a circular buffer with as many columns as the maximum number of in-progress functions supported, ordered by their program order. Each column has Valid (V), Read (R), and Write (W) bits, and space to store the updated record.

Figure 9.9 shows a Data Buffer with two rows and three columns per row. Assume that the non-speculative function uses the leftmost column and, as successor functions are executed speculatively, they take the second and third columns in order. When a write or a read to a record from function i reaches the Data Buffer, the controller accesses the row for the record and performs the following operations.

Write Operation. The controller scans the R bit of all the successor functions of i in order. The scanning ends at (and includes) the first column that has the W bit set. If any column has the R bit set, the controller sends a squash message to the corresponding function and all its successors. Then, the columns for the squashed functions are invalidated and re-assigned

Address	$Function\ i-1$				$Function\ i$				$Function\ i+1$			
	V	R	W	Data	V	R	W	Data	V	R	W	Data
Record 1	1		1	Value 1								
Record 2									1	1		Value 2

Figure 9.9: Data Buffer for an application invocation. Each row corresponds to a record, and each column to an in-progress function of the application invocation.

to a new speculative execution of the squashed functions. Moreover, the local cache of the squashed functions is invalidated. The updated record is stored in Function i's column and the W bit is set.

Read Operation. The controller scans the W bit of all the predecessor functions of i in reverse order. As soon as a set W bit is found, the data in that column is read and provided to the requester. If no W bit is set, the data is requested from the global storage and provided to the requester. In either case, the R bit in Function i's column is set.

When a Function i that executes non-speculatively completes, it can commit. In the Data Buffer, committing involves writing back to global storage the records in Column i that have the W bit set, clearing out Column i, and re-assigning it to a new, most speculative function. The immediate successor function becomes non-speculative.

As an example, consider Figure 9.9. If Function i issues a write to *Record 2*, an out-of-order RAW dependence is detected and the controller will squash Function *i+1*. If Function i issues a read to *Record 1*, an in-order RAW dependence is detected and the Data Buffer will provide *Value 1* generated by Function *i-1*.

The logic described seamlessly handles WAR ($R_1 \rightarrow W_2$) and WAW ($W_1 \rightarrow W_2$) dependences. Indeed, assume out-of-order dependences: W_2 occurs first; later, when the other access (R_1 or W_1) occurs, it will neither read from W_2 nor squash W_2's function. Assume now in-order dependences: when W_2 occurs last, it affects neither R_1 nor W_1.

Minimizing the frequency of squashes. It is possible that a communication over remote storage between two functions of the same application triggers a squash in many of the application's invocations. To avoid this case, we augment the controller as follows. When the controller detects that a function is frequently squashed due to prematurely reading a given record that a predecessor function later updates, the controller remembers the producer-consumer function pair and the record causing the data dependence. The next time that the consumer tries to read the record, if the record is not yet updated by the producer, the controller stalls the consumer. The consumer remains stalled until the producer either updates the record or completes its execution. With this support, we minimize the number of squash operations.

9.5.4 Speculating Implicit Workflows

The presence of applications with implicit workflows requires some extensions to SpecFaaS. To describe them, we use as an example the workflow of Figure 9.10(a), where function f_1 can call subroutine functions f_2 and f_3.

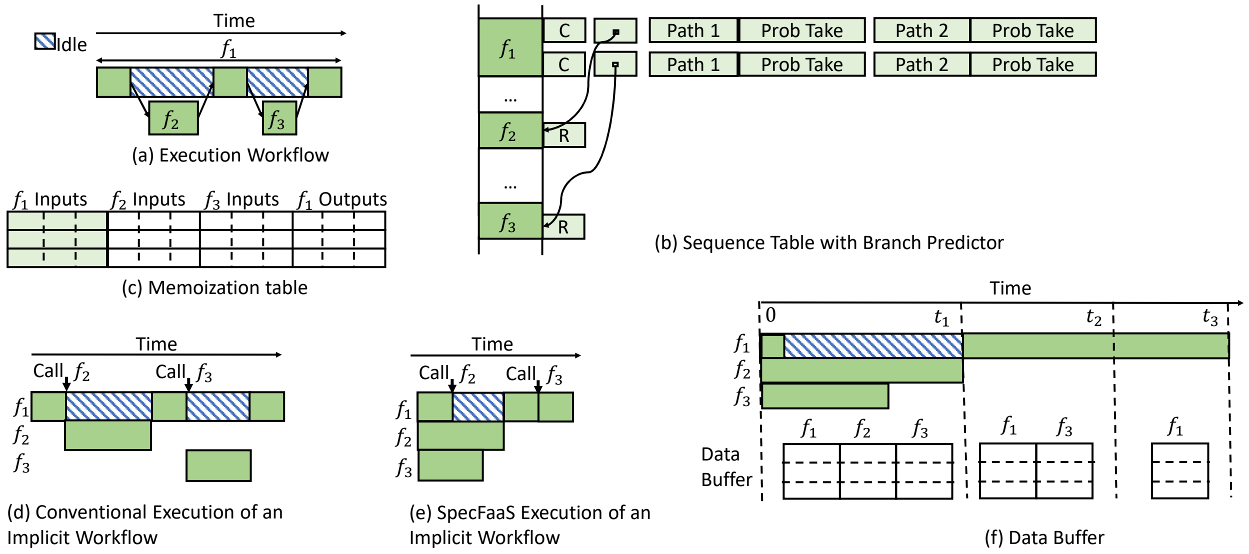


Figure 9.10: Extensions to the SpecFaaS structures to support implicit workflows.

As indicated in Section 9.2, FaaS frameworks do not typically know the internals of functions and, therefore, do not know the static call graph of implicit workflows. Consequently, SpecFaaS may be able to augment the Sequence table, Data Buffer, and Memoization tables with information for f_2 and f_3 *only* after one or more dynamic invocations of f_1. After those, the structures are augmented as follows.

The Sequence table is augmented to support implicit workflows as shown in Figure 9.10(b). The entry for f_1 has as many pointers as functions it can call. The pointers have a Call (C) bit set, to indicate that this is a call. In the example, the f_1 entry has pointers to the entries for f_2 and f_3. Each of these entries has a Return (R) bit that, when set, tells the controller to return to the caller on completion. In addition, the entry for f_1 has one Branch Predictor entry for each of the functions it can call. As in Figure 9.8, a Branch Predictor entry has as many sub-entries as possible paths that may reach f_1. Thus, when the controller launches f_1, it checks the two Branch Predictor entries and decides whether to speculatively launch any combination of f_2 and f_3—possibly along with the function following f_1 in sequence.

The Memoization table of f_1 is also augmented to support implicit workflows. As shown in Figure 9.10(c), each row of the f_1 table contains, in addition to the usual f_1 input and output values, the expected input values for f_2 and f_3. With this support, when the controller launches f_1 (speculatively or otherwise) with the input values from one row of the Memoization table, it can also speculatively launch f_2 and f_3 with their corresponding input values from the same row.

With these structures, we can now examine the execution of the f_1, f_2, and f_3 functions. Figure 9.10(d) shows the conventional execution of this implicit workflow, assuming that both f_2 and f_3 are executed and that we have three cores (one for each row of Figure 9.10(d)). We see that f_1 stalls while f_2 and f_3 execute, keeping the core idle. In contrast, Figure 9.10(e) shows the execution under SpecFaaS. The three functions execute concurrently, with at least f_2 and f_3 executing speculatively. When the core running f_1 reaches the point where it needs to call f_2, if f_2 has not yet completed, it stalls until f_2 completes; then, it resumes executing f_1. We choose to stall because the continuation of f_1 is often data-dependent of f_2 and, if these dependences are violated, the whole f_1 has to be squashed. The same policy is used for f_3. However, as shown in Figure 9.10(e), when f_1 reaches the point of calling f_3, f_3 has already completed. Therefore, f_1 does not need to stall.

Our policy of stalling the caller until the callee has finished executing speculatively is also key to simplifying the operation of the Data Buffer. With this policy, before a function f calls a subroutine s, both f and s have columns in the Data Buffer and f is a predecessor of s. When s returns, the Data Buffer column of s is merged into the column for f.

Figure 9.10(f) shows the Data Buffer for our example workflow. Between times 0 and t_1 (when f_2 completes), the Data Buffer has columns for f_1, f_2, and f_3—as usual, columns to the right are more speculative. At t_1, the f_2 column is merged into f_1's. Between t_1 and t_2 (when f_1 is about to call f_3 and finds that f_3 has already completed), the Data Buffer has two columns. At t_2, the f_3 column is merged into f_1's. At any time, reads and writes always manipulate the Data Buffer with the algorithm of Section9.5.3.

9.5.5 Implications on Scalability and Security

SpecFaaS does not introduce scalability bottlenecks. In particular, the controller is not a bottleneck, as a machine has many independent controllers spread across different nodes—each controller managing a set of invocations of the same or different applications. The same is true for the controllers in current systems.

In current systems, the controller for an application invocation already keeps track of the function chain's progress and receives results from all the functions in the chain. With

SpecFaaS, the controller additionally keeps the sequence table with the branch predictor, memoization tables, and Data Buffer for the chain. These structures are relatively small. The sequence table with the branch predictor and the memoization tables are kept on a per-application basis, the Data Buffer is kept on a per application invocation basis. The Data Buffer is destroyed at the end of the application invocation.

SpecFaaS can guarantee that executing and squashing speculative functions does not result in information leakage as follows. First, structures like branch predictors and memoization tables are never updated with speculative data. Second, on a squash, any software structure that holds speculative state (e.g., the Data Buffer) is invalidated. Finally, any state that a speculative function loads into microarchitectural processor structures (e.g., caches or TLBs) can be handled with known defenses (e.g., partitioning the structure, flushing the structure on context-switch or squash, etc.). We implement the first two but not the last one. The reason is that, for the microarchitecture to be safe from speculative attacks, one would also have to implement known defenses against branch miss-speculation attacks (e.g., Spectre), which would slow-down both SpecFaaS and the baseline configuration substantially.

9.6 SPECFAAS IMPLEMENTATION ASPECTS

We have implemented SpecFaaS on top of Apache OpenWhisk [57], a serverless cloud platform. SpecFaaS runs transparently to serverless applications. We modify the OpenWhisk controller to support speculative execution, the software structures required, and the function runtime to intercept remote storage operations. The implementation takes about 1K lines of Scala and 500 lines of Python. In this section, we describe some of the implementation challenges.

Minimizing Squash Cost. When SpecFaaS detects a mis-speculation or dependence violation, it needs to squash the incorrectly-executed functions. There are three ways in which SpecFaaS can implement the squashing mechanism. First, SpecFaaS could allow the squashed functions to complete their execution in the background but never propagate the functions' global state updates. This approach allows container reuse for subsequent invocations but wastes CPU cycles. Second, SpecFaaS could squash the functions by terminating the containers as soon as the mis-speculation or violation is detected. This approach frees the CPUs from useless execution, but it takes a long time to stop a container (≈10s). Moreover, this approach disallows the reuse of the containers for subsequent invocations. Finally, our chosen mechanism is to kill only the processes executing the functions inside the containers while leaving the containers alive. This approach frees CPU cycles, is very fast (≈1ms), and allows safe container reuse for subsequent invocations.

We modify the function runtime. Instead of having a single process perform the function initialization and serve all invocation requests, the SpecFaaS runtime has two types of processes: the *Initializer* process that performs the function initialization and waits on new requests, and *Handler* processes that serve requests. When the initializer receives a request, it forks a handler process and forwards the input values to it. The handler executes the function, returns the results, and dies. Every request is served by a new handler process. In this way, squashing a function is cheap, as it only involves terminating the handler process. The container and its initializer process remain active and can serve subsequent requests.

Side-effect Handling. As observed in Section 9.3, FaaS functions have three common types of side effects: writes to global storage, writes to temporary local files, and HTTP requests. SpecFaaS handles writes to global storage with the Data Buffer mechanism (Section 9.5.3). Writes to temporary local files are handled by a scheme similar to copy-on-write for memory pages. As long as a handler process only reads from a file, it can use the shared initial file. However, when it tries to update the file, it creates its own temporary file, with a unique name. From this moment on, all reads and writes by the handler are directed to the new temporary file. When the handler completes execution, all of its temporary files are discarded. We implement this functionality by intercepting file-related syscalls and redirecting the operations to the appropriate files.

SpecFaaS detects when a speculative function tries to issue an HTTP request that is not a function call or a storage access. Then, it delays the operation until the function turns non-speculative. This is implemented by intercepting *sendto* socket syscalls and checking flags to see the origin of the call and if the caller function is speculative. When a stalled speculative function turns non-speculative, if the speculation is validated, the *sendto* operation in performed; otherwise the function is squashed without performing the *sendto* operation.

In the applications we measure in this chapter, functions do not have other side effects. However, SpecFaaS handles a wide range of other potential side effects. It handles them in a manner similar to how it handles the *sendto* operation. Specifically, all globally-visible side effects need to invoke syscalls. Some syscalls are safe, while others are not. For example, the getters syscalls (i.e., get PID, UID, time, capabilities, file-status, etc.) account for about 50 syscalls and are safe. SpecFaaS maintains a list of unsafe syscalls and, whenever a speculative function issues one such call, SpecFaaS intercepts it and suspends the function until the function turns non-speculative.

Storage Request Interception. SpecFaaS is transparent to the application. Its runtime intercepts read and write operations issued by functions to the global storage. Specifically, we modify the function runtime to intercept the `get` and `set` operations to the Redis [315] key-value store. The runtime redirects operations first to the OpenWhisk controller, which may update the Data Buffer. When a function commits, its buffered updates are flushed to global storage; when a function is squashed, its buffered updates are discarded. Given that the key-value interface is the main storage interface for FaaS [40, 64, 194, 195, 325], our interception is generically applicable to other storage services.

Function Annotations. SpecFaaS gives programmers the option to specify custom speculation policies for functions in a workflow. We implement this capability with OpenWhisk's function annotation support [392]. SpecFaaS currently supports two annotations. The first one is `non-speculative`, which specifies that the function should not be executed speculatively. In this case, the controller does not launch the function until all its predecessor functions are committed. We use this annotation when we know that the function has dependences that would typically induce squashes. The second annotation is `pure-function`, which specifies that the function is pure and, therefore, the controller should skip its execution if it finds a matching input in the function's Memoization table.

Configurability. SpecFaaS provides configurations to tune the level of speculation based on the workload type and the load of the machine. First, SpecFaaS does not perform branch speculation for branches whose current probability of being taken is within a configurable, short range around 50%. Second, SpecFaaS reduces the depth of speculation (i.e., the number of functions that are speculative at a time) to a configured threshold when the load of the machine is above a certain other threshold.

9.7 EXPERIMENTAL METHODOLOGY

Platform. We run our experiments on five AMD EPYC 7402P servers. Each server has one socket with 24 cores (each 2-way multi-threaded), a 128MB Last Level Cache (LLC) and 128GB of DRAM. The OS is Ubuntu 20.04.2 LTS.

Our evaluation focuses on warmed-up scenarios, where functions and containers are available in main memory. Before starting our measurements, we first run the functions, and do not evict any containers from memory—our servers have sufficient memory to host all the containers in memory. Therefore, our evaluation does not count cold-start effects, which have been the main focus of much prior work (e.g., [53, 64, 191, 300, 301, 393, 394, 395, 396]). Note that SpecFaaS is effective in both warmed-up and cold-start scenarios (Section 9.4).

We set the following values for the configurable parameters. First, a RAW dependence that causes the squash of a given function three times in a row triggers the controller to stall

Table 9.2: Applications used in the evaluation.

FaaSChain – *6 real-world FaaS applications with explicit workflows*	
Login	Login user and send profile info (3 functions)
Banking [397]	Withdraw money from account (6 functions)
FlightBook [398]	Book flight, hotel and car for trip (11 functions)
HotelBook [40]	Book the best available nearest hotel (7 functions)
SmartH [390]	Turn A/C if temp is above threshold (7 functions)
OnlPurch [386]	Buy an item from the closest store (13 functions)
TrainTicket – *5 open-source FaaS applications with implicit workflows*	
TcktApp	Get all tickets for a given trip (15 functions)
TripInfApp	Get information about the trip (24 functions)
QueryTrvl	Get travel-specific information (8 functions)
GetLeftApp	Get unsold tickets for given time frame (5 functions)
CancelApp	Cancel an order (4 functions)
Alibaba – *5 implicit workflow applications from production-level traces*	

the function. Second, branches whose current probability of being taken is 60% or higher, 40% or lower, or between 40-60% are speculatively taken, speculatively not taken, and not speculated, respectively. Finally, for machine loads between 60-70%, between 70-80%, and above 80%, the maximum speculation depth is 6, 5, and 4 functions, respectively.

Application Suites. To comprehensively evaluate SpecFaaS, we use three application suites: *FaaSChain*, *TrainTicket* [78], and *Alibaba* [384]. Table 9.2 lists the applications in each suite, and Table 9.1 characterizes these suites.

FaaSChain. We developed this new FaaS application suite that has six real-world FaaS applications. The applications have different characteristics, with chain lengths varying from 2 to 10. Functions are implemented in Python and use explicit workflow, following the best practices in AWS [399].

TrainTicket. We select five representative applications from the serverless TrainTicket suite [302]. Applications are composed with implicit workflows, mainly because the code is ported from a microservice-based implementation.

Alibaba. We select five representative implicit application workflows from Alibaba's production microservice traces [384]. Traces provide the call graphs of each application (from which we infer the workflow) and the execution time of each function of the application. However, the function code is unavailable. For space reasons, in the evaluation, we show data only for the average across the five applications.

Application Input Data Sets. For *FaaSChain*, we use real-word data sets from repositories in the web [400, 401, 402]. In some cases, the data set does not provide enough information to determine the outcomes of control dependences, such as for login information in the workflow of Figure 9.1. In such cases, we use synthetic data for the outcomes of branches. Specifically, we assume a 90% hit rate for the branch predictor, which is the average hit rate observed in Alibaba's traces. We discuss the impact of branch prediction accuracy in Section 9.8.5.

For *TrainTicket*, we lack a sizable input data set of train tickets. Hence, as a high-fidelity input set, we use a real-world dataset of three million airline tickets purchased in 2021 from the Bureau of Transportation Statistics [403].

Like in prior research on serverless systems [196, 300, 319, 320, 321, 322, 395], we use the Poisson distribution to model the request inter-arrival time. In the evaluation, we use *Low*, *Medium*, and *High*, to refer to load levels of 100, 250 and 500 application requests per second (RPS), respectively.

9.8 EVALUATION

9.8.1 Response Time and Speedups

We measure the end-to-end response time of applications, from when the client sends a request to the point it receives the result. Using this response time, we compute the speedup of SpecFaaS over the OpenWhisk baseline. Recall that all our experiments are performed under warmed-up conditions. Figure 9.11 shows the average speedup of each application for different loads. The figure also shows bars for the average of each application suite.

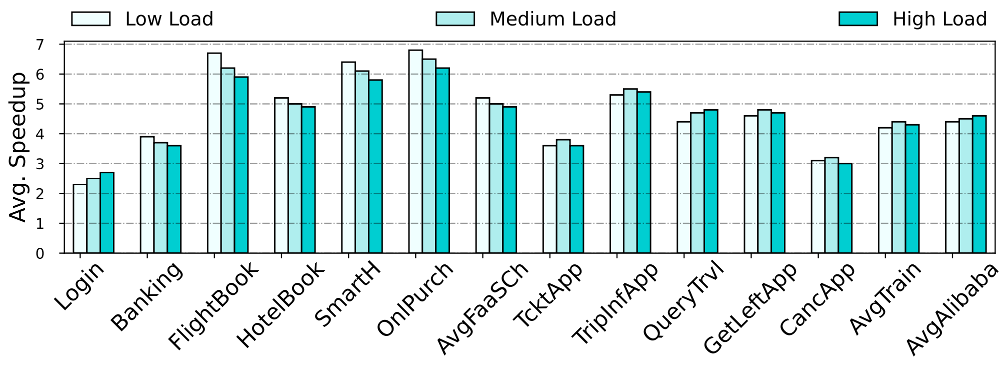


Figure 9.11: Speedup of SpecFaaS over the baseline for different request loads.

SpecFaaS effectively reduces the response times and, as a result, delivers speedups across all applications and load levels. On average, the speedup is 4.6×.

Consider the FaaSChain applications. On average, SpecFaaS delivers speedups of 5.2×, 5.0×, and 4.9× in low, medium, and high load, respectively. Typically, the speedups slightly decrease with higher load because of the higher resource utilization induced by parallel function execution in SpecFaaS. However, for short-running applications like Login, speedups increase. The reason is that, as the load increases, Platform and Transfer Function overheads (Section 9.3) account for a larger fraction of the execution time, and SpecFaaS reduces them.

The TrainTicket and Alibaba applications show similar speedups. For TrainTicket, SpecFaaS attains average speedups of 4.2×, 4.4×, and 4.3× in low, medium, and high loads. For Alibaba the average speedups are 4.4×, 4.5×, and 4.6×.

As indicated in Section 9.4, SpecFaaS is effective in both cold and warmed-up environments. We repeat the experiments in Figure 9.11 without warming-up the environment and see similar average speedups across all loads: 5.2×, 4.5×, and 4.7× for FaaSChain, TrainTicket, and Alibaba, respectively.

9.8.2 Speedup Breakdown

We attribute the speedups of SpecFaaS to three main components: branch prediction, data memoization, and squash optimization. Squash optimization was described in Section 9.6: rather that using the naive approach of terminating the containers on mis-speculation or violation (second approach in that section), SpecFaaS terminates processes but not containers. To assess the impact of each component, we apply one technique at a time. Figure 9.12 shows the speedups for the different applications averaged across all loads. We apply, in order and cumulatively, branch prediction, data memoization, and squash optimization.

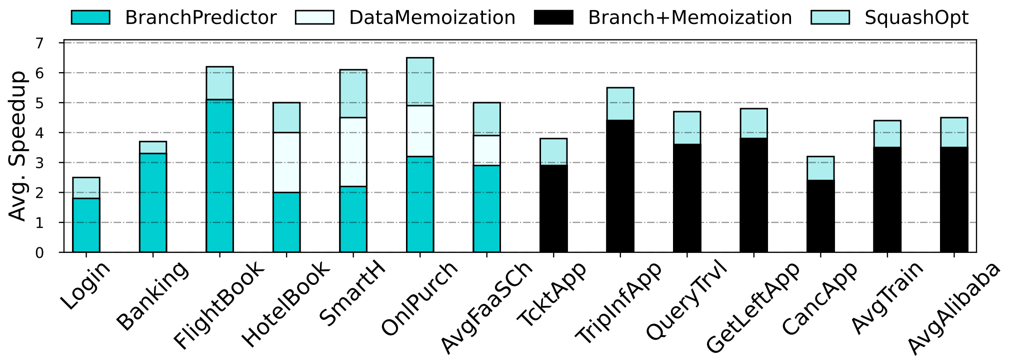


Figure 9.12: Breakdown of SpecFaaS speedups.

Three of the FaaSChain applications (Login, Banking, Fight Booking) do not have data dependences; consequently, their bars have only two categories. Moreover, for implicit workflows, as used in TrainTicket and Alibaba, the branch predictor and the memoization table optimizations cannot work without each other. First, without branch prediction as shown in Figure 9.10(b), we cannot even build a memoization table, like the one in Figure 9.10(c). Second, consider a function *f* that calls subroutines. Without a memoization table for *f*, branch prediction cannot help because we cannot speculatively call the subroutines since we do not know their inputs. In contrast, in explicit workflows, if *f* finishes with a branch, we know the input of its successor functions because they take the same input as *f*. Therefore, for TrainTicket and Alibaba, a single category combines branch prediction and memoization.

The figure shows that all three techniques contribute substantially to the speedups. Consider the branch predictor first. Without it, SpecFaaS would not be able to speculate on control dependences. With it, SpecFaaS attains an average speedup of 2.9× in FaaSChain. The branch predictor of SpecFaaS obtains an average branch prediction hit rate of 98% and 90% in TrainTicket and Alibaba, respectively. For FaaSChain, we assume a 90% hit rate.

Consider now memoization. Without it, SpecFaaS would not be able to speculate on data dependences. With it, SpecFaaS attains substantial additional speedups. The combination of branch prediction and memoization delivers average speedups of 3.9×, 3.5×, and 3.5× for FaaSChain, TrainTicket, and Alibaba, respectively. A modest-sized memoization table suffices. For example, a 50-entry memoization table obtains an average 96% hit rate in TrainTicket applications. In FaaSChain applications, which vary more, the average hit rate with 50 entries ranges from 65% to 98%. In addition, many functions are pure and, therefore, SpecFaaS could use the memoization table to avoid executing them completely. For example, this is the case for more than 57.6% of the function invocations in TrainTicket. However, to be conservative in the evaluation, we do not perform this additional optimization. The Data Buffer size is also modest: it has at most 12 columns and 4 rows, for a total of 3KB.

Finally, the squash optimization is also effective. By applying it on top of the other two optimizations, SpecFaaS attains average speedups of 5.0×, 4.4×, and 4.5× for FaaSChain, TrainTicket and Alibaba, respectively.

9.8.3 Throughput Improvement

We define effective throughput as the maximum number of requests per second that are serviced without QoS violation. A QoS violation occurs when the average response time of the requests is more than 2× the response time when the system only serves one single request. Table 9.3 shows the average effective throughput of each application suite for baseline and SpecFaaS, and the improvement obtained by SpecFaaS. From the table, we observe that SpecFaaS improves the effective throughput substantially. On average across application suites, SpecFaaS improves the throughput by 3.9×.

Table 9.3: Effective throughput in requests per second.

Application Suite	**Baseline**	**SpecFaaS**	**Improvement**
FaaSChain	118.3	485.0	4.1×
TrainTicket	90.3	346.0	3.8×
Alibaba	81.6	304.2	3.7×
Average	96.7	378.4	3.9×

9.8.4 Tail Latency

SpecFaaS also reduces tail latency significantly. Figure 9.13 shows the P99 (99th percentile) response time of SpecFaaS normalized to the baseline for the three application suites and different loads. On average across loads, SpecFaaS reduces the tail latency by 62%, 56% and 58% for FaaSChain, TrainTicket, and Alibaba, respectively. The average tail latency reduction across loads and applications is 58.7%.

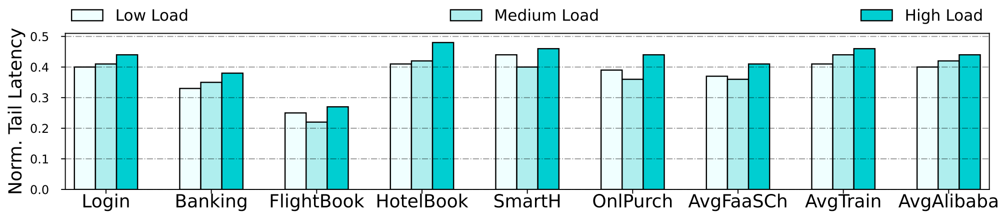


Figure 9.13: Tail latency of SpecFaaS normalized to the baseline for varying loads.

The reason for the high tail latency in baseline is the sequential allocation of resources and execution of functions, and the high transfer function overhead. The system waits for a function to complete before allocating resources for the following function. Instead, SpecFaaS allocates resources and executes functions early on.

9.8.5 Effect of Branch Prediction

We analyze the effect of branch prediction hit rates on overall performance. Figure 9.14 shows the speedup of SpecFaaS over the baseline for the FaaSChain applications and averaged across all loads, as we vary the branch prediction hit rates. We show data for 100%, 90%, 70%, and 50% hit rates.

As we decrease the hit rate from 100% (a perfect predictor) to 90% (the hit rate observed with Alibaba's trace [384]), the average speedup decreases by 5.7%. As the hit rate continues to decrease, the speedups decrease substantially. The impact of branch misprediction depends on where the branches are in the application workflow. If they are in the front-end, more work is likely to be discarded. Although not shown, the difference between the perfect

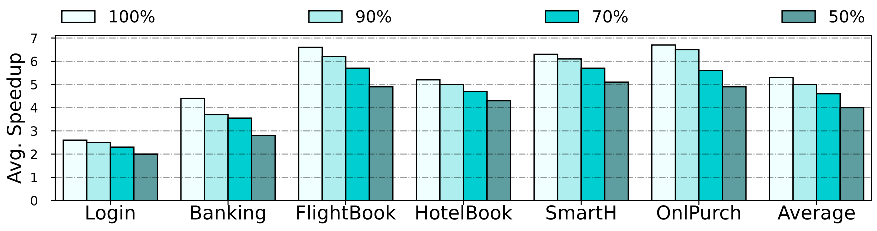


Figure 9.14: Speedup of SpecFaaS over the baseline for different branch prediction hit rates.

and imperfect predictors becomes larger when the load is high because misspeculation is relatively more costly in such environments.

9.8.6 Resource Utilization

SpecFaaS uses more resources than the baseline because some speculative work is squashed. In this section, we measure the contribution of the squashed work to the CPU utilization. We take the FaaSChain applications and vary the hit rate of speculation (i.e., how frequently control and data speculation succeed) from 100% to 50%. Table 9.4 shows the average CPU utilization for two speculative environments: *LazySquash* (where mis-speculated functions are allowed to complete their execution in the background before squashing) and SpecFaaS (where mis-speculated functions are immediately squashed). The CPU utilizations are normalized to the baseline, which is also shown in the table with a utilization of one. The last column of the table shows the average speedup of SpecFaaS over baseline. The data corresponds to the average of all loads.

Table 9.4: Normalized CPU utilization for FaaSChain for different speculation hit rates.

HitRate	**Baseline**	**LazySquash**	**SpecFaaS**	**Speedup**
100%	1	1	1	5.2×
90%	**1**	**1.09**	**1.03**	**5.0**×
70%	1	1.24	1.08	4.6×
50%	1	1.43	1.15	4.0×

The 90% row is bolded because it corresponds, approximately, to the average speculation hit rate attained by SpecFaaS. We see that, for this hit rate, SpecFaaS increases CPU utilization by only 3%, while achieving a 5× speedup. This is a tolerable cost, as the CPUs are typically busy only 60-80% of the time (Section 9.3). Moreover, squashing mis-speculated functions immediately saves substantial CPU cycles.

9.9 RELATED WORK

Most prior works address performance bottlenecks when executing a *single* function. They use lightweight containers specialized for serverless environments [85, 301], snapshotting techniques to reduce VM boot time [86, 188], and techniques that provide isolation for multi-tenancy [383]. Moreover, to reduce cold start latency, serverless platforms keep function containers alive for a grace period [63, 64, 65], and researchers propose advanced techniques [53, 187, 298, 375, 376]. SpecFaaS speeds-up multi-function applications.

Function workflows have recently gained great attention. However, prior work primarily focuses on addressing the cascading cold start via proactive and just-in-time resource provisioning [319, 320, 324, 381]. These solutions bring the next function to execute into warm state and thus, can only mitigate the cold start effect. SpecFaaS addresses both cold and warmed-up invocations through function overlap.

Netherite [382] is a distributed execution engine that allows a function to pass global updates to its successor function before the updates make it to global storage—a process they call speculation. SpecFaaS uses a similar form of data forwarding in the Data Buffer, plus speculation of function execution.

SAND[300] and Faastlane [193] reduce latency and improve resource efficiency of function chaining by executing all the functions of a workflow in the same container. The techniques are orthogonal to and can be combined with SpecFaaS.

Prior work proposed different local and remote in-memory caching techniques to bring data closer to processing units in FaaS systems [191, 194, 195, 328, 329]. SpecFaaS is orthogonal to these techniques and, additionally, uses caching to buffer speculative data.

Many works proposed memoization techniques for dataflows [404, 405, 406, 407, 408, 409, 410]. Contrary to SpecFaaS, these systems are not speculative: they do not need mechanisms to detect wrong memoization, squash incorrect executions, or restart from the correct state.

9.10 CONCLUSION

This chapter proposes to accelerate serverless environments with a novel approach based on *speculation*. Our proposal, *SpecFaaS*, executes functions in an application early, speculatively, before their control and data dependences are resolved. The execution of downstream functions is overlapped with that of upstream ones, substantially reducing the end-to-end execution time of applications. Our evaluation on OpenWhisk showed that SpecFaaS is very effective. On average, it attains an application speedup of 4.6× in a warmed-up environment. The application throughput increases by 3.9× and the tail latency reduces by 58.7%.

CHAPTER 10: Energy Management for Cloud-Native Services

10.1 INTRODUCTION

In previous chapters, we focused primarily on improving the performance of cloud-native services. However, one aspect that has barely been examined is the energy consumption of such environments. Specifically, substantial work has shown the unique performance issues of serverless computing, but how these issues translate into power/energy consumption is unknown. Hence, a framework for energy management in serverless systems is sorely needed. Advancing this area is critical, as serverless services are an increasing fraction of datacenter loads [353, 411, 412], and datacenters contribute substantially to the world energy consumption [413, 414] and carbon footprint [415, 416].

To address this shortcoming, this chapter starts by performing a thorough characterization of energy consumption in serverless environments. It shows that these environments pose a set of challenges not met by the existing energy management schemes designed for traditional datacenters with long-lived applications (e.g., [230, 235, 367, 417, 418, 419, 420, 421]).

Specifically, serverless functions are highly sensitive to core frequency—much more, e.g., than to the amount of memory resources. However, because cloud providers have no visibility into the performance requirements of user functions, they always execute functions at the highest frequency, potentially wasting substantial energy. Also, as applications are comprised of multiple functions with different properties, it is hard for users to reason about how different frequency settings for individual functions affect total performance and energy.

In addition, many functions spend a considerable amount of time waiting on remote function calls or accesses to remote storage, both implemented as Remote Procedure Calls (RPCs). Hence, cores need to frequently context switch between invocations of different functions. Further, these invocations may need different optimal frequencies, but changing core frequency from virtualized sandboxes incurs significant software overheads. Finally, providers densely pack many functions with various properties on a shared server, and the mix of such co-located functions changes rapidly. Thus, the per-core frequency setting that is optimal for the performance-energy trade-off at one time may quickly become suboptimal.

Based on the insights of our characterization, we propose *EcoFaaS*, the first energy-management framework for serverless environments. EcoFaaS takes energy efficiency as a first-class design principle to achieve an energy-optimized serverless environment. At the same time, EcoFaaS ensures the end-to-end performance target of serverless applications based on their Service Level Objectives (SLOs).

EcoFaaS achieves its goals through four main design principles. First, EcoFaaS is an SLO-driven serverless framework. In EcoFaaS, users specify the end-to-end SLO of the entire application—i.e., the maximum time that 99% of application invocations can take. EcoFaaS automatically splits the end-to-end application latency budget into per-function time budgets. Then, the functions, within their sandboxes, monitor the performance and comply with the assigned budget.

Second, EcoFaaS profiles and predicts the execution time and energy consumption of function invocations, taking into account their idle times and specific invocation inputs. Based on these predictions, EcoFaaS identifies the optimal core frequency to use for each function invocation. To take corrective actions on mispredictions, EcoFaaS continuously tracks the progress of invocations and updates the cores' frequencies as needed.

Third, to minimize the high overheads of changing core frequency, EcoFaaS dynamically splits the cores in the server into *Core Pools*. Within a pool, all cores run at the same frequency and are controlled by a single scheduler. Different pools have different core counts and use different frequencies. When EcoFaaS determines the best frequency to use for a given invocation, it assigns the invocation to the corresponding pool. In this way, EcoFaaS avoids changing frequency as much as possible while executing function invocations efficiently.

Finally, EcoFaaS makes the Core Pools *elastic*—it dynamically changes pool sizes and their frequencies. EcoFaaS supervises the system use and periodically recomputes the Core Pool set-up to adapt to the workload.

We implement EcoFaaS on top of two open-source serverless platforms OpenWhisk [57] and KNative [60]. We evaluate EcoFaaS with a diverse set of serverless applications [171, 227, 229]. Compared to state-of-the-art energy management systems, and averaging across various system loads, EcoFaaS reduces the total energy consumption of serverless clusters by 42%, while reducing tail latency by 34.8% and increasing throughput by 1.8×.

This chapter makes the following contributions:

- An analysis of the energy consumption of serverless systems.
- The design and implementation of EcoFaaS, the first energy-management framework for serverless environments.
- Evaluation of EcoFaaS on two open-source FaaS platforms.

10.2 TOWARD ENERGY-EFFICIENT SERVERLESS ENVIRONMENTS

To understand the unique energy-efficiency challenges in serverless environments, we characterize real-world serverless workloads [136, 226] and open-source applications [171, 302] on an Intel Haswell E5-2660 server. Section 10.6 describes our methodology, including the datasets and applications in detail. Based on our observations, we propose the following recommendations to attain highly energy-efficient serverless computing environments.

1. Serverless environments should be SLO-driven. We characterize the execution time and energy consumption of serverless functions at different core frequencies, ranging from 1.2GHz to 3GHz. We show their response time (Figure 10.1a) and energy consumption (Figure 10.1b). For example, consider the execution at 3GHz, and set the SLO to a few times the function execution time, as it is commonly done [283, 284]. We can see that many functions can be executed at substantially lower frequencies without violating such SLO, while saving substantial energy. For example, running CNNServe at 2GHz rather than at 3GHz increases its response time by 23% while reducing its energy consumption by 40%. As another example, running WebServe at 1.2GHz rather than at 3GHz increases response time by only 12% while reducing energy consumption by 47%.

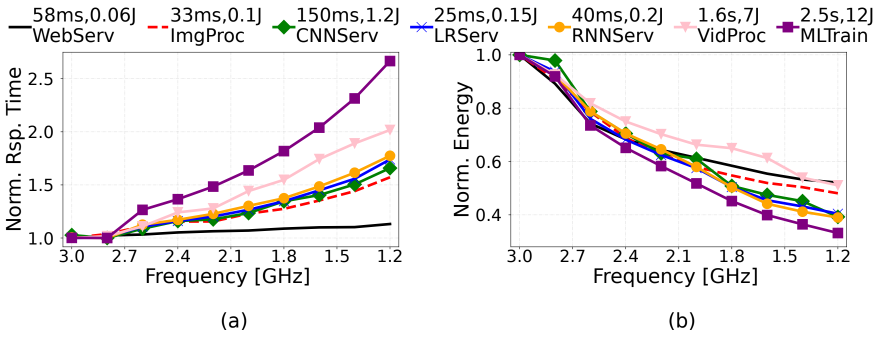


Figure 10.1: Normalized (a) response time, and (b) energy consumption of serverless functions with different core frequencies. The numbers above the legend show the execution time and energy consumption per function execution at 3GHz core frequency.

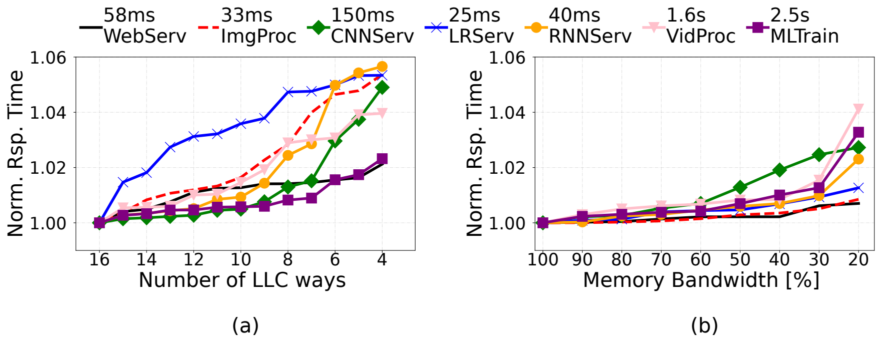


Figure 10.2: Normalized response time of serverless functions at 3GHz with different (a) number of LLC ways, and (b) memory bandwidth. The numbers above the legend show the function execution time with 16 LLC ways and 100% memory bandwidth.

Furthermore, functions are typically single-threaded and have relatively small memory footprints. Hence, they do not benefit from extra cores and benefit little from more memory resources. For example, Figure 10.2 shows the normalized response time when executing a function at the highest frequency of 3GHz with different (a) number of LLC ways and (b) percentage of memory bandwidth, both controlled by the `pqos` tool [208]. The latencies are normalized to the setup with the maximum amount of resources (16 LLC ways or 100% memory bandwidth). We see that, when using 4 LLC ways or 20% memory bandwidth, the response time increases *at most* by 6% and 4%, respectively. Hence, the core frequency is the main controllable knob that affects a function's performance and energy consumption.

Unfortunately, as cloud providers have no visibility into the performance requirements of user functions, they constantly operate at the highest frequencies, unnecessarily consuming energy. Furthermore, the energy-performance trade-offs are complex, as serverless applications are composed of multiple functions chained together into an application workflow. Different functions can have different latency and energy consumption profiles. Thus, it is hard for end users to reason about an execution plan that minimizes the energy consumption while satisfying the application's performance requirements.

We believe that, to substantially increase the energy efficiency of serverless environments, users need to specify the overall performance expectations of their applications, commonly expressed as SLOs. Then, the platform should automatically and transparently optimize the execution of each function of the application for overall minimal energy consumption while operating within the performance constraints.

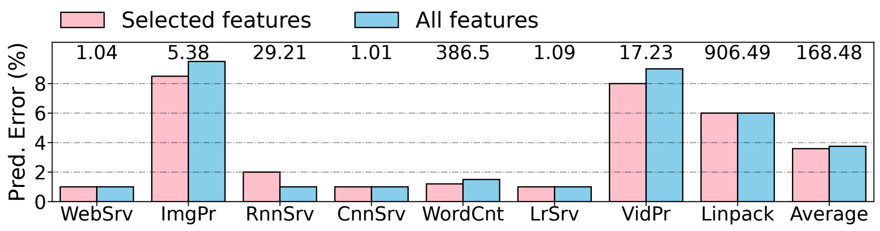


Figure 10.3: Prediction error of function execution time when functions are called with inputs not seen during training. The numbers on top of the bars are the ratio of longest to shortest execution time.

2. Serverless environments should be input-aware. The execution time of serverless functions can depend on the functions' inputs. To understand this effect, we execute our functions with various inputs from large open-source datasets [422, 423, 424, 425, 426]. After profiling over 100 open-source serverless functions [171, 201, 227, 228, 229], we observe that, typically, the execution time as a function of the inputs is either constant or can be approximated with simple polynomial functions. Consequently, like prior work [417], we extract high-level features from inputs, such as the size of the file, the duration of the video, or the resolution of the image, and train a simple three-layer neural network. After that, when a new input is received with certain values for the high-level features, the network estimates the execution time of the function with the new input.

In addition, in another set of experiments, we train the model with *all* the inputs of a function—not just those that appear to impact the execution time. With this approach, developers do not have to specify which are the important input features.

Figure 10.3 shows, for each function, the prediction error of our models, defined as $(|E - A|)/A$, where E is the estimated execution time and A the actual one. Each function has two bars, one for the model trained with selected features and one for the model trained with all the features. On top of the bars, we show the ratio of the longest to the shortest execution time for individual functions. We see that, although functions have a large range of execution times, their execution is highly predictable. On average, the error is only 3.6% when the model is trained with selected input features and only 3.8% when it is trained with all the input features. Therefore, we train our model with all the features and keep the prediction accuracy high while reducing the burden on developers.

3. Serverless environments should exploit the substantial idle time within function invocations. Function execution time is short, ranging from a millisecond to a few seconds [63, 299]. Even during such a short time, functions are mostly idle, waiting for RPC responses from remote functions or remote storage. In our applications, functions accessing data from remote storage commonly spend 70% of their execution time idling. Thus, to improve resource utilization and keep cores busy, advanced serverless systems perform a context switch when a core stalls. As a result, context switches are typically as frequent as one every few hundreds of μs [35, 189, 196].

However, state-of-the-art energy-management frameworks for traditional applications are designed with a run-to-completion model [417, 418, 419, 421]. As an example, Gemini [418] maintains a FIFO request queue and detects the arrival of a request whose deadline will be missed if executing at the current frequency. Then, it executes all queued requests, one at a time, at a higher frequency to meet the critical request's deadline. The system relies on the run-to-completion model to predict the queuing time for each incoming request. This approach works well for applications that are not I/O bound. However, it can be detrimental for the performance and energy efficiency of serverless environments with many I/O-bound functions. With the run-to-completion model, as the request queue builds up, the invocations execute at higher frequencies, while wasting the time that functions spend being blocked.

We measure the energy consumption needed to meet the SLOs in two environments: *Run-To-Completion* and *Context-Switch-on-Idle*. The latter uses the idle time of an invocation

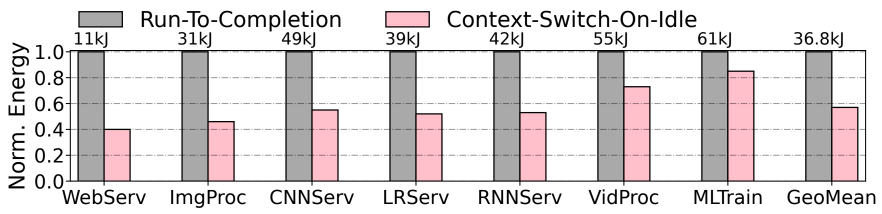


Figure 10.4: Normalized total energy consumption of functions averaged across loads when executed under *Run-To-Completion* or *Context-Switch-On-Idle*. The numbers on top of the bars show the total energy consumption for 10-minute runs in kJ.

to run another ready-to-run invocation. We execute our functions at different loads, varying the request inter-arrival time with a Poisson distribution. We set the SLO of a function to be 5× its execution time on an unloaded system.

Figure 10.4 shows the total energy consumption averaged across all loads when running each function in either environment. Context-Switch-on-Idle allows for more invocations to execute at lower frequencies. Hence, as we see in the figure, it reduces the energy consumption by 42.3%. As the idle time within invocations increases and as the load gets higher, the savings become more substantial. It can be shown that Context-Switch-on-Idle also improves the performance of serverless functions, especially when the load is high. For example, on average across all functions in high load, the average and tail latencies are reduced by 48.2% and 67.4%, respectively.

4. Serverless environments should minimize the number of core frequency changes. To exploit the idle time within an invocation, when a core becomes idle, it should context switch to another function invocation. This new invocation may have a different optimal frequency. Thus, at every context switch, the system might need to change the core's frequency.

Such changes are expensive. Recall that a server can be running serverless functions from different users, and they are isolated from each other in different VMs [85, 136] or containers [57, 60, 427]. To change the core's frequency, sandboxed serverless functions need to communicate with the host and cross the OS kernel boundary (i.e., switch between user and kernel spaces). Consequently, even though the hardware overheads of changing the core frequency are only a few tens of microseconds, our results show that, in Linux-based systems with an ACPI frequency driver, the overhead of changing core frequencies from userspace processes or containers is 10-20ms. This overhead is of the same order of magnitude as the execution time of a serverless function.

To see this effect, we execute our functions in two environments. First, in *ConstFreq*, we keep the frequency constant at 2.5GHz throughout the whole run. Second, in *SwitchFreq*, we keep setting the frequency to 2.5GHz at every context switch. Thus, the two environ-

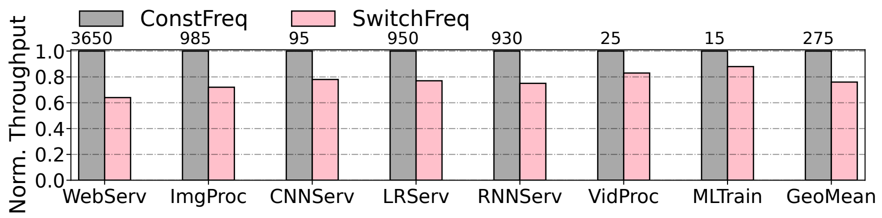


Figure 10.5: Normalized function throughput (higher is better) when executed in *ConstFreq* and *SwitchFreq*. The numbers on top of the bars show the *ConstFreq* throughput in RPS.

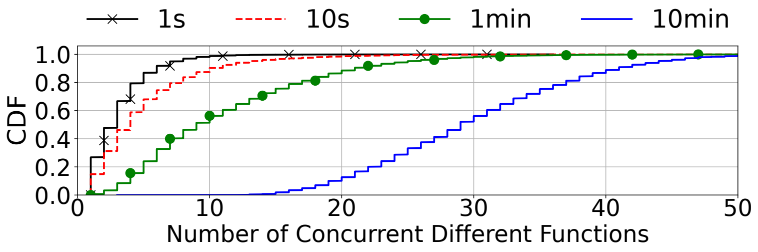


Figure 10.6: CDF of the number of *different* functions executed in a small cluster within 1s, 10s, 1min, and 10min in Azure Functions.

ments execute under the same conditions; the only difference is the overhead of invoking the kernel to set the core's frequency in the second case. Figure 10.5 shows the throughput of the functions in the two environments. We see that the overhead of changing the core's frequency from the virtualized sandboxes can significantly reduce system throughput. For short functions like WebServ, the throughput losses are higher. On average across all functions, *SwitchFreq* reduces the throughput of *ConstFreq* by 24.1%. Hence, for energy efficiency, one should minimize frequency changes.

5. Serverless environments should dynamically adapt to workload changes. An intuitive way to minimize the number of core frequency changes is to dedicate some cores to each frequency level. Functions with similar frequency needs can be grouped together into classes and better utilize the shared cores while minimizing core frequency changes. The key challenge is to address FaaS workload dynamics: the workload is ephemeral, functions are frequently loaded/unloaded from memory, and their loads continuously fluctuate [63, 299].

We analyze open-source production-level traces from Azure Functions [136] to understand the co-location of different functions. Figure 10.6 shows the CDF of the number of different functions executed in a small cluster within 1 second, 10 seconds, 1 minute, and 10 minutes. We see that, within a second, the system executes on average 3 different functions, but it may execute up to 36 different functions. Within 10 seconds, it may execute up to 52 different functions. This is in contrast to the traditional VM-based cloud environments, where only a few long-lived VMs share the cluster for long durations. This shows that the optimal allotment of cores to function classes at a given time can rapidly become suboptimal.

10.3 LIMITATIONS OF CURRENT ART

Researchers have proposed energy-management frameworks for systems running long-lived applications. For example, Pegasus [367] achieves energy proportionality by setting the power limit of the entire server to meet, but not exceed, the SLO of the application running on that server. EETL [420] places requests on slow cores, and reschedules them to fast cores when it predicts that the SLO will not be met. Adrenaline [421] predicts long requests before scheduling them on cores and boosts their frequency from the beginning of their execution. ReTail [417], Rubik [419] and Gemini [418] take one step further and predict the optimal frequency for every request individually. NCAP [428] and NMAP [429] proactively transition a processor to an appropriate performance or sleep state based on the rate of received and transmitted network packets containing latency critical requests. These schemes are effective for latency-critical monolithic applications or backend services. Unfortunately, they are not a good fit for serverless environments.

First, these schemes target a single service or a monolithic application, while serverless applications are workflows of many functions glued together. As these functions may execute on different machines, have different performance and energy profiles, and execute in highly dynamic environments, it is non-trivial to specify individual function deadlines, while keeping end-to-end application target response time satisfied.

Second, these schemes rely on a *run-to-completion* model when determining the per-application or per-request optimal frequency. They assume that the running request has to finish before a new request can be scheduled for execution [417, 418, 419]. The run-to-completion model is a reasonable assumption for applications that keep their cores busy throughout most of the execution. However, it is energy inefficient for the typical serverless function, which is mostly idle waiting on I/O. Researchers have explored techniques to efficiently coordinate sleep states with frequency states during idle periods [430, 431]. However, these works assume server-wide idleness. In a serverless setup, while a single invocation experiences idle time, there are other invocations waiting in the queue ready to execute. Thus, if the core refuses to execute waiting invocations and goes to sleep, it will need to execute these invocations later at a higher frequency, to compensate for their longer queuing.

Third, these schemes assume a negligible cost for frequency changes. The assumption is true for long-lived applications with dedicated cores and infrequent context switches. However, in serverless environments: *(i)* cores frequently context switch between invocations of the same or different functions, and *(ii)* functions run in virtualized sandboxes, unable to directly change the core frequency without contacting the host.

Lastly, these schemes are mainly designed for server machines running a single long-lived application [417, 418], a latency-critical application co-located with a batch application [419], or a few co-located applications using exclusively partitioned resources [230, 235]. Serverless environments have a very different resource model: many functions with various performance requirements can concurrently execute on the same machine, and the machine is typically overprovisioned with more functions than available cores [85].

On the other hand, state-of-the-art serverless systems are not optimized for energy efficiency. They improve performance by *(i)* minimizing cold start overheads [53, 63, 64, 86, 188, 301, 325], *(ii)* improving resource utilization [35, 189, 193, 300, 319, 320], *(iii)* reducing the cost of remote storage access [35, 191, 194, 335], and *(iv)* proactively scheduling or executing functions of an application's chain [37, 324, 382]. They do not consider energy consumption in their designs.

10.4 ECOFAAS OVERVIEW

Based on our insights of Section 10.2, this section presents *EcoFaaS*, the first energy-management framework for serverless environments. EcoFaaS is based on four main ideas.

1. EcoFaaS is driven by SLO metrics. In existing serverless platforms, to meet the expected performance, end users specify the number of cores or the amount of memory devoted to the function [432, 433, 434, 435]. Unfortunately, it is not easy for users to reason about the exact impact of these resources on the performance and energy efficiency of functions [436]. Furthermore, applications are composed of multiple functions. Thus, specifying these resources for individual functions makes the end-to-end outcome even more opaque. As a result, cloud providers simply execute all functions with the requested cores or memory at the highest core frequency.

EcoFaaS argues for a different environment, where end users only specify the end-to-end SLO of the entire application—i.e., the maximum time that 99% of the application invocations can take. This approach enables the end user to provide an accurate specification of what they need, without increasing the user's specification burden. As importantly, it enables cloud providers to optimize execution for energy efficiency. Specifically, EcoFaaS automatically splits the SLO of an application into per-function time budgets, and then dynamically sets the frequency of cores executing individual functions based on the current system conditions. The goal is to execute functions at their most energy-efficient frequencies while satisfying the SLO of the end-to-end application.

2. EcoFaaS profiles and predicts the execution time and energy of function invocations. During execution, EcoFaaS profiles the execution time and energy consumption of functions at different core frequencies. It also takes into account variations due to different function inputs. The execution time is broken down into: 1) execution on a core (T_{Run}), 2) I/O blocking due to RPCs (T_{Block}), and 3) waiting in a queue to be processed (T_{Queue}). Note that T_{Block} and T_{Queue} are substantial and often comparable to T_{Run}. EcoFaaS uses values of profiled T_{Run} and T_{Block}, and values of predicted T_{Queue} based on current system conditions, to select the core frequency for each function that leads to the most energy-efficient application execution while satisfying the SLO of the entire application.

3. EcoFaaS splits cores into frequency classes. For highest energy efficiency, different functions and even different invocations of the same function may need to run at different frequencies. However, functions typically have millisecond-level execution times and include blocking events that cause context switches. Since changing core frequency at every context switch is inefficient, EcoFaaS avoids changing frequency as much as possible. It keeps *Core Pools* running at different frequency levels and tries to assign the execution of individual function invocations to the pool that has a frequency equal or slightly higher than the invocation's optimal frequency. Each pool is controlled by a *Frequency Pool Scheduler* (FPS).

4. EcoFaaS changes pools and pool frequencies dynamically. In serverless environments, the functions being invoked and their popularity change over time. A partition of cores into frequency pools that was optimal at a given time may quickly become suboptimal. Hence, EcoFaaS supervises the system use and periodically recomputes the assignment of cores to pools and the frequency of each pool. The goal is to execute the functions in the most energy-efficient manner while capturing the dynamics of the workloads executing.

10.5 ECOFAAS DESIGN

Figure 10.7 shows the organization of an EcoFaaS serverless platform. Like existing schemes [57, 60, 353], EcoFaaS has Frontend and Load Balancer modules, and per-node Node Controllers. EcoFaaS adds a *Workflow Controller* per application and, inside each node, a *Function Dispatcher* per function container, *Core Pools*, and a *Frequency Pool Scheduler* per pool. Except for the core pools, all these structures are software structures.

The Workflow Controller maintains the SLO of the application and calculates the execution deadline for each function of the application to attain maximum energy efficiency. Then, it sends the computed deadlines for each function to the corresponding Function Dispatchers.

A Function Dispatcher manages a function container. It receives requests for its function, forks a *Handler* process to execute each request, and selects the optimal frequency to run the requests based on the function profile and the execution deadline. Additionally, each

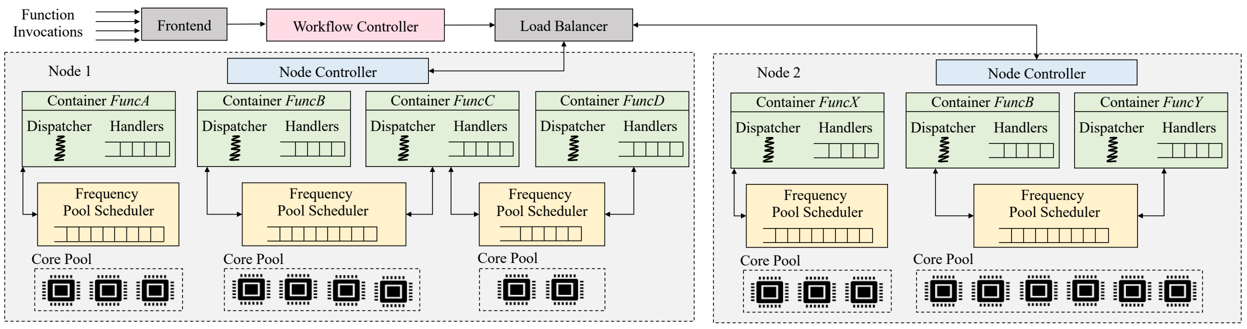


Figure 10.7: Overview of an EcoFaaS serverless platform.

Function Dispatcher profiles the execution of the invocations of its function and, every T_{update}, sends the profile to the corresponding Workflow Controller to update the optimal per-function deadlines.

Based on the chosen frequency, the Function Dispatcher registers the function invocation to execute in a specific Core Pool. The cores in that pool run at a selected frequency and are managed by a *user-space* Frequency Pool Scheduler (FPS). When a core in that pool becomes available, the function invocation is scheduled for execution; when the execution blocks due to an RPC, another invocation registered with the same pool (from the same or a different function) is scheduled.

The Node Controller periodically collects runtime metrics from every FPS, including the number of queued and served invocations, and invocations that could have been executed at lower frequencies if an appropriate core pool had been available. Based on this information, the Node Controller adjusts the allocation of cores to pools and the frequency of each pool.

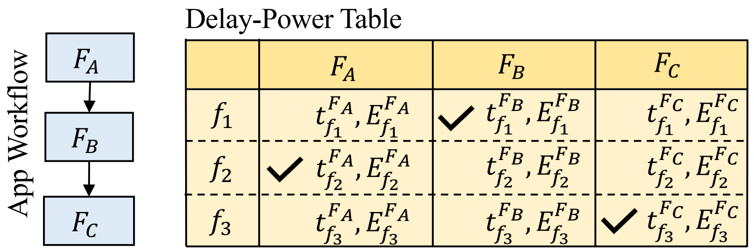
10.5.1 SLO-Aware Workflow Controller

When a user submits an application invocation to EcoFaaS, they specify the end-to-end SLO for the application. Then, EcoFaaS automatically and transparently splits the application's SLO into the optimal per-function deadlines. The response time and energy consumption of different functions in the application may have different sensitivity to core frequency. The optimal execution is the one that minimizes the combined energy consumption of all the functions of the application while finishing the application within the SLO.

To achieve this, the Workflow Controller of an application uses profile data regularly provided by the Function Dispatchers of the constituting functions. Such information is kept in a per-application software structure called *Delay-Power Table* (DPT). The DPT has an entry for each function F_i of the application running at each of the possible frequencies f_j. The entry contains the predicted execution time $t_{f_j}^{F_i}$ of the function (equal to $T_{Run} + T_{Block}$

$+ T_{Queue}$) and the predicted energy consumption $E^{F_i}_{f_j}$. Then, the Workflow Controller uses Mixed-Integer Linear Programming (MILP) [437] to pick the frequency for each function that minimizes $\sum E^{F_i}_{f_j}$ under the constraint $\sum t^{F_i}_{f_j} \leq SLO$. MILP is a suitable technique as both objective function and constraint are linear relations with FP numbers.

Figure 10.8 shows the DPT for an application composed of functions F_A, F_B, and F_C, and supporting frequencies f_1, f_2, and f_3. The Workflow Controller determines the optimal frequency of each individual function and the resulting deadline of each function. For example, Figure 10.8 shows with tick marks the optimal frequencies of each function. Based on the figure, F_B's optimal frequency is f_1, and F_B's deadline is $t^{F_A}_{f_2} + t^{F_B}_{f_1}$.



	F_A	F_B	F_C
f_1	$t^{F_A}_{f_1}, E^{F_A}_{f_1}$	✓ $t^{F_B}_{f_1}, E^{F_B}_{f_1}$	$t^{F_C}_{f_1}, E^{F_C}_{f_1}$
f_2	✓ $t^{F_A}_{f_2}, E^{F_A}_{f_2}$	$t^{F_B}_{f_2}, E^{F_B}_{f_2}$	$t^{F_C}_{f_2}, E^{F_C}_{f_2}$
f_3	$t^{F_A}_{f_3}, E^{F_A}_{f_3}$	$t^{F_B}_{f_3}, E^{F_B}_{f_3}$	✓ $t^{F_C}_{f_3}, E^{F_C}_{f_3}$

Figure 10.8: Organization of the Delay-Power Table for one application. Check marks show the chosen frequencies for each function.

This approach handles cases where a function invokes multiple parallel children functions. The workflow controller assigns a deadline to the group of parallel functions based on the slowest function in the group.

10.5.2 Energy-Aware Function Dispatcher

When a container receives a function invocation, the dispatcher in the container creates a handler to execute the invocation. To execute the invocation in the most energy-efficient manner, the dispatcher uses: (i) information provided by the Workflow Controller that is included in the function invocation message and (ii) profiling information on the function that the dispatcher has gathered from past executions.

The function invocation message includes the deadline for executing the function in absolute time, as computed by the Workflow Controller. It does not include the recommended execution frequency. The reason is that the current execution environment may be different from the one used by the Workflow Controller to estimate the optimal computation.

To see why, consider our running example of the application with functions F_A, F_B, and F_C, and assume that we are at a point when we want to run F_B. Figure 10.9a shows the timeline of the optimal execution computed by the Workload Controller, where F_A's deadline is t_A, F_B's is t_B, and F_C's is t_C. In practice, assume that the execution of F_A was faster (e.g., the load was low and there was little queuing) and took only t'_A (Figure 10.9b). Now, given F_B's unchanged deadline, the dispatcher for F_B can pick a lower frequency.

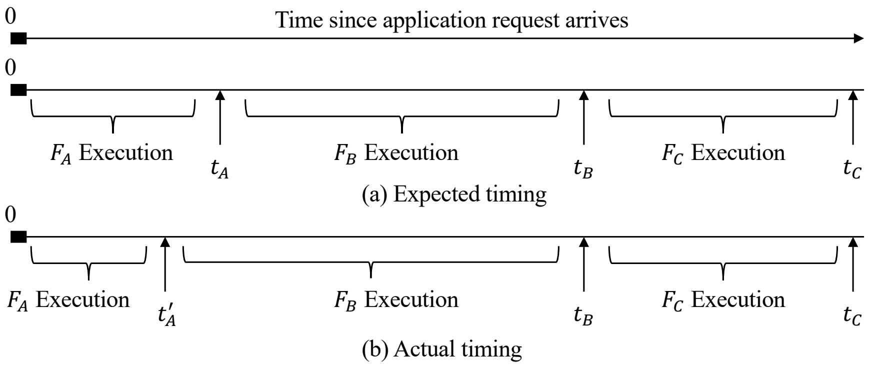
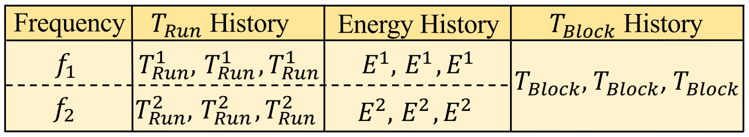


Figure 10.9: Timeline of a three-function application execution.

To pick the best frequency at which to run F_B, the dispatcher examines the history of past executions of F_B. The dispatcher has been profiling prior executions of the function and stored the profile in a software structure for that function called *History Table*. Figure 10.10 shows the design of the table. It contains T_{Run}, T_{Block}, and *Energy* for the few most recent executions of the function. Recall that the execution time of the function is $T_{Run} + T_{Block} + T_{Queue}$, while *Energy* is the energy consumed during T_{Run}. Since T_{Run} depends on the frequency, both T_{Run} and *Energy* have entries for multiple frequencies. This is not the case for T_{Block}. The History Table is continuously updated: every time a handler executes the function, it measures and saves T_{Run}, T_{Block}, and *Energy*.



Frequency	T_{Run} History	Energy History	T_{Block} History
f_1	$T^1_{Run}, T^1_{Run}, T^1_{Run}$	E^1, E^1, E^1	$T_{Block}, T_{Block}, T_{Block}$
f_2	$T^2_{Run}, T^2_{Run}, T^2_{Run}$	E^2, E^2, E^2	

Figure 10.10: The History Table in a dispatcher.

With this information, the dispatcher can estimate the expected T_{Run}, T_{Block}, and *Energy* for different frequencies. We propose two different approaches. The first, simpler one, is for the dispatcher to use the exponentially weighted moving average ($EWMA$) [438] of these three parameters. EWMA assigns higher weights to more recent measurements, and uses adaptive smoothing with the Holt-Winters method [439] to dynamically adjust a parameter α based on the changes in the system state.

A better, more advanced approach is to also record in the History Table the inputs of the function for each invocation, and then use a machine learning model to estimate the expected T_{Run}, T_{Block}, and *Energy* for different frequencies. We discuss this design in Section 10.5.5.

After the dispatcher estimates the expected T_{Run}, T_{Block}, and *Energy* for each frequency, it estimates T_{Queue}. For this, the dispatcher examines the number of waiting jobs in the queues of the Core Pools in the server. This is enough to estimate T_{Queue}, given that we use: 1) FIFO queueing, 2) pre-emption of a younger job when an older job is ready to run, and 3) a user-space scheduler whose scheduling overhead is largely negligible. Finally, with all this information, the dispatcher picks the frequency with the lowest *Energy* that still satisfies $t'_A + T_{Run} + T_{Block} + T_{Queue} \leq t_B$, where t_B is the deadline to complete F_B according to the Workload Controller.

When a function is unloaded from the system, its History Table is saved as part of the function's context. It is then used as a starting point when a new instance of the function is created in the future, hence avoiding cold start effects. When there is no prior knowledge about a function's execution, the dispatcher picks the highest possible frequency.

10.5.3 Core Pools and Frequency Pool Schedulers (FPS)

As shown in Section 10.2-3, cores in serverless environments experience context switches as frequently as one every few hundreds of μs. If, in EcoFaaS' quest to execute invocations at optimal frequencies, EcoFaaS had to change the frequency of a core at every context switch, the overhead would be intolerable. Indeed, changing the frequency takes about 10 μs without software overhead and often 1000x more with software overhead (Section 10.2-4). Therefore, EcoFaaS faces a tradeoff between fine-tuning the core frequency to attain higher energy efficiency and not doing it to reduce overhead.

In practice, serverless workloads are often bursty: the same function is invoked many times in a short period. When this happens, it offers the ability to reuse the core frequency across function invocations. Also, different functions co-located on the same server may have the same frequency requirements. For these reasons, and because cores typically offer a limited number of frequency levels, EcoFaaS organizes the cores in a server into dynamic *Core Pools*. The cores in a pool run at the same frequency, while different pools run at different frequencies. Core pools dynamically change the number of cores and the frequency based on application demands.

When a Function Dispatcher has estimated that a function should optimally run at a given frequency, the dispatcher searches for and picks a Core Pool that is running at the same or slightly higher frequency. Since different invocations of the same function may have different optimal frequencies, the Function Dispatcher can register different invocations of the function with different Core Pools.

A Core Pool is managed by a *Frequency Pool Scheduler* (FPS) that schedules function invocations on the cores. The FPS is user-level, uses FIFO, allows a ready-to-run older job to immediately preempt a younger job, and has negligible scheduling overhead.

Recall that EcoFaaS estimates T_{Queue} before deciding the optimal core frequency to use to run a function. To make it easy to do so, each FPS maintains *Estimated Wait Time (EWT)* counters that give the wait time for executions at the current and all higher frequency levels. An EWT counter contains the sum of the expected T_{Run} of all the queued and running jobs. When a new invocation is added to the queue, the EWT counter is incremented by the invocation's estimated T_{Run}; when an invocation completes, the EWT counter is decremented by the invocation's T_{Run}. An EWT counter divided by the number of cores in the Core Pool estimates T_{Queue}.

With FPSs, user VMs or containers use a clean interface to register invocations for execution at different frequencies—while maintaining their black-box property. Cloud providers do not have, and do not need, visibility into the code of users' functions. Instead, Function Dispatchers perform all the profiling, monitoring, and optimal frequency calculations inside the sandboxed VMs or containers. Then, the Function Dispatchers register the invocations with the chosen Core Pool, and give to the pool's FPS only the estimated time that the invocation will spend running on a core.

10.5.4 Elastic Core Pools

Due to the dynamic nature of serverless environments, the number of cores in each pool and their frequency is unlikely to be optimal or even usable for more than a short time. Consequently, EcoFaaS dynamically changes the Core Pools.

Consider first the case when a function invocation I_0 may be unable to meet its deadline in any of the Core Pools. In this case, the corresponding dispatcher checks if the deadline can be met in one of the pools by temporarily increasing the frequency. Specifically, the dispatcher first looks for a pool where I_0's deadline is met if the existing frequency is kept for the currently-queued jobs, and the frequency is temporarily boosted only when it comes to I_0's turn to execute. If such pool exists, this strategy is used. Otherwise, the dispatcher looks for a pool where I_0's deadline is met if the frequency is temporarily raised for both the currently queued jobs and I_0. If such pool exists, this second strategy is used. Otherwise, I_0's deadline will likely be missed, but the dispatcher still picks the queue with the shortest $T^{'}_{Queue}$ (defined as the predicted queuing time at the highest possible frequency) and the system increases the frequency of all the queued jobs and I_0 to the maximum possible value.

Consider now the case when the workload changes substantially. For example, initially, there are many invocations choosing frequency f_1 and few choosing f_2; then, suddenly, frequency f_2 becomes popular while f_1 is rarely needed.

EcoFaaS tackles this challenge by periodically reassigning cores among pools and changing the frequency of pools. Specifically, during each $T_{refresh}$ interval, the FPS in each pool records the number of served invocations, the average waiting time in the queue, the number of invocations that could have been executed at a lower frequency (but found no appropriate pool), and the number of invocations that required temporary increases in frequency to meet deadlines. Then, at the end of the interval, all FPSs send their collected information to the Node Controller.

The Node Controller uses this information to apportion cores to pools, and to set the frequency of the pools for the next interval. The process is as follows. Each pool i is assigned a weight W_i. Pools that served more requests or had longer waiting times receive higher weights. After that, the Node Controller assigns a number of cores N_i out of the total N_{total} to pool i based on $N_i = \frac{W_i * N_{total}}{\sum W_i}$. Then, those pools that often had to temporarily increase their frequency to meet invocations' deadlines are assigned the next higher frequency level, while pools that often took invocations that could have executed at a lower frequency are assigned the next lower frequency level.

10.5.5 Improving the Robustness of EcoFaaS

We improve the robustness of EcoFaaS in three ways.

Cold starts. When an invocation experiences a cold start (no warm VM/container), the system starts-up the VM/container and initializes the function before executing the invocation. Despite recent work that minimizes this cost [35, 53, 63, 65, 189], the time of a cold start is of the same order of magnitude as function execution time at best. If the cold start is on the critical path, EcoFaaS executes both cold start and the function's handler at a high core frequency, substantially degrading energy efficiency. Otherwise, the Workflow Controller tries to execute the cold start early, off the critical path, and at lower frequencies. The process is as follows.

The controller checks if any function in the application has no available container in the cluster and, hence, will require a cold start. If such a function exists, the controller *prewarms* the container at a lower frequency, in the background, while the predecessor functions in the application's chain are being executed. If there are multiple functions that will experience cold starts, their containers are prewarmed in parallel.

To determine the frequency to use in cold starts of the function, EcoFaaS prewarms the container at different frequencies in different cold starts of the function, and populates the Delay-Power Table of the function. After that, in subsequent cold starts of the function, EcoFaaS picks the minimal core frequency that can complete the cold start within the sum of the predecessor functions' deadlines. For example, the cold start of F_c in Figure 10.8 has to be completed in $t_{f_2}^{F_A} + t_{f_1}^{F_B}$. In this way, when F_c executes, it will not suffer a cold start.

Function Input Sensitivity. As indicated in Section 10.2-2, the execution time of some functions varies with the function inputs, but these variations are highly predictable. Hence, similar to prior work [417], EcoFaaS uses an ML model to estimate the expected T_{Run}, T_{Block}, and *Energy* of a function invocation. As indicated in Section 10.5.2, every time that a handler executes the function, it measures and saves T_{Run}, T_{Block}, *Energy*, and the function inputs in the corresponding History Table. Then, when a new invocation of the function is received (with potentially new inputs), the Function Dispatcher uses the ML model to estimate T_{Run}, T_{Block}, and *Energy* for different frequencies. The model we use is lightweight, has three fully connected (linear) layers and ReLU activations, and takes the features of all the inputs of the function, to eliminate any annotation burden on developers. The model is trained online using live traffic of function invocations. As indicated in Section 10.2-2, its prediction error is less than 4%, while its overhead is only a few tens of μs.

Heterogeneous Servers. While datacenters strongly favor machine homogeneity [440, 441], many of them have heterogeneous servers. In this case, the Delay-Power Table (DPT) generated for one server type (e.g., Intel Haswell [442]) cannot be directly reused by another server type (e.g., Intel Skylake [443]). Hence, EcoFaaS needs to profile the functions on all server types. Note that, on average, only a few tens of application invocations are needed to populate the DPT of the functions in the application for each type of server. During most of these invocations, the application maintains good performance, although its energy efficiency is suboptimal. Since applications are invoked thousands of times a day [226, 299], the impact of this profiling period is largely negligible.

However, to minimize this inefficiency, EcoFaaS uses transfer ML techniques. Given the profiles (execution time and energy consumption) of functions on machine A and a small subset of function profiles on machine B, transfer ML generates a model that predicts the profiles of the rest of the functions on B. In EcoFaaS, we train and test a simple Linear Regression model that performs transfer ML from Intel Haswell [442] servers to Broadwell [444] and Skylake [443] servers. By using only 1/4 of the samples from the last two machines, the model achieves an accuracy of 93.1%.

10.6 METHODOLOGY

Evaluation environment. We evaluate EcoFaaS in the state-of-the-art MXFaaS [35] serverless platform, on top of OpenWhisk [57] and KNative [60]. Here, we discuss only the results with OpenWhisk, as the results with KNative are similar. In our experiments, we vary the number of servers in a cluster from 5 to 20. Each server is an Intel Haswell E5-2660 v3 with 20 cores organized in two sockets, with 160GB of DRAM, a 50MB LLC, and running Ubuntu 22.04.2 LTS.

We use the ACPI frequency driver with the "userspace" governor to allow 7 user-defined frequency settings ranging from 1.2GHz to 3.0GHz in 0.3GHz increments. We measure the energy consumption of each server with CPU Energy Meter [445]. This includes the energy consumed by the package and DRAM for both sockets running the whole software stack. Similar to prior research [446, 447, 448, 449, 450], we use power modeling to apportion the overall socket power to individual cores. The model takes into account the core's frequency, the number of active core cycles (stall cycles are excluded), and a few performance counters.

For the configurable parameters in EcoFaaS, we perform sensitivity studies and pick the following values: *(i)* keep the last 100 invocations in History Tables, *(ii)* update the Workflow Controller's Delay-Power Table every 5s (T_{update}), and *(iii)* update the pools' sizes and frequencies every 2s ($T_{refresh}$).

Evaluated functions and applications. We use functions from FunctionBench [171] (widely used for serverless research) [53, 64, 136, 312, 313, 314]. The functions include ML training and model serving, image/video processing, and web services. In addition, we use a set of real-world serverless applications from AWS Samples [227], SeverlessBench [228] and vSwarm [201]. The applications include ML workflow, data analytics, online banking and booking, and video streaming. The evaluated benchmarks are summarized in the Table 10.1. We use Azure Blob Storage [334] as the storage service for all the functions and invoke functions with inputs from open-source datasets [422, 423, 424, 425, 426]. Like prior work [283, 284], we set the SLO of an application to 5$\times$ the application's warm latency on an unloaded system at the highest frequency.

We first evaluate these benchmarks using the invocation patterns from the open-source production-level traces of Azure Functions [136]. Then, we vary the load using a Poisson distribution to model the request inter-arrival time [136, 196, 300, 319, 320, 321, 322]. We generate low, medium and high loads corresponding to CPU utilizations of about 25%, 50%, and 70%, which are representative [152, 316, 317, 318].

Baselines. We compare EcoFaaS to two advanced baseline frameworks: *Baseline* and *Baseline+PowerCtrl*. *Baseline* is the state-of-the art MXFaaS [35] serverless platform, as

Table 10.1: Serverless benchmarks used in the evaluation.

Benchmark	**Description**
Standalone Functions (all from FunctionBench [171]	
WebServ	Processing JSON file fetched from the storage
ImgProc	Image processing: Resize image
CNNServ	ML model serving: CNN-based image classification
LRServ	ML model serving: Logistic regression
RNNServ	ML model serving: RNN-based word generation
VidProc	Video processing: Apply gray-scale effect
MLTrain	ML model training: Logistic regression
Serverless Applications	
MLTune [451]	Tuning an ML model (6 functions)
DataAn [228]	Wage-data analysis workload (8 functions)
eBank [227]	Withdraw money from an account (6 functions)
eBook [201]	A hotel reservation service (7 functions)
VidAn [201]	A video analysis system (3 functions)

described in Chapter 7.

Baseline+PowerCtrl is *Baseline* plus a state-of-the-art energy-management framework for long-lived applications based on Gemini [418]. This framework saves energy by setting the frequency for a function invocation based on the function's deadline. The framework assumes a run-to-completion model and, on a context switch, it changes the core's frequency only if the predicted best frequency of the next request differs from the current core frequency. *Baseline+PowerCtrl* is an upper-bound of schemes such as Gemini because it predicts an invocation's execution time at a given frequency with 100% accuracy. For applications with multiple functions, *Baseline+PowerCtrl* ditributes the application's SLO to functions in proportion to their execution time at the highest frequency [319, 320].

10.7 EVALUATION RESULTS

10.7.1 Energy Savings with Real-World Invocation Patterns

We use traces from Azure Functions [136] to mimic real-world invocation patterns while executing the functions from our benchmark suite. The traces capture the typical bursty behavior of serverless workloads. During a 10-second window, 119 different functions are invoked. On average, a function executes for 50 ms and is invoked 14 times during the window. However, about 10% of the functions are invoked more than 113 times, and there is at least one time when 33 invocations of the same function are executing concurrently.

From this trace, we select the 12 most popular functions, which account for over 76% of all invocations. We assign 12 of our benchmarks to these popular functions. We run the

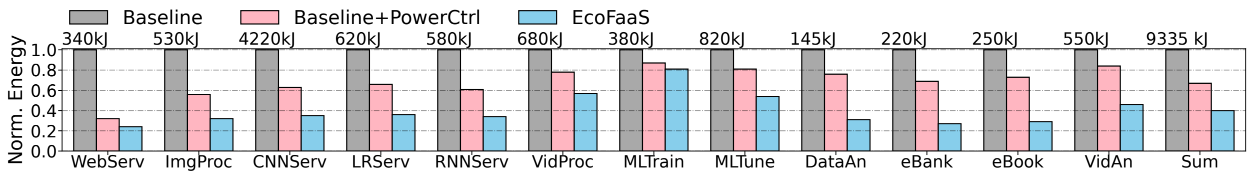


Figure 10.11: Normalized energy consumption of *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* with real-world invocation traces. The numbers on top of the Baseline bars show the absolute energy consumption of Baseline for the 6-hour run in all 5 servers.

traces for 6 hours on a cluster of 5 servers. On average, each server receives 50-100 RPS.

Figure 10.11 shows the normalized and absolute energy consumption of Baseline, Baseline+PowerCtrl, and EcoFaaS for each individual benchmark and for the sum of all the benchmarks. The bars in the figure show the total energy consumed by all the invocations of a given function. We see that Baseline+PowerCtrl and EcoFaaS reduce the total energy consumption of the benchmarks by 33% and 60%, respectively, over Baseline.

EcoFaaS curbs the energy consumption of all the evaluated benchmarks. Compared to Baseline, the benefits are higher for benchmarks that are sensitive to core frequency; compared to Baseline+PowerCtrl, the benefits are higher for benchmarks with significant idle time, such as ImgProc and RNNServ. In addition, in applications with many functions such as eBank and eBook, the Workflow Controller in EcoFaaS assigns optimized per-function deadlines. The result is higher energy reduction of EcoFaaS over Baseline+PowerCtrl. Finally, by prewarming the missing function containers, the Workflow Controller effectively minimizes the number of function cold starts on the application's critical path and their energy cost by executing them at lower frequencies. It can be shown that, in total, the prewarming technique contributes to 10.2% of the energy savings of EcoFaaS over Baseline+PowerCtrl.

To get more insight into the sources of energy savings, Figure 10.12 shows the average frequency across all cores in a server over time during the peak load for Baseline and EcoFaaS. We can see that EcoFaaS always operates at lower frequencies than Baseline. Recall that the Core Pools in EcoFaaS are reconfigured every $T_{refresh}$ = 2s to adjust to load changes. This is why the average frequency fluctuates. Figure 10.13 shows the distribution of core frequencies used by *EcoFaaS* across different dynamic function invocations. More than half of the invocations need <2.0GHz. Most invocations (25%) run at 1.8GHz, while the least number of invocations run at the highest frequency (4%) and at the lowest frequency (7%).

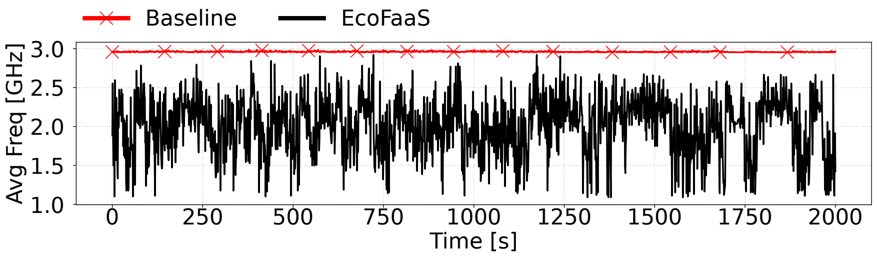


Figure 10.12: Frequency over time during the peak load averaged across all cores in a server for *Baseline* and *EcoFaaS*.

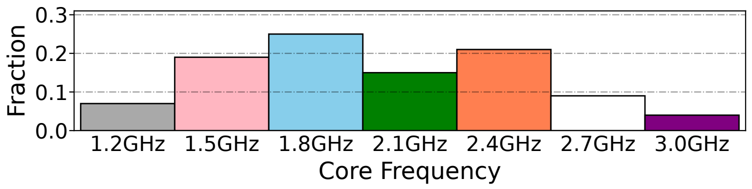


Figure 10.13: Distribution of core frequencies used by *EcoFaaS* across invocations.

10.7.2 Energy Savings with Varying System Load

We evaluate EcoFaaS with different system loads in a cluster with 20 servers. We generate *Low*, *Medium*, and *High* loads with a Poisson distribution for request inter-arrival times. Recall that these loads correspond to CPU utilizations of 25%, 50%, and 70%, respectively. Figure 10.14 shows the normalized and absolute energy consumption of Baseline, Baseline+PowerCtrl, and EcoFaaS with the three load levels for each individual benchmark and for the sum of all the benchmarks. Every client request invokes one of the twelve evaluated benchmarks randomly. Therefore, the distribution of invocations to the different benchmarks is different than in Figure 10.11.

With low load, the overall CPU utilization is low. Hence, even Baseline consumes only a modest amount of energy. With medium load, Baseline+PowerCtrl reaches its sweet spot, mainly for two reasons: (i) the queuing effects are not substantial and, therefore, Baseline+PowerCtrl does not overestimate queuing time and does allow most of the requests

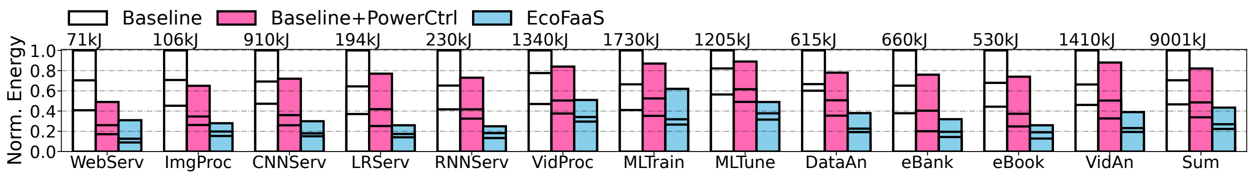


Figure 10.14: Normalized energy consumption of *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* with Low, Medium, and High loads. In a given bar, the two horizontal lines show the values for Low and Medium loads, and the total bar corresponds to High load. All bars are normalized to Baseline-High. The numbers on top of the Baseline bars are the absolute energy consumption of Baseline-High for a 1-hour run on 20 servers.

to execute at lower frequencies, and (ii) there are few context switches across different containers and, therefore, containers keep their owned cores and the core frequencies do not need to be changed often. However, the benefits of Baseline+PowerCtrl diminish with high load. Request queues start building up and Baseline+PowerCtrl assigns higher frequencies than are actually needed to invocations. Further, cores frequently context switch between different containers, which results in crossing the boundary between user and kernel space.

Overall, while Baseline+PowerCtrl reduces the energy consumption of Baseline by 18%, 31%, and 27% with low, medium, and high load, respectively, EcoFaaS reduces it by 56%, 61%, and 52%, respectively.

10.7.3 Performance Improvements

To assess EcoFaaS' performance impact, we measure the reduction in end-to-end average and tail latency of requests, and the increase in their throughput. The end-to-end latency of a function or application invocation is the time from when the client sends a request until when it receives the result.

Tail latency. Figure 10.15 shows the normalized and absolute tail latency of the benchmarks when running with Baseline, Baseline+PowerCtrl, and EcoFaaS. The results are averaged across low, medium, and high load. Baseline+PowerCtrl substantially increase the tail latency over Baseline due to the frequent and expensive core frequency changes. For example, under high load, Baseline+PowerCtrl runs LRServ at the highest frequency and WebServ at the lowest frequency. Thus, at every context switch between invocations of these two functions, Baseline+PowerCtrl changes the core frequency on the critical path. This overhead is higher than the time an LRServ invocation spends running on a core. Moreover, with increased load, more functions concurrently invoke the OS to change their core frequency, creating contention and further increasing tail latency.

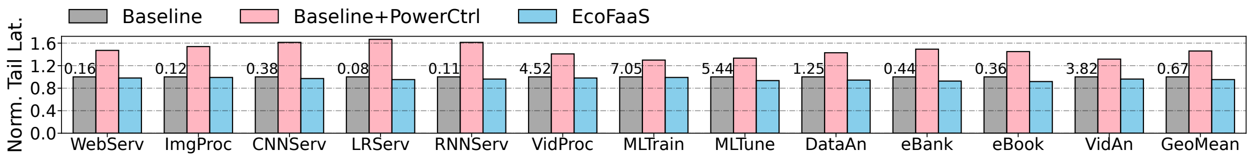


Figure 10.15: Normalized tail latency with *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS* averaged across different loads. The numbers on top of the Baseline bars are the absolute values of the benchmark tail latency measured in seconds.

On the other hand, EcoFaaS is able to keep the tail latency on-par with Baseline and even slightly reduce it. The reduction in tail latency comes from the ability to share cores between multiple functions, which reduces load imbalance. On average, EcoFaaS reduces the tail latency by 5.0% and 34.8% over Baseline and Baseline+PowerCtrl, respectively.

Average latency. We measure the average latency of Baseline, Baseline+PowerCtrl, and EcoFaaS with different loads. As both Baseline+PowerCtrl and EcoFaaS allow invocations to execute at lower frequencies, their average response time is higher than the one with Baseline—which executes all requests at the highest frequency. As the load increases, more requests also need to execute at high frequencies with Baseline+PowerCtrl and EcoFaaS and, thus, their difference with Baseline shrinks. On average, it can shown that EcoFaaS increases the average response time over Baseline by 1.51×, 1.33×, and 1.17× in low, medium, and high load, respectively. The reason is the deliberate decision of EcoFaaS to slow-down function execution to the point of its SLO, in order to save energy. EcoFaaS finishes all requests within their deadline and reduces the average response time over Baseline+PowerCtrl by 13%, 19%, and 18% in low, medium, and high load.

Throughput. We measure a system's throughput as the highest sustained load that allows the benchmarks to still meet their SLO—which is defined as a tail latency below 5× the execution time in an unloaded system. Figure 10.16 shows the tail latency of CNNServ as we increase its load with Baseline, Baseline+PowerCtrl, and EcoFaaS. The dashed line is CNNServ's SLO. We see that EcoFaaS and Baseline keep the tail latency below the SLO until a load of 850 RPS, which is their throughput. On the other hand, Baseline+PowerCtrl reaches the function's SLO at 350 RPS, which is its throughput. The reasons for EcoFaaS's high throughput are the improved CPU utilization and the reduced number of core frequency changes. For CNNServ, EcoFaaS improves the throughput over Baseline+PowerCtrl by 2.4×.

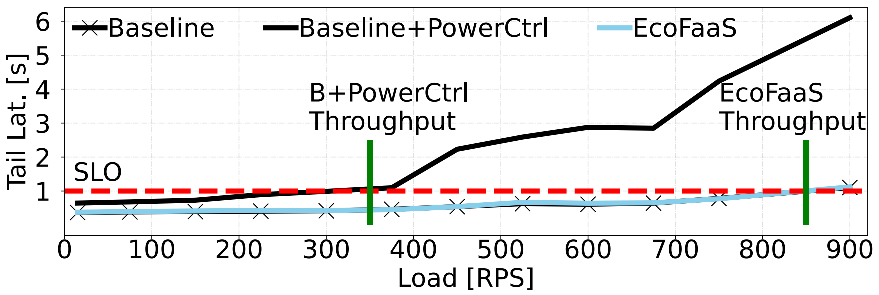


Figure 10.16: Tail latency of CNNServ with the three systems while increasing the load. The dashed line indicates CNNServ's SLO.

For all the benchmarks, Figure 10.17 shows the normalized and absolute throughput with the three systems. On average, EcoFaaS improves the throughput over Baseline+PowerCtrl by 1.8×.

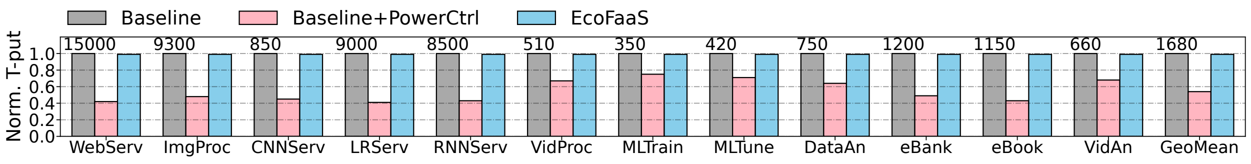


Figure 10.17: Normalized throughput with *Baseline*, *Baseline+PowerCtrl*, and *EcoFaaS*. The numbers on top of the Baseline bars are the absolute values of the benchmark throughput measured in Requests-Per-Second (RPS).

10.7.4 EcoFaaS Component Analysis

We measure the overheads introduced by different EcoFaaS components. First, for applications with multiple functions, the Workflow Controller uses the PuLP library [452] for its MILP solver to compute the optimal per-function deadlines. We test the time for the solver to produce its outputs as we vary the number of functions in an application (from 2 to 20) and the number of different frequency levels (from 2 to 10). The overhead of the MILP solver is around 10ms. Since EcoFaaS executes this operation only once every 5s and in the background, off the critical path, this overhead accounts for only 0.2% of the CPU cycles.

Second, Function Dispatchers communicate with the FPSs via shared memory to register invocations for execution. The communication overhead is only a few μs. Finally, the Node Controller periodically reassigns cores across pools and sets the cores' frequencies. The controller runs with root privileges. Hence, core frequency changes are triggered by writing to the MSR registers [453], and take effect in a few 10s of μs.

We measure the accuracy of our prediction system. First, for functions whose response time does not change with different inputs (such as WebServe), we use a simple EWMA [438] approach. The Mean Absolute Percentage Error (MAPE) observed when predicting T_{run}, T_{block}, T_{queue}, and $Energy$ is 1.8%, 2.4%, 3.5%, and 1.9%, respectively. For functions whose response time and energy depend on the inputs (such as ImgProc), we predict the time and energy with a three layer neural network with ReLU activations. Its accuracy for both execution time and energy is 96.5% on average, while the prediction time is only 10-30μs.

10.8 CONCLUSION

This chapter proposed *EcoFaaS*, the first energy-management framework for serverless environments. EcoFaaS automatically splits the user-provided end-to-end application SLO into per-function deadlines that minimize the total energy consumption. It profiles functions online and, based on their deadlines, picks the optimal core frequencies. Further, EcoFaaS splits the cores into multiple pools, where all the cores in a pool run at the same frequency, and dynamically changes the size and frequency of the pools based on system conditions. Compared to state-of-the-art systems, EcoFaaS reduces the total energy consumption of serverless clusters by 42% while simultaneously reducing the tail latency by 34.8%.

CHAPTER 11: Processor Overclocking for Further Energy Savings

11.1 INTRODUCTION

Cloud services provision resources to meet their peak performance requirements [152, 316, 318, 391, 454]. Many services need to keep their high-percentile latency (*e.g.*, P99) below a predetermined Service-Level Objective (SLO) [66]. These services incur high operating costs to reserve enough resources for handling infrequent load spikes and leave a substantial portion underutilized or even idle for the majority of time when their load is below its peak.

As an example, Figure 11.1 illustrates the aggregate load pattern on a typical weekday of three services that are part of Microsoft's productivity and collaboration suite. Collectively, these three services use ∼1M virtual cores (across regions) to handle peaks that last for a few hours per day - between 10 am to noon for *Service A* and 5 minutes at the top and bottom of the hour for the other two services.

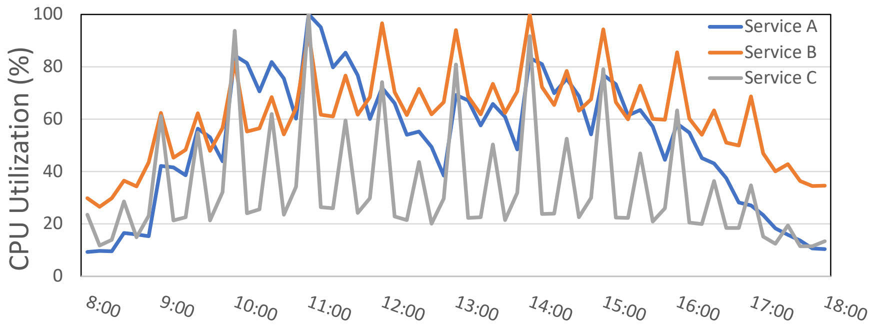


Figure 11.1: Load pattern on a typical weekday in one region. Utilization is normalized to the peak of each service.

Emerging cloud paradigms, such as autoscaling [455, 456, 457] and serverless computing [19, 20, 21, 22], can be used to dynamically remove and add Virtual Machine (VM) instances for managing cost. However, these solutions (1) can increase the application's tail latency as booting up a new VM can take up to a few minutes [458], and (2) cannot be easily applied for stateful services [459, 460]. Hence, many applications still statically provision for infrequent load spikes.

On the other hand, advances in processing and datacenter cooling technologies have enabled component (*e.g.*, CPU, GPU) overclocking, *i.e.*, operation beyond typical voltage and power design limit [461]. Overclocking boosts a workload's performance and enables handling transient load spikes in a cost-efficient manner. For example, CPU overclocking during a service's peak can keep the tail latency below the required SLO, while saving cost by reducing provisioned resources.

However, overclocking is not free. If used naively, it increases power draw and can cause frequent power capping events that diminish performance benefits. Worse, it degrades component lifetime (or reliability) through accelerated wear-out and, thus, cannot be used indefinitely. The limited amount of overclocking needs to be used smartly as it may not benefit all workloads at all times: overclocking the CPU of a memory-bound workload, or overclocking a workload while experiencing a low load will not provide much benefit. Finally, providers also need to protect workload SLOs when overclocking is unavailable. For example, a workload might have been under-provisioned due to reliance on overclocking, but it would miss its SLOs under peak load if its VMs cannot be overclocked. Therefore, providers must use overclocking carefully while managing the associated risks.

Our work. For efficient use of overclocking in the cloud, we analyze cloud workloads and production traces, including the services from Figure 11.1. We observe the following. First, overclocking improves the performance of popular cloud workloads. However, a workload-agnostic overclocking scheme is suboptimal and often leads to missed SLOs or wasted overclocking cycles. Second, power and lifetime headroom exists to overclock most of the times without triggering power capping or compromising on reliability. Third, resource utilization history can be used to predict the availability of power and reliability impact from overclocking. Fourth, servers' power draw within a power delivery unit (*e.g.*, a rack) is diverse, but the limit is still evenly distributed, which disproportionately hurts the performance of power-hungry servers during a capping event. However, predictability in power draw enables assigning heterogeneous limits. Finally, a decentralized approach for power draw enforcement enables servers to find an efficient limit in case of initial assignment mispredictions.

Based on our insights, we design SmartOClock, the first distributed overclocking management platform for the cloud. It enables a variety of cloud workloads to run with high performance at a lower cost. SmartOClock achieves its goals through four design principles.

First, SmartOClock uses bidirectional communication with the application to maximize the application's benefits from overclocking. Applications can use metrics (*e.g.*, latency, CPU utilization) or schedule-based policies and the overclocking decisions can be made based on instance- and deployment-level monitoring. Second, SmartOClock uses *admission control* to reserve power (from any headroom) and overclocking budget for workloads. This step provides a predictable overclocking experience for workloads and SmartOClock can take corrective actions, like scale-out, if it is unable to honor a reservation. Third, SmartOClock leverages power predictability for assigning *heterogeneous* server power budgets, which provide better performance during capping for power safety. Finally, SmartOClock makes *decentralized* overclocking decisions for improved fault tolerance. Each server takes local decisions for granting overclocking requests based on its assigned power and overclocking budgets. It can also perform explorations to revise inefficient assignments (*e.g.*, due to mispredictions).

We evaluate SmartOClock on a real server cluster and through simulations by using production traces. The cluster evaluation is performed on 36 overclockable servers (across 2-racks) running latency-sensitive microservices as candidates for overclocking and throughput-optimized power hungry machine learning (ML) training workloads, which are not overclocked. Our results show that SmartOClock reduces the P99 latency by 8.9% and application cost by 30.4% for latency-sensitive microservices, and the total cluster energy consumption by 10% over a state-of-the-art autoscaling solution. To validate our findings at scale, we use traces from hundreds of production racks and simulate SmartOClock. When compared to all practical policies, SmartOClock reduces the number of power capping events by up to 94.7% while increasing the overclocking success rate by up to 61.8%. We have also created a 2-rack overclockable cluster for production experimentation and share some lessons in Section 11.6.

Related work. While there is a rich body of work on CPU turbo-boost [462, 463, 464, 465, 466, 467, 468] and datacenter power management [469, 470, 471, 472, 473, 474, 475], overclocking introduces unique challenges not addressed by the prior work. First, a cloud provider does not need to manage any reliability impact from turbo since CPU vendors design turbo to meet a provider's lifetime requirements. Cloud CPUs operate in performance mode, which always operates them at the highest turbo frequency within constraints (*e.g.*, power, thermal) [467, 476]. Vendors do not specify turbo timing limitations nor advise software-level core wear leveling in their warranty terms [477, 478], and non-judicious turbo use does not degrade reliability [479]. Generally, CPU failure is amongst the lowest types of failure in cloud servers [480, 481]. Second, the power oversubscription policies factor the higher demand from turbo. Although this approach increases the total cost of ownership, it is necessary to meet the performance Service-Level Agreements (SLAs) [482, 483, 484]. In contrast, overclocking (beyond turbo) further improves performance but has a reliability impact that is not covered at design time by the vendors. Furthermore, a provider does not need to provision power for overclocking since turbo is sufficient to meet its performance SLAs. Therefore, overclocking is opportunistic - a provider needs to manage the power and reliability impact, while protecting workload SLOs when overclocking is unavailable; a problem setting not explored by prior work.

This chapter makes the following main contributions:

• We characterize the opportunities and challenges of overclocking cloud workloads, including the impact on power and component lifetime.
• We propose SmartOClock, a distributed overclocking management platform specifically designed for the cloud.
• We evaluate SmartOClock in a real system running latency-critical workloads, and using large-scale production traces.
•We share lessons from overclocking production workloads.

11.2 BACKGROUND

Power management in cloud datacenters. The power delivery system in a cloud datacenter is organized in a hierarchy [471, 472, 474, 475]; the power budget of each parent node is split equally among its children. As providers oversubscribe power to improve utilization, the sum of the peak power draw of children nodes can exceed the budget of the parent (*e.g.*, servers in a rack) [471, 472, 474]. Under normal operation, child nodes can draw more than their even share if the cumulative power is below the parent's limit. When it exceeds a threshold, power capping mechanisms (*e.g.*, Intel RAPL [485], prioritized capping [473, 474]) are used for safety. These mechanisms hurt performance as they reduce CPU frequency and can even throttle memory to restrict server power. To meet their performance SLAs, providers carefully oversubscribe to minimize/avoid capping events.

Component overclocking. Prior work shows the feasibility of overclocking in the cloud [461]. Overclocking operates components (*e.g.*, CPUs, GPUs) beyond their specifications to get frequencies even higher than turbo [486, 487].

A large fraction of cloud workloads, such as search or video conferencing [152, 488], are user-facing applications with transient load spikes. These workloads collectively consume millions of virtual cores to handle peak load. For Microsoft's productivity and collaboration services, although chat and conference calls occur throughout the day, the peak that governs resource provisioning lasts for a few hours each day (Figure 11.1). Overclocking can be used during these peaks to save costs. However, a provider needs to manage the risks from overclocking. For example, for reliability management, the peak duration needs to be within the daily overclocking budget (*e.g.*, 10% per day) that satisfies component lifetime goals. Overclocking impacts reliability [489] due to three main reasons: (1) gate oxide breakdown, (2) electro-migration, and (3) thermal cycling. These processes are time-dependent and accelerate the lifetime reduction. Prior work has showed that there is an exponential relationship between temperature, voltage, and component lifetime [461, 490, 491, 492, 493].

11.3 CHALLENGES AND OPPORTUNITIES FOR OVERCLOCKING IN THE CLOUD

A successful overclocking management scheme needs to satisfy workload performance requirements, while managing the impact of overclocking on power and component lifetime. To design such a scheme, we answer the following questions.

Q1: When do workloads benefit from overclocking? To efficiently use overclocking, a cloud platform needs to understand workloads' behavior and needs. Treating VMs as opaque and using workload performance proxies (*e.g.*, instructions per cycle, CPU utilization) for overclocking can be suboptimal as the relationship between proxies and target performance metric is not always clear. Without knowing a workload's performance goals, the platform may overclock prematurely (*i.e.*, under low load that does not impact tail performance) and, due to the lifetime impact, lose the ability to overclock when really needed. Combining IPC with CPU utilization as a proxy for load can be inefficient too because the performance of some workloads is impacted at a moderate CPU utilization while others can sustain high utilization. Finally, operators can even have *deployment-level* goals for provisioning (number of VMs) and overclocking based on instance-level monitoring only will be inefficient.

To illustrate these scenarios, we profile two classes of popular cloud workloads: (1) microservices from the largest open-source benchmark suite, DeathStarBench [40], and (2) a proprietary web conferencing application called WebConf.

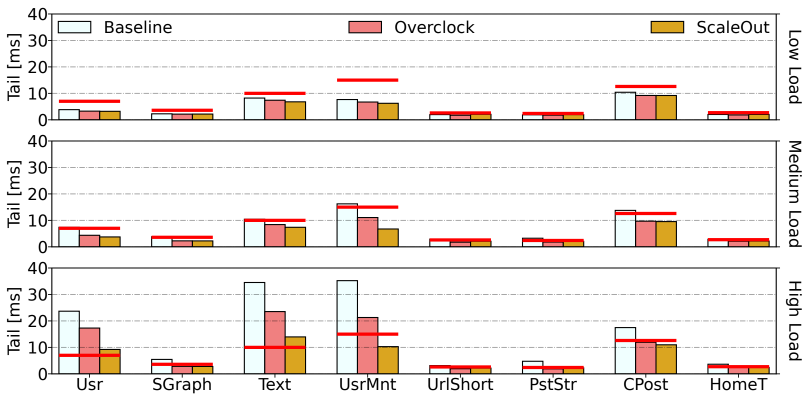


Figure 11.2: Tail latency of SocialNet microservices with different loads in Baseline, Overclock, and ScaleOut environments.

Microservices. We run eight SocialNet microservices [40] under varying loads (low, medium, and high) in three environments: *Baseline*, *Overclock*, and *ScaleOut*. *Baseline* and *Overclock* run a single VM at turbo (3.3 GHz) and overclocked (4.0 GHz) frequency. *ScaleOut* has two VMs running at turbo. Figure 11.2 shows the tail latency of the microservices. The red

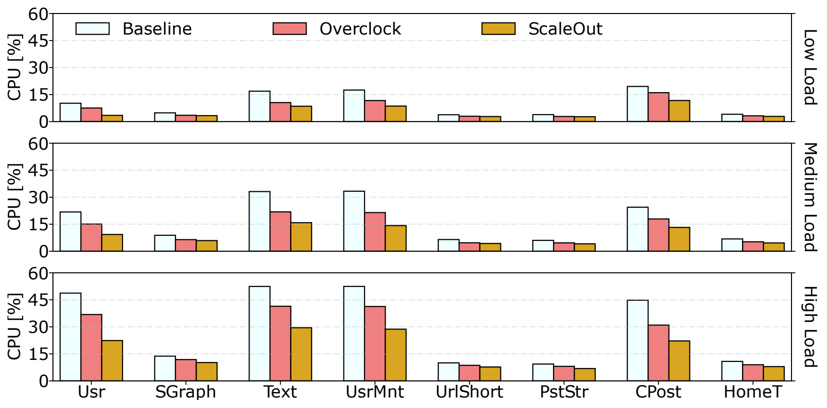


Figure 11.3: CPU utilization of SocialNet microservices with different loads in Baseline, Overclock, and ScaleOut environments.

horizontal line indicates SLO, where the SLO for each service is set to be 5 times its execution time on an unloaded system [283, 284, 361]. Figure 11.3 shows their CPU utilization.

ScaleOut is provisioned to handle the peak load and always operates 2 VMs that run at turbo. Although it provides the best performance, it also incurs the highest cost. In contrast, Overclock uses a single VM and still keeps the tail latency below the SLO in many cases, thereby avoiding the need to scale out. However, some services (*e.g.*, Usr) can tolerate higher CPU utilization without violating their SLO while others (*e.g.*, UrlShort) violate their SLO even under a low utilization. Therefore, a workload-agnostic policy using CPU utilization for overclocking will make suboptimal decisions. These observations hold for any cloud workload with similar characteristics – bursty load with tail latency as the key metric. For example, ML inference servers [361, 494], serverless computing [63], and key-value stores [495] amongst others.

WebConf. The workload hosts conferences in a VM. For fault-tolerance, operators provision VMs across availability zones (AZ) in a region. In an AZ, provisioning keeps the average *deployment-level* CPU utilization below 50% to handle load from another failed AZ. Overclocking can save cost for WebConf through deployment-level decisions. Individual VMs can have high utilization, but overclocking them is suboptimal since the deployment-level utilization may be below the target.

To illustrate, we execute WebConf on two VMs. VM_1 has a low load while VM_2's load is high. Figure 11.4 shows the VM- and deployment-level average CPU utilization. Although overclocking provides a benefit, it is unnecessary since the baseline already meets the workload performance (provisioning) goal.

Q2: Are there enough resources for overclocking? Since overclocking increases power draw and component wear out, we need headroom for these resources.

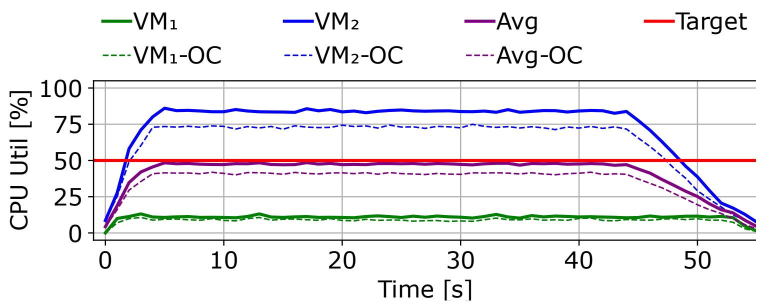


Figure 11.4: CPU utilization timeline with and without overclocking for two WebConf VMs.

Power headroom. We analyze the power draw of 7.1k dedicated racks that run Microsoft's productivity and collaboration services, including those from Figure 11.1, which are used by millions of users across the world. The racks span all major regions (*e.g.*, United States, Europe, Asia) and each rack has 24-32 servers. The analysis period is 6 weeks (April 10th – May 12th, 2023). Figure 11.5 shows the CDF of average, median (P50), and P99 rack power utilization. Half the racks have an average utilization lower than 66%. Importantly, 50% and 90% of the racks have P99 lower than 73% and 89%, respectively. We observe similar power patterns on non-dedicated racks with a mix of first- and third-party workloads.

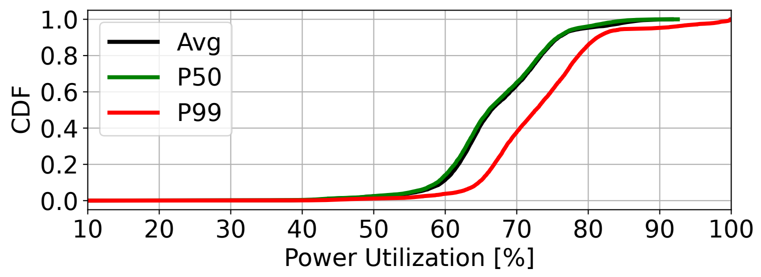


Figure 11.5: Average, median (P50), and peak (P99) power utilization of 7,100 racks over 6 weeks in three regions.

To estimate the power impact from overclocking, we use the overclocking requirements of critical user-facing workloads that constitute 45% of the deployed cores. Their requirements vary – some require overclocking for several minutes per hour, while others for multiple hours per weekday. Figure 11.6 shows the power draw of a rack without and with overclocking for five weekdays; the red line shows the rack power limit. Each server in this rack hosts VMs of many distinct services and captures a typical datacenter environment with multi-tenant servers. The rack power draw is below the limit for the baseline, but overclocking exceeds the limit and causes capping. More generally, overclocking the selected workloads will not result in capping for 85% of the time. For the remaining 15%, naive overclocking causes 30-50% degradation in workload performance (core frequency) due to capping. However, there is still headroom available on these power-constrained racks, but it is insufficient to

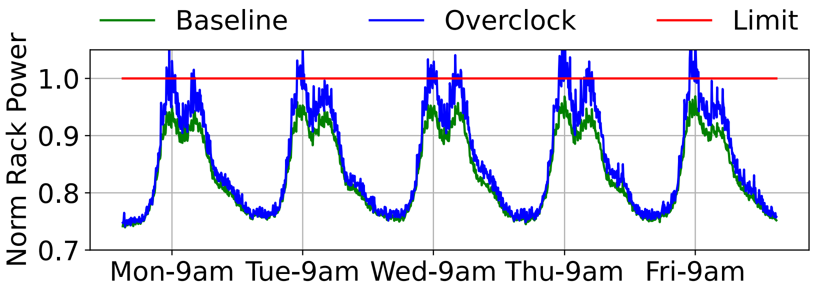


Figure 11.6: Example of rack power draw over 5 weekdays.

overclock to the highest frequency; the available headroom is 75% of the requisite at P99.

Therefore, most of the time (85%) racks have the needed power headroom for overclocking. However, a power-aware policy is needed for the constrained scenarios.

These findings are with the default Azure VM scheduler that uses a set of resource-centric placement rules [496]. Providers can add power-aware scheduling policies to aid overclocking, but this exploration is future work. Nonetheless, even with optimized placement, there will still be power-constrained scenarios where overclocking has to be performed carefully.

Component lifetime headroom. Prior work shows that advanced cooling (*e.g.*, wax, immersion) is needed for enabling sprinting/overclocking [461, 462, 463, 497]. However, there is opportunity to overclock even in air-cooled server deployments. Cloud server cooling is designed for operating components at their rated thermal design power (TDP). However, servers rarely consume their TDP due to low resource utilization in the cloud [152]. Several factors contribute to the low utilization. First, over-provisioning and diurnal workload patterns result in low VM utilization. Second, workload heterogeneity on servers results in low server utilization. Each server hosts many small VMs (2-8 cores). For resiliency, operators spread their VMs across servers and racks. Consequently, the VMs on any given server belong to different workloads. This heterogeneity results in low server utilization as the workloads have different peak times. Consequently, components are not thermally constrained for overclocking in air and advanced cooling can be used to enhance the capability (*e.g.*, duration) as lower operating temperatures reduce ageing [461]. Finally, since overclocking does not exceed the TDP nor the rack limit, it will not cause additional cooling-related failures.

In fact, under-utilization enables overclocking in air. Vendors assume near-100% usage for determining frequencies/voltage (*e.g.*, turbo) that satisfy the lifetime goals. Under-utilization accumulates lifetime credits that can be consumed via overclocking. To understand the opportunity, we use a 7nm composite processor model from TSMC. It uses a complex relation between overclocking (voltage scaling) and CPU utilization (time period at the specified voltage) to model the ageing from wear-out in the form of gate oxide failure [498, 499]. The model predicts that a CPU ages by 2.5 years over a 5-year period for a

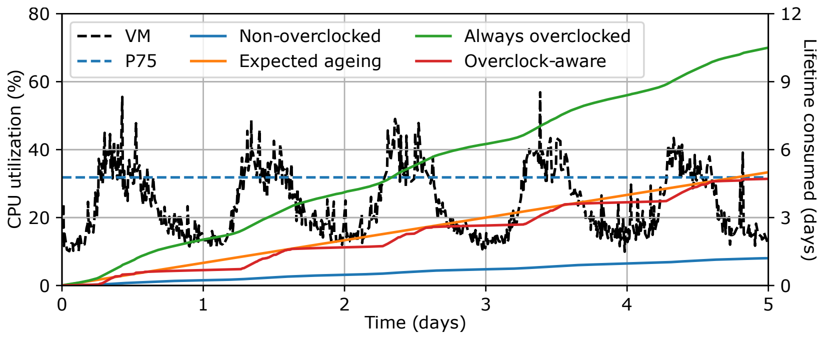


Figure 11.7: CPU ageing for a VM running a workload with a diurnal pattern under multiple overclocking policies.

conservative fleet usage. The remaining 2.5 years can be used for overclocking. But naively overclocking for 50% of the time ages the CPU by 5 years in less than a year use due to accelerated wearout. A smart system can constrain overclocking so that the part ages according to the reference (*i.e.*, 1 year ageing over a 1-year period).

Figure 11.7 illustrates the effect of overclocking policies on ageing. It shows the 5-day CPU utilization of a production workload with a diurnal pattern of daily midday peaks above 50% and valleys lower than 20% at night. The expectation is that the processor ages 5 days over the same period ("Expected ageing"). However, the actual ageing is less than 2-days for the "Non-overclocked" baseline. "Always overclock" ages the CPU over 10 days, indicating that, for the same CPU utilization, overclocking significantly increases wearout. On the other hand, an "Overclock-aware" policy can consume the accumulated credits by overclocking for 25% of the time and not exceed the expected ageing. Offline modeling assumes CPU utilization is unchanged while overclocking for worst-case analysis. However, overclocking's ageing impact will be less if the utilization reduces. To address this limitation, we are working with the CPU vendors on "wearout counters" for online calculation of the ageing impact (see Section 11.6).

Therefore, overclocking is enabled due to low-utilization and can be improved with advanced cooling. A system must carefully manage overclocking to comply with lifetime goals.

Q3: Can we predict the availability of resources? An efficient overclocking system must perform admission control based on available power and lifetime. We observe that a prediction-based approach can yield high accuracy.

Power predictability. A system needs to predict how much power can be used by overclocking without triggering capping. Figure 11.6 shows the baseline power draw of a rack and gives us insights that historical observations of power profiles can be leveraged for prediction. The rack hosts multiple services, where each service can have a distinct power profile. However, due to statistical multiplexing, the combined power draw of the rack with heterogeneous

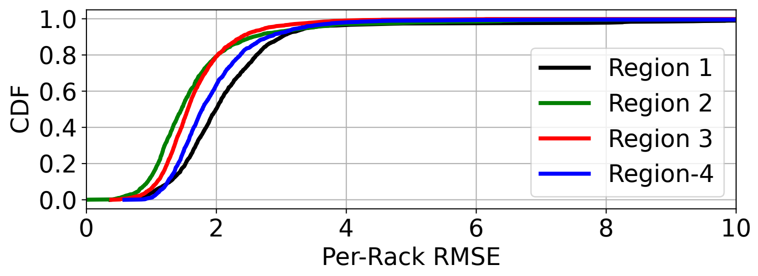


Figure 11.8: CDF of RMSE with the power draw patterns of our predictions across 7.1k racks in four regions.

services shows a repeatable pattern. We analyzed the power predictability of 7.1K racks (1000s of servers) that collectively run >100 services. Although the racks are dedicated for Microsoft's productivity and collaboration services, this dataset accounts the fact that racks and servers on a public cloud host heterogeneous workloads. Furthermore, the dynamicity of cloud platforms (*e.g.*, VM churn according to a workload's needs) is also reflected.

Figure 11.8 shows the CDF of Root Mean Squared Error (RMSE) of rack power predictions of four Azure regions. The RMSE is low even at high percentiles indicating high predictability. For example, in Region 3, 50% and 99% of the racks have an RMSE lower than 1.95W and 5.11W, respectively. The findings are similar in the other regions. Furthermore, an analysis of 20K non-dedicated racks, running a mix of first- and third-party workloads, in the three most popular Azure regions yielded similar results. A major reason for this predictability is long-lived VMs that govern resource utilization. Prior work shows that long-lived VMs (or jobs) account for >95% of allocated resources [152, 454, 500].

Component lifetime impact predictability. To remain within the overclocking lifetime budget, a system needs to predict how much overclocked CPU cycles a given workload will consume. As a server's power draw depends on CPU utilization, predictability in power indicates predictability in CPU utilization. Using the aforementioned methodology for a rack's power, we now analyze the CPU utilization predictability. Our results show that CPU utilization of servers are also predictable: more than 50% and 90% of the servers have an RMSE of CPU utilization lower than 3.13% and 7.82%, respectively.

Therefore, historical observations of power draw and CPU utilization can be used to predict the available power and component lifetime headroom for overclocking.

Q4: How to assign power budgets? A server's power budget for "safe" overclocking depends on the power draw of the other servers in the hierarchy (*e.g.*, a rack). Under fair share, the rack power budget is split equally across all servers and each server can locally ensure that its power draw stays below the limit to avoid capping while overclocking. However, this approach is inefficient since some servers may not be able to overclock even

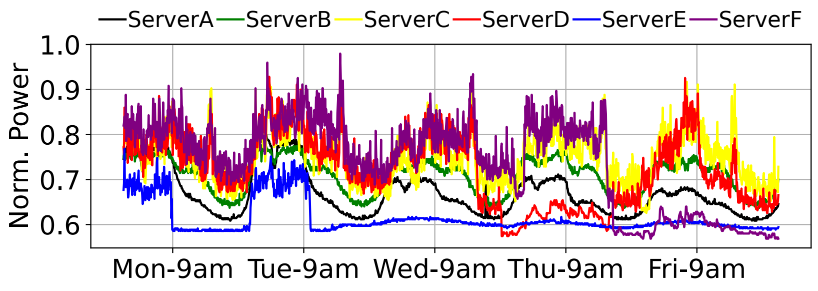


Figure 11.9: Normalized power draw over time of six randomly chosen servers within a rack.

while the rack is not power-constrained.

Figure 11.9 shows the normalized power draw over 5 weekdays of six randomly chosen servers in a rack; each server is a different color. We can see that servers have very different power profiles. Some servers may use even 30% less power than others. In addition, servers that consume the most power in a rack change over time. For example, at different timestamps, ServerC, ServerD, or ServerF may be the power dominant one.

Therefore, an efficient overclocking system needs to split the rack power budget *heterogeneously* across servers. Historical observations of server power demand and rack-level headroom can be used for the heterogeneous attribution.

Q5: How to efficiently use the power? The power headroom for overclocking in a rack is consumed by all servers in that rack. Thus, to grant or reject an overclocking request, each server should contact a centralized entity that has the global view of the rack's total remaining power headroom. Unfortunately, this approach is expensive and limits the system's fault-tolerance – if the centralized entity fails, then all overclocking requests would be rejected. Making local overclocking decisions using assigned server power budgets improves fault tolerance. However, overclocking requests may still be rejected due to inefficient assignments. For example, a scheme that uses power predictions for budget assignments can be suboptimal due to mispredictions.

Therefore, a high-performance and fault-tolerant overclocking system needs to be decentralized and should allow servers to explore beyond their potentially stale power limits.

11.4 SMARTOCLOCK

Driven by the characterization insights, we propose *SmartOClock*: a distributed overclocking management platform for the cloud. It is readily integrated with existing platforms and enables a wide variety of workloads to run with high performance at lower cost. SmartOClock responds to the outlined questions for an efficient overclocking scheme through four novel features. First, it is *workload-intelligent* as it uses hints provided by workloads to ex-

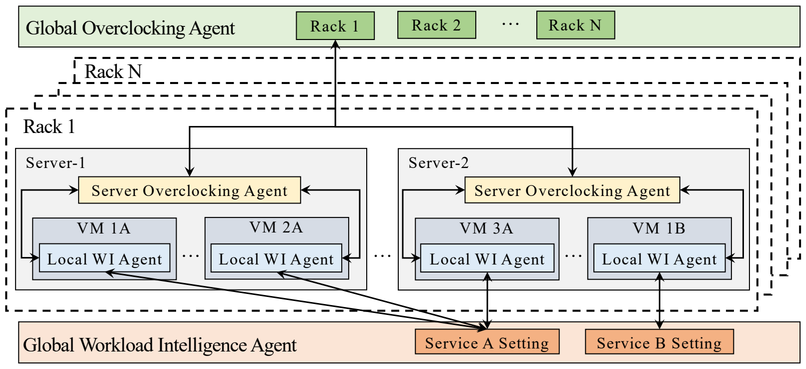


Figure 11.10: Overview of the SmartOClock overclocking system.

tract the most benefits from overclocking. Second, SmartOClock performs *prediction-based admission control* of overclocking requests to avoid power capping and premature component wearout. Third, it uses predictions to split the rack power limit *heterogeneously* across servers. Finally, SmartOClock uses a *decentralized scheme* for budget enforcement while overclocking and allows controlled exploration to revise inefficient assignments.

Architecture. Figure 11.10 shows the architecture of SmartOClock. The system is organized hierarchically where each controller manages the components on its level and communicates with the controllers from the upper and lower levels. First, when deploying their services, the workload owners configure the *Global Workload Intelligence Agent* for their service. They specify the conditions under which the workload needs to be overclocked. As workloads are composed of one or more VMs, each VM is deployed with its own *Local Workload Intelligence Agent*. Like conventional auto-scaling, the local agent collects the metrics of interest from the VM and sends them to the global agent. Thus, this setup does not introduce new security or privacy challenges. The global agent uses the metrics to decide if any VM needs to be overclocked and sends a signal to the local agent of such VMs. On receiving a signal, a local agent sends an overclocking request to the *Server Overclocking Agent* (sOA). The request can be submitted via a local interface, such as a hypervisor-specific shared memory implementation [501, 502, 503] or locally-terminated network endpoint [504, 505, 506]. The sOA predicts if there are enough resources to satisfy the request and, accordingly, grants or rejects the request. If the request is rejected, the local agent informs the global agent which then takes corrective actions (*e.g.*, request scale-out or redistribute the load towards the overclocked VMs). In the background, each sOA monitors a server's power and overclocking needs, and creates a profile to be periodically sent to the *Global Overclocking Agent* (gOA). The gOA uses the profiles to assign efficient per-server budgets. In turn, an sOA uses the assigned budget for admission control until the budget gets updated.

11.4.1 Workload-Aware Overclocking

Overview. SmartOClock extends the existing autoscaling interface with overclocking. A workload specifies the scale-up (start) and scale-down (stop) thresholds for overclocking. The overclocking hints can be inserted by developers after profiling or they can be automated using the existing tools for automatic instance scaling [35, 507, 508, 509, 510]. Like conventional autoscaling, the overclocking thresholds can be: metrics-based or schedule-based. Under *metrics-based* overclocking, workloads can use application metrics (*e.g.*, tail latency, queue length) or resource utilization (*e.g.*, CPU, network) to trigger overclocking. The granularity of application hints can be per-function in the case of tail latency or per-VM in the case of resource utilization. These metrics can then be monitored per- and across-VM instances for specified time intervals to meet an application's goals. Additionally, workloads that have predictable times for high traffic (*e.g.*, 9-10 AM local time) can use *schedule-based* thresholds. Finally, workloads can also use a combination of metrics- and schedule-based. Importantly, extending the autoscaling interface for overclocking enables using scaling out (creating new VMs) as a fallback mechanisms for when overclocking is not possible. The scale-out signal can also be triggered proactively by SmartOClock using predictions for the ability to overclock (see Section 11.4.4).

Adopting WI by cloud users. Although workload owners already carefully tune the metrics and thresholds for horizontal scaling, there is overhead in repeating the process for vertical scaling (overclocking). To ease adoption, SmartOClock can be extended to infer the overclocking thresholds. It can leverage workload historical data to determine scale-up values. The lifetime impact of overclocking can be factored in this analysis. For example, use P90 of historical value if overclocking can be performed for 10% of the time only to comply with lifetime goals. The overclocking impact needs to be estimated to determine the scale-down value. An inaccurate estimate can either cause dithering if it is too close to the scale-up threshold or waste precious overclocking time if the estimate is too low. Performance models using low-level architectural counters can be used for the estimation. Workload owners can also leverage the inferred thresholds as an initial estimation.

11.4.2 Overclocking Admission Control

Overview. Naively granting overclocking requests (1) increases the chance of power capping events that deteriorate performance, and (2) accelerates wear out of server components. Instead, SmartOClock performs admission control for the overclocking requests based on *power and component lifetime impact predictions.* It predicts (1) the rack's power draw to assess if overclocking will result in capping, and (2) the CPU utilization of VMs requesting overclocking to assess if overclocking them will exceed the lifetime budget. Using the predictions, SmartOClock decides (1) if the requested power and overclocking budgets can be reserved for a schedule-based workload, or (2) for how long a given VM can be overclocked under a metrics-based policy before needing corrective actions. Note that the power reservation is *soft*, the power can be taken by workloads outside of the system that do not need overclocking and SmartOClock needs to adjust.

Managing power. As observed in Section 11.3, the power draw of racks and servers is highly predictable. Hence, the gOA and sOA continuously monitor the server and rack power draw and use the data gathered during monitoring to periodically (*e.g.*, weekly) recompute the per-rack and per-server power templates. The templates are used to predict if the additional power of overclocking will trigger a capping event.

SmartOClock creates a power template using *per-day aggregation* of power draws across all weekdays in the prior week. The template represents a single day and the same template is used for predictions for all days in the following week. For example, the template's value at 9AM is the median of rack's power draw at 9AM across all five weekdays. A separate template is used for weekends. The intuition for this approach is that (1) using a coarse-grained measurement (*e.g.*, the maximum over a week) is too conservative (*i.e.*, it unnecessarily rejects many overclocking request) and (2) using fine-grained measurements (*i.e.*, all power measurements from the prior week) is insufficiently robust to outliers (*e.g.*, holidays during the prior week). Section 11.5.2 compares the accuracy of several template-creation strategies.

Managing lifetime impact from overclocking. A max time to overclock a component is obtained through an offline analysis with the vendors (*e.g.*, 10% over a 5-year period). This analysis uses realistic, yet conservative, utilization of cloud components to determine the opportunity. The duration of individual overclockings can vary, but SmartOClock needs to honor the *total* overclocking time assumption to comply with component lifetime goals. This requirement is the same as for using turbo-boost on non-overclockable CPUs.

To get uniform overclocking over a component's expected lifetime, SmartOClock divides the overall budget into epochs. The definition of an epoch is configurable (*e.g.*, a day, week). Using a longer epoch, such as a week, enables assigning unused budgets from the weekend to the weekdays. Hence, SmartOClock defines an epoch to be a week and calculates per-weekday max overclocking time.

Each sOA ensures that the overclocked time-in-state of a component (*e.g.*, per-core of a CPU) does not exceed the limit. Tracking and enforcement is per-server; an sOA uses mechanisms like Intel PMT [511] for the time-in-state tracking and denies overclocking requests if the budget is exhausted. Due to hardware heterogeneity, vendor-specific APIs are needed for the tracking; calling such APIs is already supported by operating systems (*e.g.*, Intel PMT [512] and AMD HSMP [513] on Linux), and enforcement is via standard interfaces (*e.g.*, CPPC [514] for CPU cores). For a predictable overclocking experience, an sOA also reserves overclocking budgets for scheduled requests. Unused budgets can be used by metrics-based overclocking and/or carried over to the next epoch.

11.4.3 Heterogeneous Power Budgets

Overview. SmartOClock splits the rack power budget *heterogeneously* amongst servers. Each sOA collect its server's power draw and overclocking needs over time to create power and overclocking *templates*. The power template specifies a server's draw at a given timestamp. The overclock template specifies the number of cores that *requested* and were *granted* overclocking. The sOAs periodically (*e.g.*, weekly) exchange their templates with the gOA. The gOA combines power and overclocking templates of all sOAs and computes individual power budgets. It grants power credits to servers for periods when VMs are overclocked, per the reported template.

Power budget computation. The power budget computation happens in three phases. First, the gOA uses its power model to separate the server's power into the regular and overclock power; the number of cores from the server's overclocking template enable the gOA to discriminate the two portions. Second, the gOA assigns to each sOA the initial power budget that is equal to the server's regular power draw. Finally, the gOA splits the remaining power headroom based on the overclocking requirements, *i.e.*, servers with more overclocked cores in the past get larger extra power budgets for the future.

For example, a rack has two servers (X and Y) and 1.3kW power limit. Typical power draw without overclocking of X and Y at 9AM is 400W and 300W, respectively. Thus, the unused power is 600W. In addition, at 9AM, X and Y typically need to overclock 5 cores (extra 50W) and 10 cores (extra 100W), respectively. Based on this history, the gOA computes the power budgets for 9AM: for X $400W + \frac{50\times600}{50+100}W = 600W$, and for Y $300W + \frac{100\times600}{50+100}W = 700W$.

11.4.4 Decentralized Budget Enforcement

Overview. SmartOClock takes *decentralized* decisions by allowing servers to locally process overclocking requests from their VMs. An sOA uses the server's power profile to predict if overclocking will exceed the server's power budget. As the budget computations rely on predictions, they may become stale. Thus, SmartOClock allows sOAs to explore beyond their initial assignments. Similarly, an sOA tracks the overclocking time of VMs and predicts if a VM will run out of budget. Then, to avoid missed SLOs, the sOA informs the global WI agent (via local) of the inability to overclock; in turn, the global WI agent can take corrective actions using the configured scale-out policies. Enabling local decisions is key for reactively handling activity bursts under metrics-based overclocking. The overclocking trigger by a WI agent is conveyed to the (local) sOA that can start/stop overclocking in order of a few milliseconds. Furthermore, if the assigned power budget is insufficient (*e.g.*, misprediction, due to change in load), then the sOA can independently explore a higher budget to maximize the extent (frequency) of overclocking.

Power budget enforcement. The gOA periodically sends the heterogeneously assigned power budgets to each sOA. Then, each sOA performs prioritized per-VM power management [474] via a feedback loop to control the server power draw while overclocking. For example, scheduled overclocking VMs can be of higher priority compared to unscheduled (metrics-based) ones. In the feedback loop, an sOA changes the frequency of the overclocked VMs per priority in discrete steps (*e.g.*, 100 MHz). Based on the impact of the last frequency change on the server's power draw, the sOA either: (1) maintains the VMs at the current frequency (if $threshold \leq draw < limit$, where $threshold = limit - buffer$), (2) increases frequency by step size (if $draw < threshold$), or (3) reduces frequency by step size (if $draw > limit$). Prioritization enables overclocking the more important VMs to the maximum extent before less important VMs are overclocked.

Exploring beyond the local budgets. Due to mispredictions, the initial power allotment may become inefficient—some servers may consume less than predicted while others are limited by their budget and cannot overclock VMs to the maximum. Thus, SmartOClock allows sOAs to explore beyond their allocated power budgets. On constrained servers, the sOA tries to gradually exceed the limit through two phases: *exploration* and *exploitation.*

Exploration. A sOA *conditionally* increases its budget by a step size (*e.g.*, 20W) that causes the feedback loop to start increasing the frequency of the overclocked VMs. If within a short timespan (*e.g.*, 30 seconds), the sOA does not receive any *warning* messages from the rack power capping system (run in the rack manager on each rack), then it further increases the budget. The sOA stops when all VMs are overclocked to the highest frequency or when it receives a warning message. The rack manager sends a warning message to all sOAs when the rack's power draw reaches a *warning threshold* (*e.g.*, 95% of the rack's power limit). An sOA ignores the message if it is not exploring. Otherwise, it reduces its budget by the step size and uses *exponential back-off* for the next exploration phase.

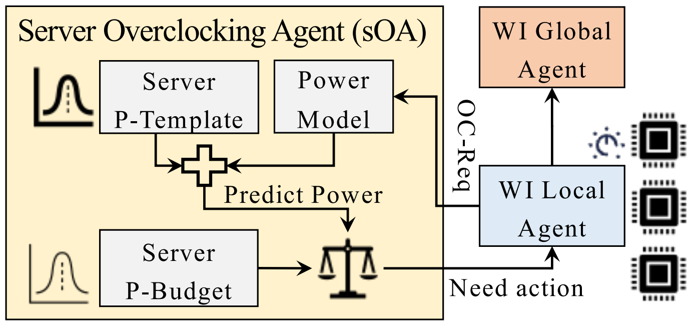


Figure 11.11: Server Overclocking Agent in SmartOClock.

Exploitation. After establishing a safe power budget (*i.e.*, no warning messages), a sOA enters the *exploitation* phase. In this phase, it uses the new budget to grant overclocking requests until either the *time to exploit* expires or upon receiving a *power capping event.* When the time to exploit expires, the sOA starts a new exploration phase if needed. Whereas, on a capping event, it goes back to its initial power budget.

Similarly, a sOA can explore beyond the local per-core overclocking budget. If a VM requires overclocking for longer than its assigned cores can sustain, the sOA will first overclock the VM's assigned cores until their budget is exhausted. Then, the sOA explores rescheduling the VM on other cores in the server with available budget.

Managing resource exhaustion. When an overclocking request is rejected, the global WI agent takes corrective actions per an operator-chosen policy. A simple policy is to scale out while factoring the number of VMs that cannot be overclocked across a deployment (*e.g.*, create x new if y existing VMs cannot be overclocked). Figure 11.11 shows the operations performed by SmartOClock for managing power exhaustion. First, a sOA predicts when it will run out of power for overclocking. For this check, it first predicts the extra power from overclocking a given VM (for a worst-case CPU utilization). Next, via the template, it finds the time when the predicted extra power exceeds the server's budget. It then sends a signal to the global WI agent if the time to exhaustion is within a configurable window (*e.g.*, 15 minutes). To minimize performance impact from a lack of overclocking, the length of the window should be greater than the time to scale out, so that overclocking is still available for the time it takes to scale out. Finally, this operation can be performed ahead of time for scheduled overclocking requests to protect workload SLOs. For metrics-based overclocking, the scale-up (overclocking) threshold can be set before scale-out, where SLOs would be missed if resources are inadequately provisioned after the scale-out threshold is exceeded. Setting an earlier scale-up threshold allows using overclocking to handle load spikes and enables reverting to scale-out if overclocking is not possible. An sOA also predicts the time to exhaustion of the overclocking budget and informs the global WI agent.

11.5 EVALUATION

To evaluate SmartOClock, we perform real-system experiments running cloud applications in an overclockable server cluster, and large-scale analysis using production traces.

11.5.1 Cluster-Level Experiments

Methodology. We implement SmartOClock and conduct the experiments on 36 overclockable servers (all 28 from one rack, and 8 from another during scale-out). Each server has a 64-core (128 threads) AMD EPYC 7763 CPU with customizations to facilitate overclocking experimentation. Its default max turbo frequency is 3.3GHz, which can be increased to 4.0 GHz on these custom parts for overclocking. The CPU is configured to operate in performance mode [476] and the active cores can steadily run at 4.0 GHz while TDP-unconstrained.

To set the load for each server, we take an example production rack from Azure. Based on the power traces of these production servers, we select which application to run in each individual server to mimic the *same* power profile. We run VMs hosting two open-source applications: (1) the latency-critical social network microservices (*SocialNet*) from DeathStarBench [40] and, (2) the throughput-optimized machine learning training (*MLTrain*) from FunctionBench [171]. In the power traces, 14 of the servers show constant high power while the other 14 show a diurnal pattern. For the first 14 servers, we use MLTrain and SocialNet for the other 14. The load for each benchmark instance is configured to mimic the power draw of the corresponding production server.

We define the per-server load in our experiments based on the production traces. As the profiled servers run different, independent, workloads, each server runs an independent set of SocialNet instances. Thus, there is no correlation in the power draw or loads across servers (*i.e.*, the load on one server does not affect the load on others). Auto-scaling is set for SocialNet based on its tail latency (initial count is 14). We set the SLO of each microservice to be 5 times its execution time on an unloaded system [283, 284, 361].

We compare SmartOClock with a *Baseline* system that does not scale horizontally (number of instances) nor vertically (core's frequency), and *ScaleOut* and *ScaleUp* systems that only scale out/in and up/down, respectively, SocialNet instances based on the observed tail. In the evaluation we use a metric-based overclocking policy, which is less predictable; experiments with a schedule-based policy show better results due to higher predictability.

Application performance. Figure 11.12 shows the P99 tail and average latency of SocialNet microservices in four environments. We group the 14 instances into three classes based on their load: Low, Medium, and High Load. Bars in the figure are the average across all instances with the same load level.

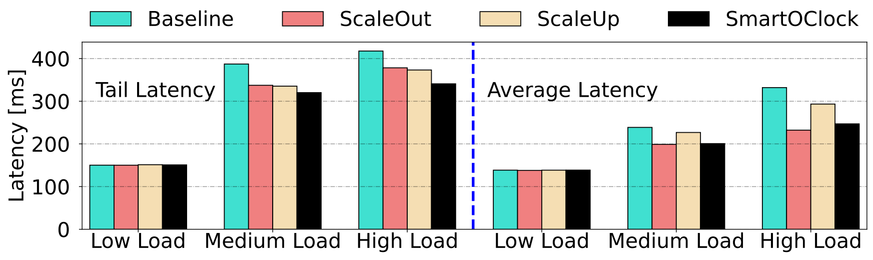


Figure 11.12: P99 tail and average latencies of SocialNet services.

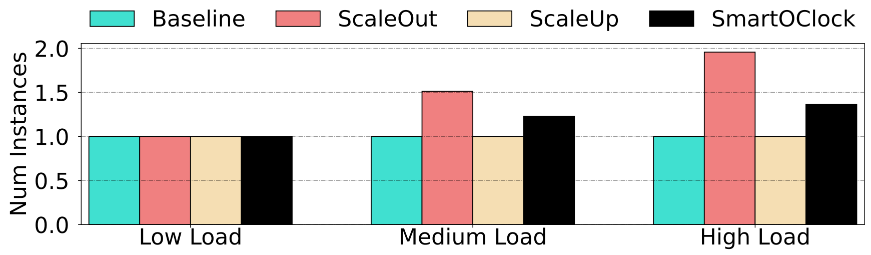


Figure 11.13: Average number of SocialNet VMs varying load.

All systems perform equally well under low load. The impact on tail latency becomes prominent with increased load. Under high load, SmartOClock reduces the tail latency of Baseline, ScaleOut, and ScaleUp by 19.0%, 10.5%, and 8.9%.

The average latency of SmartOClock is lower than Baseline and ScaleUp, but slightly higher than ScaleOut. The reason is that, to reduce the application's cost and prevent scaling out, SmartOClock operates for a longer time with higher latencies that are still below the SLO. However, SmartOClock significantly reduces the number of missed SLOs. The total number of missed SLOs at high load is reduced by 26×, 4.8×, and 2.3× over Baseline, ScaleOut, and ScaleUp, respectively. These results show that overclocking (via ScaleUp or SmartOClock) reduces missed SLOs compared to ScaleOut. However, overclocking alone is insufficient at higher loads as evidenced by the greater missed SLOs with ScaleUp, despite it overclocking for 5x longer. A combination of ScaleUp and ScaleOut via SmartOClock provides the best performance. Finally, SmartOClock reacts fast to sudden workload shifts and keeps the application performance within its SLO: even on servers that triggered overclocking for more than 140 times within 5 minutes, SmartOClock did not miss any deadlines.

Cost. Performance improvements from SmartOClock result in cost savings for the users as they need to pay for fewer VMs. Figure 11.13 shows the average number of concurrently active VM instances for each environment over the entire run. Under high load, SmartOClock saves substantial cost by reducing the number of required instances by 30.4% over ScaleOut.

Energy consumption. Figure 11.14 shows normalized (1) per-single-server energy consumption under low, medium, and high load, and (2) total energy consumption of the system.

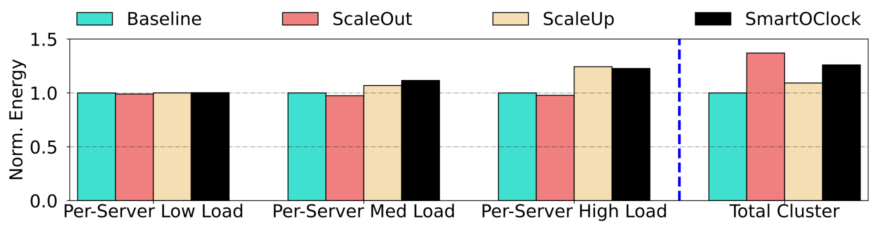


Figure 11.14: Normalized per-single-server energy.

Note that ScaleOut and SmartOClock are the only systems that meet SLOs. As the load increases, SmartOClock frequently overclocks cores, which increases the per-server energy consumption. However, as it uses fewer instances, the total energy consumption is reduced by 10% on average over ScaleOut. The savings are larger if we only consider servers running latency-critical microservices — 23% on average over ScaleOut.

Power-constrained environments. We evaluate SmartOClock's overclocking admission control and heterogeneous power budgeting under constraints. We reduce the rack's limit and measure the performance in two systems: NaiveOClock and SmartOClock. NaiveOClock grants all overclocking requests and on a power capping event splits the rack's budget equally among the servers. SmartOClock reduces the SocialNet tail latency by 6.7% and 8.4% for medium and high loads, respectively, and improves the MLTrain throughput by 10.4%.

Overclocking-constrained environments. To evaluate SmartOClock's proactive scale-out, we restrict the overclocking budget and measure the number of missed SLOs with and without proactive scaling. As we reduce the budget to 75%, 50%, and 25% of its initial value, reactive scale-out misses the SLO for 5.0%, 6.1%, and 7.2% of time, while SmartOClock's proactive approach eliminates all SLO violations.

11.5.2 Large-Scale Simulations

Methodology. We use production traces of dedicated racks running Microsoft's productivity and collaboration services (see Section 11.3) from multiple datacenters. Each datacenter deployment is composed of hundreds of racks and a few thousand servers with either Intel or AMD CPUs. Each workload's VMs are spread across servers and racks. The traces include rack and server power, and VM-level CPU utilization. All data is collected for 6 weeks (April 10^{th} - May 12^{th}, 2023), at a 5-minute granularity. Overclocking requirements (*e.g.*, time of day) are obtained from the workload operators.

We develop a discrete event simulator to evaluate SmartOClock. Models are used to estimate the power impact of overclocking; CPU utilization and core frequency are the input. We validate the model for each server generation.

Table 11.1: Comparison of SmartOClock to different baselines.

System	Norm. # of Power Caps	Good OClock Reqs	Penalty on Power Cap	Norm. Perf.
High-Power Clusters				
Central	1.0	92%	21%	1.186
NaiveOClock	118.6	55%	34%	0.963
NoFeedback	5.5	72%	22%	1.122
NoWarning	27.4	81%	23%	1.081
SmartOClock	6.3	89%	22%	1.164
Medium-Power Clusters				
Central	1.0	96%	11%	1.195
NaiveOClock	36.6	79%	19%	1.022
NoFeedback	3.4	83%	11%	1.163
NoWarning	7.2	87%	12%	1.160
SmartOClock	3.9	93%	11%	1.185
Low-Power Clusters				
Central	1.0	99%	1%	1.208
NaiveOClock	14.0	99%	5%	1.172
NoFeedback	1.0	98%	1%	1.205
NoWarning	1.1	99%	2%	1.205
SmartOClock	1.0	99%	1%	1.208

We compare SmartOClock to *(1) Central* – an oracle with a global view of power draw that can precisely decide if an overclocking request will result in capping, *(2) NaiveOClock* – a system that grants all overclocking requests, *(3) NoFeedback* – a system that adheres to the per-server power budgets with no exploration beyond, and *(4) NoWarning* – a system that allows exploring but with no warnings. The servers go back to their initial power budget on a capping event.

Overclocking success and power capping. Table 11.1 shows the results: (1) number of power capping events in each system normalized to Central, (2) percentage of successful overclocking requests, (3) performance penalty of capping on non-overclocked VMs, and (4) normalized performance over Baseline. We define the performance penalty and improvement as reduction and increase in VM frequency compared to the Baseline (max turbo). Clusters are split into three groups based on power draw: *High*, *Medium*, and *Low-Power*.

First, naively granting overclocking requests causes many power capping events. NaiveOClock causes 118.6×, 36.6×, and 14.0× more events than Central in High, Medium, and Low-Power clusters, respectively. In contrast, SmartOClock lowers the events by 18.9× in High-Power clusters via prediction for admission control. Adding the warning messages efficiently controls overclocking beyond a server's budget: it reduces the number of events over NoWarning by up to 4.3×.

Second, SmartOClock successfully grants majority of overclocking requests. It is within 4%, 3%, and 1% of the success rate of an oracle Central system in High, Medium, and Low-Power clusters, respectively. The feedback-loop for exploring beyond the per-server budget is important: SmartOClock has up to 1.24× higher success rate than NoFeedback approach.

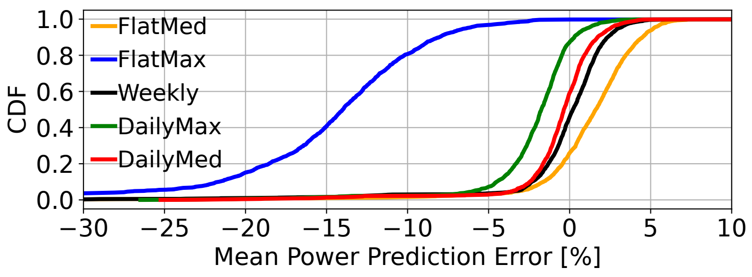


Figure 11.15: CDF of mean power prediction for each technique.

Finally, heterogeneous power distribution by SmartOClock reduces the performance penalty from power capping events. All systems bar NaiveOClock employ this optimization. The heterogeneous power budgets reduce the performance penalty due to power capping events over NaiveOClock by 1.62× and 1.72× in High and Medium-Power clusters, respectively.

Power predictions accuracy. Figure 11.15 shows the CDF of prediction accuracy for computing the power templates. *FlatMed* and *FlatMax* use a constant prediction: a median or maximum of all prior measurements. FlatMed is opportunistic and underpredicts power, leading to high P99 prediction errors. Whereas, FlatMax is conservative and overpredicts power, resulting in negative prediction errors at low percentiles.

Weekly uses a time series of power measurements from the previous week for predictions in the following week. It is impacted by outliers since it treats each day separately: a few hours may behave differently due to the unexpected events. Thus, at high percentiles, its prediction error can be significant.

Finally, *DailyMed* and *DailyMax*, aggregate the power measurements across a week to represent a single, typical, day. The templates are time series of median or maximum values. DailyMed, used in SmartOClock, has the highest accuracy.

11.5.3 Experiments with Production Services

We evaluate overclocking *Service B* and *C* under production load. Each service consumes hundreds of virtual cores across tens of VMs. The deployment resource usage is similar to Figure 11.1 and the SLOs are consistent with each service's goals.

Figure 11.16 shows the average CPU utilization of *Service B*'s VMs for different request rates (bucketized by 0.1 due to live load variability). Overclocking reduces CPU utilization of VMs by 23% at a peak of 1.8k requests per second (RPS); the baseline operates at turbo (3.3 GHz). Alternatively, for the same CPU utilization, baseline can service 1.4k RPS vs 1.8k (28% higher) with overclocking. Figure 11.17 shows that overclocking reduces *Service C*'s 5-minute peaks over a weekday by 16%. The deployment load is similar on both days.

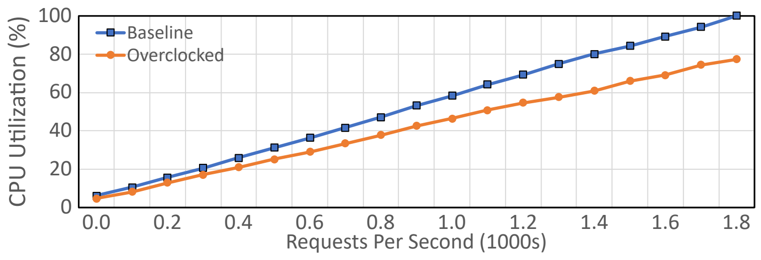


Figure 11.16: Impact of overclocking Service B.

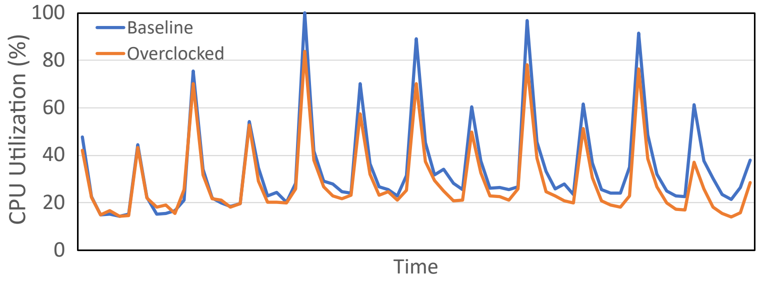


Figure 11.17: Impact of overclocking Service C.

Both results show the opportunity to down-provision while meeting the performance SLOs. Finally, overclocking enables servicing 25% additional load by *Service A* VMs under synthetic traffic; production experiments are being setup.

11.6 LESSONS FROM PRODUCTION DEPLOYMENT

We built a first-of-its-kind 2-rack (56 servers) overclockable cluster at a cloud provider for CPU overclocking in production. Our deployment does not yet include cluster-wide coordination. Here we present lessons from the deployment.

Motivation for building a cluster. Although CPU overclocking can provide substantial performance and cost benefits, a comprehensive analysis (*e.g.*, TCO reduction, revenue increase) is needed for introducing hardware features at scale. Projecting improvements is challenging due to workload-specific variations, as previous work shows [461]. Furthermore, evaluating in a lab is not possible for even Microsoft's internal workloads due to software dependencies (*e.g.*, deployment framework) and security concerns that prevent experimentation with production traffic. Thus, building an overclockable cluster was imperative.

Using experimental hardware in production datacenters. Azure has a rigid process for ensuring stability (*e.g.*, thermal limitations), reliability (*e.g.*, firmware errors), and high performance for hardware deployment at scale that adds overhead for limited-scale experimentation. To address, we retrofitted overclocking onto existing hardware by installing overclockable CPUs and firmware updates (*e.g.*, BIOS) on already-deployed servers in a production datacenter. We also bypassed software checks that remove servers with unexpected configuration. A drawback of our approach is that the platform (*e.g.*, motherboard, cooling) is not optimized for overclocking, leading to thermal and max current throttling under heavy loads that affect performance. Additionally, we provisioned adequate power for the racks to avoid capping; the limits are lowered for power management evaluations.

Experiments with first-party workloads. Since the cluster contains experimental hardware, we enforce strict admission control. However, this policy led to atypical workload placements that impact overclocking. Typically servers house VMs from various workloads due to dynamic cloud environments and scheduler efforts to optimize resource utilization [496, 515], but our policy caused VMs from the same workload to occupy entire servers. Although this impacts overclocking efficiency, it is useful for conservative benefit estimation.

Finer-grained overclocking. SmartOClock can overclock individual VMs but first-party operators want finer-grained overclocking (*e.g.*, containers in VMs). Although overclocking VMs still works, it is inefficient because of the higher power and reliability impact. Since containers are scheduled inside a guest VM without host visibility, we need guest participation for finer-grained overclocking. Unsupervised control of frequency by guests can compromise reliability and power management. We are exploring a safe and efficient solution.

Hardware support for overclocking. Overclocking lifetime budgets can be improved with *wear-out counters* that indicate how a component's (*e.g.*, CPU core) lifetime is impacted by utilization (voltage) and operating temperatures. SmartOClock can use wearout counters to upgrade from a conservative offline model to a *per-part* online calculation for safety.

Furthermore, the prioritized feedback loop for managing power while overclocking can be offloaded to the hardware for efficiency. We are extending the ACPI [514] CPPC interfaces to configure VM priority while scheduling (no affinitization) on CPU cores. The firmware can use these priorities to assign per-core performance (frequency) while managing power.

Vendor engagements to enable overclocking. As overclocking is enabled by underutilization (Section 11.3), instead of overclocking, vendors (*e.g.*, Intel, AMD) inquire about designing a CPU with revised time-in-state assumptions for offline certification. However, this is still inefficient as it does not leverage the utilization variability from workload demands (with and without overclocking) and temperature fluctuations on ageing at cloud scale. Using wear-out counters to track the usage impact on ageing does not have these limitations.

We are working with the vendors to ensure all cores can hit a minimum-desired overclocking frequency (*e.g.*, 15-20% beyond max turbo). Some cores can run faster, but this variability is not exposed on server CPUs (even for turbo); we are exploring bringing mechanisms from client CPUs (*e.g.*, ACPI CPPC preferred cores [514]) to leverage this variability.

Overclocking beyond CPUs. SmartOClock is a general framework and its principles can be easily applied for overclocking any server component. Our initial focus was CPU since it provides the highest benefits, but we have started exploring overclocking of other components (*e.g.*, GPU).

Silent data corruption (SDC). Prior work shows the risks from SDC at scale [516, 517, 518]. Although overclocking can aggravate error rates due to aggressive circuit timing and sudden voltage drops, our extensive lab and production experiments do not show an increase in errors, with frequencies ∼20% beyond max turbo; this is inline with a prior work [461]. Nonetheless, for safety, we work with the vendors to define a max overclocking frequency. Furthermore, techniques from the SDC work can also be used for added safeguarding.

11.7 RELATED WORK

Computational sprinting. Extensive research [462, 463, 464, 465, 466, 467, 468, 497, 519, 520, 521] has explored computational sprinting (*i.e.*, boosting CPU frequency for short periods). Mechanisms like game theory [463], formal control [522], and performance modeling [468] have been proposed to manage sprinting. Researchers have also investigated efficiency factors like resource interference [521], power availability [464], processor design [523], and cooling [461]. However, none of these works holistically address the overclocking challenges in the cloud. They either focus on a single-server setup, assume a transparent-box knowledge of the applications, or overlook multi-tenancy on a server or rack.

The closest related work is Computational Sprinting Game (CSG) [463]. There are two major differences between CSG and SmartOClock. First, CSG leverages turbo and is constrained by thermal/power limits. In contrast, overclocking also affects reliability whose time scales are orders of magnitude (months/years) more than for power/thermal (minutes). It is nontrivial to add reliability under CSG when evaluating sprint utility. In contrast, MosaicCPUuses epochs to divide the overclocking budget across coarse-grain time scales (days) that local agents enforce. Second, a lack of sprinting/overclocking (of even a few VMs) can impact the SLOs of workloads that under-provision while relying on sprinting to handle their peaks. Therefore, a mitigation mechanism to protect performance is needed when sprinting is unavailable, a problem not addressed by CSG. Section 11.5 presents the impact of proactive scaleout by SmartOClock to protect workload SLOs.

Undervolting. Prior work has proposed decreasing the voltage for a frequency below its safe marginal value for reducing power [524, 525, 526, 527, 528, 529]. However, undervolting can introduce instability and pipeline (*i.e.*, timing) errors, thereby necessitating hardware designers to add mechanisms for fault tolerance. For example, Razor [529] uses additional latches that run on a delayed clock in vulnerable paths to detect/recover from errors. This body of work is complementary and can create additional power and component lifetime headroom (reduced wearout from lower voltage) for overclocking.

Datacenter power management. Prior work has proposed oversubscription through leveraging statistical properties of concurrent power usage across servers [469, 470, 471, 472, 473, 474, 530, 531, 532] to improve datacenter power utilization and save costs. These works are complementary and influence our non-overclocked baseline. Azure leverages policies based on these works to oversubscribe power. The policies factor the power demand from turbo for meeting the performance SLAs [482, 483, 484] and prioritized throttling [473, 474] is used to protect (turbo) performance of critical workloads under rare power capping events.

Naively adding overclocking to the baseline power utilization increases the probability of power capping events. Increasing provisioned power cannot be used to address this problem due to the TCO impact, especially when turbo is sufficient to meet a provider's performance SLAs. Consequently, an overclocking system can only leverage unutilized power while meeting workload SLOs when overclocking is not possible; problems not addressed by the prior power management work.

Workload intelligence. Research has leveraged workload awareness to optimize performance, energy consumption, and cost [38, 152, 417, 418, 419, 507, 533, 534]. Sinan [507] uses ML models to allocate resources per microservice tier for minimizing cost while maintaining latency targets. ReTail [417], Rubik [419], Adrenaline [421], and Gemini [418] use application-specific features to predict optimal per-request frequencies, reducing power draw while meeting SLOs. Resource Central [152] gathers VM telemetry, learns VM behaviors offline, and provides online predictions for various resource managers. We propose a clean interface for cloud workloads to provide the necessary signals for overclocking without compromising their opaque-box implementations.

11.8 CONCLUSION

In this chapter, we proposed SmartOClock, the first distributed overclocking management platform for cloud environments. SmartOClock enables cloud providers to offer overclocking to workloads through four novel features: workload intelligence, prediction-based admission control, heterogeneous power budgeting, and decentralized enforcement. Our evaluation shows that SmartOClock reduces the tail latency by 8.9% and the application cost by 30.4%. We also discussed lessons from building an overclockable cluster in Azure. We conclude that carefully-managed overclocking has enormous potential to improve workload performance while saving cost.

CHAPTER 12: Future Work

Based on the insights and findings of this thesis, there are four promising directions for future research aimed at rethinking the design of hyperscale systems to achieve high performance, energy efficiency, and ease of use. Realizing these goals will require a comprehensive re-examination of the full cloud-computing stack and will involve interdisciplinary collaborations across systems and networking, AI/ML, and programming languages. The proposed research agenda is organized into four pillars:

Cloud-Native Servers. A natural direction for future work is to build on the architecture proposed in this thesis by further enhancing its efficiency and adapting it to better meet the demands of cloud-native environments through full-stack co-design. Given the heterogeneous nature of cloud-native services, conventional homogeneous server architectures are often inefficient. Future research should investigate re-configurable and heterogeneous core architectures capable of dynamically adapting to the characteristics of individual services. Additionally, the design of the memory subsystem warrants reevaluation. Due to the relatively small footprint of typical services and substantial memory sharing across service invocations, novel memory hierarchies must be developed. Emerging Compute Express Link (CXL) technologies further open opportunities for efficient inter-server communication through coherent shared memory. Another critical direction is addressing networking overheads through in-network computing and the development of hardware structures that support these capabilities efficiently.

Accelerator-as-a-Service. With the growing performance demands, cloud providers face pressure to offer specialized hardware accelerators for diverse and rapidly-evolving workloads. However, the current model of deploying fixed-function accelerators, each designed for a specific task, results in underutilized resources or the need for expensive reconfigurations. This model struggles to keep pace with the dynamic nature of modern cloud workloads. A promising alternative is to offer a wide range of accelerator templates that can be dynamically customized based on user queries. These templates would act as flexible building blocks, enabling on-demand adaptation to workload requirements. Future research should explore methods to reduce the overhead of on-the-fly specialization, coordinate clusters of accelerators to maximize application speedup, and ensure secure time- and space-sharing of accelerators across multiple users.

Green Clouds. The rising energy consumption and carbon footprint of cloud workloads have raised critical concerns about the long-term sustainability of large-scale cloud infrastructure. This challenge spans diverse application domains and heterogeneous hardware environments. Building on existing energy-efficient frameworks, a key direction for future research is the development of principled techniques for achieving sustainable cloud computing. Emphasis should be placed on optimizing end-to-end applications that span multiple computing paradigms—such as inference queries embedded in microservice architectures—and execute across diverse hardware platforms, including CPUs, GPUs, and accelerators. Further exploration is warranted into the trade-offs between energy-, power-, and carbon-efficiency in datacenter systems, particularly those powered by a heterogeneous mix of renewable and non-renewable energy sources. In addition, the potential of emerging devices and novel materials for processor design should be investigated as a means to further reduce the environmental impact of hyperscale systems.

Cloud-Native Language. As cloud applications continue to scale in complexity and deployment, existing programming models are increasingly strained by the need to balance design simplicity, scalability, and performance. Current approaches require developers to manually choose the granularity of services—where smaller services enable better scalability but introduce significant overhead relative to their core logic. Given that the ideal service decomposition often depends on the underlying hardware and runtime conditions, future research should investigate programming models that dynamically and transparently reorganize applications in response to system state. This includes the development of compiler and runtime infrastructures that co-optimize the behavior of interdependent services and facilitate more intuitive application development through tools such as natural language interfaces. Additionally, a shift from the current function-centric paradigm—where data is moved to compute—toward a data-centric execution model should be explored. In this model, applications are structured as data-flow graphs, with each data element annotated with a function callback, enabling computation to be moved closer to the data source. Such an approach can significantly reduce data movement and its associated performance and energy costs. Advancing this line of work will require specialized language, compiler, and runtime support designed to enhance both the programmability and efficiency of cloud-native workloads.

CHAPTER 13: Conclusions

Cloud-native workloads, such as microservices and serverless computing, have the potential to radically change the landscape of cloud computing by providing a simple programming model, cost-efficient billing, good scalability, and opportunity for high degree of workload consolidation. However, the properties of these workloads are fundamentally different from what conventional hardware and software systems are optimized for. As a result, existing infrastructures suffer from poor performance, low resource utilization, and high energy consumption when running these workloads. The goal of this thesis has been to rethink the design of hardware platforms and software stacks, and enable the execution of cloud-native workloads with orders of magnitude better efficiency.

The first part of the thesis presented a hardware architecture specifically designed for cloud-native workloads. In particular, the thesis started by designing *μManycore*, a processor architecture that minimizes the tail latency of cloud-native services. The architecture has many simple cores augmented with hardware support for scheduling and context switching. The cores are organized into hardware cache-coherent villages and connected via a hierarchical leaf-spine network topology. Then, the processor architecture was extended with *HardHarvest* to boost utilization through hardware-based core harvesting, while the microarchitecture was refined with *Mosaic* for better performance under frequent context switches. Finally, the thesis integrated on-package accelerators into the architecture and proposed *AccelFlow*, a framework that enables fine-grained, low-overhead orchestration of accelerators to reduce datacenter tax overheads in cloud-native environments.

To maximize the efficiency of the proposed hardware architecture for cloud-native environments, the second part of the thesis proposed a software stack that is tightly co-designed with the hardware. It began with *MXFaaS*, a scheme that improves resource utilization by efficiently multiplexing compute, memory, and I/O resources during bursts of same-function invocations. Then, the thesis integrated a distributed software caching layer with the *Concord* coherence protocol to reduce overheads from frequent remote storage accesses. To improve the efficiency of services, the thesis further accelerated end-to-end application workflows using *SpecFaaS*, a system for speculative service execution within an application. Finally, the thesis augmented the platform to improve energy efficiency with *EcoFaaS*, which relies on fine-grained scheduling and dynamic frequency scaling, and *SmartOClock*, which under-provisions resources and selectively overclocks cores during load spikes.

Each of these techniques enables a major advance in the performance, resource, or energy-efficiency of cloud-native workloads.

APPENDIX A: Other Works

Here, I briefly describe other research work that was conducted in parallel to this thesis.

Energy, Power, and Thermal Management for LLM Inference Serving in the Cloud. The rapid evolution and widespread adoption of generative large language models (LLMs) have made them a pivotal workload in various applications. Today, LLM inference clusters receive a large number of queries with strict Service Level Objectives (SLOs). To achieve the desired performance, these models execute on power-hungry GPUs, causing the inference clusters to consume large amounts of energy and, consequently, result in substantial carbon emissions. Moreover, from the datacenter perspective, these infrastructures require the provision of high power and cooling capacities, while running a mix of open-box first-party workloads and opaque-box third-party workloads.

First, I found that there is an opportunity to improve energy efficiency by exploiting the *heterogeneity* in inference compute properties and the *fluctuations* in inference workloads. However, the diversity and dynamicity of these environments create a large search space, where different system configurations (e.g., number of instances, model parallelism, and GPU frequency) translate into different energy-performance trade-offs. To address these challenges, I proposed *DynamoLLM* [535], the first energy-management framework for LLM inference environments. DynamoLLM automatically and dynamically reconfigures the inference cluster to optimize for energy of LLM serving under the services' performance SLOs. At a service level, on average, DynamoLLM conserves 52% of the energy and 38% of the operational carbon emissions, and reduces the cost to the customer by 61%, while meeting the latency SLOs.

Second, I proposed *TAPAS* [536], a thermal- and power-aware framework designed for LLM inference clusters in the cloud. TAPAS enhances cooling and power oversubscription capabilities, reducing the total cost of ownership (TCO) while effectively handling emergencies (e.g., cooling and power failures). TAPAS leverages historical temperature and power data, along with the adaptability of first-party workloads, to: (1) efficiently place new GPU workload VMs within cooling and power constraints, (2) route LLM inference requests across first-party VMs, and (3) reconfigure first-party VMs to manage load spikes and emergency situations. An evaluation on a large GPU cluster demonstrates significant reductions in thermal and power throttling events, boosting system efficiency.

Rethinking the Page Table Design for Virtualized Cloud Environments. A major reason why nested or virtualized address translations are slow is because current systems organize page tables in a multi-level tree that is accessed in a sequential manner. A nested translation may potentially require up to twenty-four sequential memory accesses. To address this problem, I proposed the first page table design that supports parallel nested address translation [537]. The design is based on using hashed page tables (HPTs) for both guest and host. However, directly extending a native HPT design to a nested environment leads to minor gains. Instead, my proposed design solves a new set of challenges that appear in nested environments. The scheme eliminates all but three of the potentially twenty-four sequential steps of a nested translation—while judiciously limiting the number of parallel memory accesses issued to avoid over-consuming cache bandwidth. As a result, compared to conventional nested radix tables, the proposed design speeds-up the execution of a set of applications by an average of 1.19x (for 4KB pages) and 1.24x (when huge pages are used). In addition, I also proposed a migration path from current nested radix page tables to the nested hashed page table design.

After improving the performance of hashed page tables (HPTs), a major issue that remains is the fact that HPTs need substantial *contiguous* physical memory. My subsequent work, *Memory-Efficient HPTs* (ME-HPTs) [538] addresses this problem. To minimize HPTs' contiguous memory needs, ME-HPTs introduce the *Logical to Physical (L2P) Table* and the use of *Dynamically-Changing Chunk Sizes*. These techniques break down the HPT into discontiguous physical-memory chunks. In addition, ME-HPTs also introduce two techniques that minimize HPTs' total memory needs and, indirectly, reduce the memory contiguity requirements. These techniques are *In-place Page Table Resizing* and *Per-way Resizing*. Compared to state-of-the-art HPTs, ME-HPTs: (i) reduce the contiguous memory allocation needs by 92% on average, and (ii) improve the performance by 8.9% on average. For the two most demanding workloads, the contiguous memory requirements decrease from 64MB to 1MB. In addition, compared to state-of-the-art radix-tree page tables, ME-HPTs achieve an average speedup of 1.23× (without huge pages) and 1.28× (with huge pages).

REFERENCES

[1] C. Richardson, "What are microservices?" 2023. [Online]. Available: https://microservices.io/

[2] A. Sriraman and T. F. Wenisch, "µTune: Auto-Tuned threading for OLDI microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, 2018.

[3] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*, 2016.

[4] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. Candela, "Practical Lessons from Predicting Clicks on Ads at Facebook," in *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, 2014.

[5] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[6] Old GigaOm, "The biggest thing Amazon got right: The platform," 2011. [Online]. Available: https://old.gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/

[7] K. Varshneya, "Understanding design of microservices architecture at Netflix," 2021. [Online]. Available: https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/

[8] Think Software, "Microservices Architecture of Twitter Service," 2021. [Online]. Available: https://thinksoftware.medium.com/design-twitter-microservices-architecture-of-twitter-service-996ddd68e1ca

[9] Uber, "Introducing Domain-Oriented Microservice Architecture," 2020. [Online]. Available: https://www.uber.com/blog/microservice-architecture/

[10] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The Power of Prediction: Microservice Auto Scaling via Workload Learning," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '22)*, 2022.

[11] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "CRISP: Critical Path Analysis of Large-Scale Microservice Architectures," in *USENIX Annual Technical Conference (USENIX ATC '22)*, 2022.

[12] M. Chabbi and M. K. Ramanathan, "A Study of Real-World Data Races in Golang," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*, 2022.

[13] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *IEEE International Symposium on High Performance Computer Architecture (HPCA '18)*, 2018.

[14] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale," in *Proceedings of the 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA '19)*, 2019.

[15] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[16] Google Cloud, "What is Microservices Architecture?" 2023. [Online]. Available: https://cloud.google.com/learn/what-is-microservices-architecture

[17] Kubernetes, " Production-Grade Container Orchestration." [Online]. Available: https://kubernetes.io/

[18] Docker, "Docker Compose." [Online]. Available: https://docs.docker.com/compose/

[19] Amazon AWS, "AWS Lambda." [Online]. Available: https://aws.amazon.com/lambda/

[20] Microsoft, "Microsoft Azure Functions." [Online]. Available: https://azure.microsoft.com/en-gb/services/functions/

[21] IBM, "IBM Cloud Functions." [Online]. Available: https://cloud.ibm.com/functions/

[22] Google, "Google Cloud Functions." [Online]. Available: https://cloud.google.com/functions

[23] S. Wilson and D. Pickering, "A Guide to Developing Serverless Ecommerce Workflows with AWS Lambda," 2021. [Online]. Available: https://aws.amazon.com/blogs/industries/a-guide-to-developing-serverless-ecommerce-workflows-for-commercetools-with-aws-lambda/

[24] S. Kumar and A. Puthiyavettle, "Architecting a Highly Available Serverless Ecommerce Site." [Online]. Available: https://aws.amazon.com/blogs/architecture/architecting-a-highly-available-serverless-microservices-based-ecommerce-site/

[25] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, 2017.

[26] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proceedings of the 2018 ACM Symposium on Cloud Computing (SOCC '18)*, 2018.

[27] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "INFless: A Native Serverless System for Low-latency, High-throughput Inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[28] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated Model-less Inference Serving," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[29] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards Demystifying Serverless Machine Learning Training," in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, 2021.

[30] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 2019.

[31] J. Stojkovic, C. Liu, M. Shahbaz, and J. Torrellas, "µManycore: A Cloud-Native CPU for Tail at Scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[32] J. Stojkovic, C. Liu, M. Shahbaz, and J. Torrellas, "HardHarvest: Hardware-Supported Core Harvesting for Microservices," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*, 2025.

[33] J. Stojkovic, E. Saurez, E. Choukse, I. Goiri, and J. Torrellas, "Mosaic: Harnessing the Micro-architectural Resources of Servers in Serverless Environments," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*, 2024.

[34] J. Stojkovic, A. Farrell, C. Hughes, Z. Gong, and J. Torrellas, "AccelFlow: Decentralized Orchestration of Distributed Accelerators in Microservice Environments," in *Submission*, 2025.

[35] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[36] J. Stojkovic, C. Alverti, A. Andrade, N. Iliakopoulou, T. Xu, H. Franke, and J. Torrellas, "Concord: Rethinking Distributed Coherence for Software Caches in Serverless Environments," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '25)*, 2025.

[37] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '23)*, 2023.

[38] J. Stojkovic, N. Iliakopoulou, T. Xu, H. Franke, and J. Torrellas, "EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*, 2024.

[39] J. Stojkovic, P. Misra, Goiri, S. Whitlock, E. Choukse, M. Das, C. Bansal, J. Lee, Z. Sun, H. Qiu, R. Zimmermann, S. Samal, B. Warrier, A. Raniwala, and R. Bianchini, "SmartOClock: Workload- and Risk-Aware Overclocking in the Cloud," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*, 2024.

[40] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[41] gRPC, "An RPC library and framework," 2023. [Online]. Available: https://github.com/grpc/grpc

[42] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be General and Fast," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[43] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis, "The NEBULA RPC-Optimized Architecture," in *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*, 2020.

[44] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC Tax in Datacenters," in *2021 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021.

[45] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[46] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown, "The nanoPU: A Nanosecond Network Stack for Datacenters," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*, 2021.

[47] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[48] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Maziéres, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[49] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the Killer Microsecond," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '18)*, 2018.

[50] A. Sriraman and T. F. Wenisch, "µSuite: A Benchmark Suite for Microservices," in *IEEE International Symposium on Workload Characterization (IISWC '18)*, 2018.

[51] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing Server Efficiency in the Face of Killer Microseconds," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*, 2019.

[52] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch, "Express-Lane Scheduling and Multithreading to Minimize the Tail Latency of Microservices," in *2019 IEEE International Conference on Autonomic Computing (ICAC '19)*, 2019.

[53] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[54] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud '16)*, 2016.

[55] "OpenFaaS." [Online]. Available: https://docs.openfaas.com/

[56] "Kubeless: The Kubernetes Native Serverless Framework." [Online]. Available: https://kubeless.io/

[57] "Apache OpenWhisk." [Online]. Available: https://openwhisk.apache.org/

[58] "Fn Project." [Online]. Available: https://fnproject.io/

[59] "Fission: Open source Kubernetes-native Serverless Framework." [Online]. Available: https://fission.io/

[60] "Knative." [Online]. Available: https://knative.dev/docs/

[61] "OpenWhisk Python Runtime." [Online]. Available: https://github.com/apache/openwhisk-runtime-python

[62] AWS, "AWS Lambda: Comparing the Effect of Global Scope." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/global-scope.html

[63] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.

[64] A. Fuerst and P. Sharma, "FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[65] "Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues." [Online]. Available: https://www.serverless.com/blog/keep-your-lambdas-warm

[66] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[67] Cisco, "Cisco Spine and Leaf Architecture," 2023. [Online]. Available: https://ciscolicense.com/blog/cisco-spine-and-leaf-architecture/

[68] Engineering at Meta, "Introducing data center fabric, the next-generation Facebook data center network," 2023. [Online]. Available: https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/

[69] Oracle, "MySQL," 2023. [Online]. Available: https://www.mysql.com

[70] The Apache Software Foundation, "Apache Cassandra," 2023. [Online]. Available: https://cassandra.apache.org/

[71] The Apache Software Foundation, "Apache Kafka," 2023. [Online]. Available: https://kafka.apache.org/

[72] Wordpress, "Blog Tool, Publishing Platform, and CMS," 2023. [Online]. Available: https://wordpress.org/

[73] Clang, "A C language family frontend for LLVM," 2023. [Online]. Available: https://clang.llvm.org

[74] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning," 2021.

[75] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture (HPCA '01)*, 2001.

[76] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.

[77] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021.

[78] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking Microservice Systems for Software Engineering Research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*, 2018.

[79] The Apache Software Foundation, "Apache Thrift." [Online]. Available: https://thrift.apache.org/

[80] Spring Framework, "RestController." [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html

[81] Golang, "httpPackage." [Online]. Available: https://pkg.go.dev/net/http

[82] X. Zhang, X. Zheng, Z. Wang, H. Yang, Y. Shen, and X. Long, "High-Density Multi-Tenant Bare-Metal Cloud," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[83] K. Wang, C. Wang, T. Jia, K. Chow, Y. Wen, Y. Dou, G. Xu, C. H. Hou, J. Yao, L. Z. Zhang, and Y. L. Li, "Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis," in *51st International Conference on Parallel Processing (ICPP '22)*, 2022.

[84] Google, "gVisor: Container Runtime Sandbox," 2023. [Online]. Available: https://gvisor.dev/docs/

[85] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, 2020.

[86] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[87] S. Song, T. A. Khan, S. M. Shahri, A. Sriraman, N. K. Soundararajan, S. Subramoney, D. A. Jiménez, H. Litz, and B. Kasikci, "Thermometer: Profile-Guided BTB Replacement for Data Center Applications," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*, 2022.

[88] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," in *2010 IEEE International Solid-State Circuits Conference - (ISSCC '10)*, 2010.

[89] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-Weight Communications on Intel's Single-Chip Cloud Computer Processor," *SIGOPS Operating Systems Review*, 2011.

[90] B. Flachs, S. Asano, S. Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. Brokenshire, M. Peyravian, V. To, and E. Iwata, "The microarchitecture of the Synergistic Processor for a Cell Processor," *IEEE Journal of Solid-State Circuits*, 2006.

[91] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[92] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun, "Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs," in *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019 (APNet '19)*, 2019.

[93] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014.

[94] Linux, "Thread Struct," 2023. [Online]. Available: https://elixir.bootlin.com/linux/v5.17/source/arch/\UrlBreaksx86/include/asm/processor.h\#L467

[95] Linux, "Pt Regs," 2023. [Online]. Available: https://elixir.bootlin.com/linux/v5.17/source/arch/\UrlBreaksx86/include/asm/ptrace.h\#L59

[96] ARM, "ARM Cortex A15," 2023. [Online]. Available: https://developer.arm.com/Processors/Cortex-A15

[97] Intel, "Intel Xeon Platinum 8380 Processor." [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz.html

[98] A. F. Rodrigues, J. Cook, E. Cooper-Balis, K. S. Hemmert, C. Kersey, R. Riesen, P. Rosenfeld, R. Oldfield, and M. Weston, "The Structural Simulation Toolkit," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '10)*, 2006.

[99] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, 2011.

[100] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. 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)*, 2005.

[101] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization (TACO '17)*, 2017.

[102] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*, 2009.

[103] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration the VLSI journal*, 2017.

[104] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, "Kite: A Family of Heterogeneous Interposer Topologies Enabled via Accurate Interconnect Modeling," in *2020 57th ACM/IEEE Design Automation Conference (DAC '20)*, 2020.

[105] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy, "Efficient Scheduling Policies for Microsecond-Scale Tasks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 22)*, 2022.

[106] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: A Microkernel Approach to Host Networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, 2019.

[107] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, “Caladan: Mitigating Interference at Microsecond Timescales,” in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’20)*, 2020.

[108] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, “Centralized Core-Granular Scheduling for Serverless Functions,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC ’19)*, 2019.

[109] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin, “RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’20)*, 2020.

[110] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, “GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP ’21)*, 2021.

[111] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam, “The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP ’21)*, 2021.

[112] D. Wentzlaff and A. Agarwal, “Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores,” *SIGOPS Oper. Syst. Rev.*, 2009.

[113] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. 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)*, 2009.

[114] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou, “Dagger: Towards Efficient RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs,” *IEEE Computer Architecture Letters*, 2020.

[115] J. Zhao, I. Uwizeyimana, K. Ganesan, M. C. Jeffrey, and N. E. Jerger, “ALTOCUMULUS: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO ’22)*, 2022.

[116] M. Khairy, A. Alawneh, A. Barnes, and T. G. Rogers, “SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO ’22)*, 2022.

[117] Nokia Networks, “Event Machine on ODP,” 2023. [Online]. Available: https://openeventmachine.github.io/em-odp/

[118] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible Architectural Support for Fine-Grain Scheduling," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, 2010.

[119] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*, 2007.

[120] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck, "HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*, 2011.

[121] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021.

[122] T. Wang, F. Feng, S. Xiang, Q. Li, and J. Xia, "Application Defined On-chip Networks for Heterogeneous Chiplets: An Implementation Perspective," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*, 2022.

[123] P. Ehrett, T. Austin, and V. Bertacco, "Chopin: Composing Cost-Effective Custom Chips with Algorithmic Chiplets," in *2021 IEEE 39th International Conference on Computer Design (ICCD '21)*, 2021.

[124] Y. Wu, L. Wang, X. Wang, J. Han, J. Zhu, H. Jiang, S. Yin, S. Wei, and L. Liu, "Upward Packet Popup for Deadlock Freedom in Modular Chiplet-Based Systems," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*, 2022.

[125] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-chip-module GPUs for continued performance scalability," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA '17)*, 2017.

[126] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*, 2019.

[127] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. Shoaib Bin Altaf, N. Enright Jerger, and G. H. Loh, "Modular Routing Design for Chiplet-Based Systems," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*, 2018.

[128] Microsoft Azure, "Azure Burstable VMs," 2024. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable

[129] Amazon AWS, "AWS Burstable performance instances." [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstableperformance-instances.html

[130] Microsoft Azure, "Use Azure Spot Virtual Machines," 2024. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/spot-vms

[131] Amazon AWS, "Amazon EC2 Spot Instances." [Online]. Available: https://aws.amazon.com/ec2/spot

[132] Google Cloud, "Spot Virtual Machines," 2024. [Online]. Available: https://cloud.google.com/spot-vms

[133] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini, "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[134] Y. Wang, K. Arya, M. Kogias, M. Vanga, A. Bhandari, N. J. Yadwadkar, S. Sen, S. Elnikety, C. Kozyrakis, and R. Bianchini, "SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud," in *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*, 2021.

[135] A. Fuerst, S. Novaković, I. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. 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)*, 2022.

[136] Y. Zhang, Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and Cheaper Serverless Computing on Harvested Resources," in *Proceedings of the International Symposium on Operating Systems Principles (SOSP '21)*, 2021.

[137] S. Volos, C. Fournet, J. Hofmann, B. Köpf, and O. Oleksenko, "Principled Microarchitectural Isolation on Cloud CPUs," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2024.

[138] M. Godfrey and M. Zulkernine, "A Server-Side Solution to Cache-Based Side-Channel Attacks in the Cloud," in *Proceedings of the IEEE Sixth International Conference on Cloud Computing (CLOUD '13)*, 2013.

[139] M.-M. Bazm, M. Lacoste, M. Südholt, and J.-M. Menaud, "Side-channels beyond the cloud edge: New isolation threats and solutions," in *Proceedings of the 1st Cyber Security in Networking Conference (CSNet '17)*, 2017.

[140] B. A. Braun, S. Jana, and D. Boneh, "Robust and Efficient Elimination of Cache and Timing Side Channels," *CoRR*, vol. abs/1506.00189, 2015. [Online]. Available: http://arxiv.org/abs/1506.00189

[141] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, 2013.

[142] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based Defenses against Cross-VM Side-channels," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, 2014.

[143] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. Campbell, "Scheduling, Isolation, and Cache Allocation: A Side-Channel Defense," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E '18)*, 2018.

[144] S. Volos and B. Kopf, "Preventing side-channels in the cloud," 2024. [Online]. Available: https://www.microsoft.com/en-us/research/blog/preventing-side-channels-in-the-cloud/

[145] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015.

[146] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '11)*, 2011.

[147] M. Wagenländer, L. Mai, G. Li, and P. Pietzuch, "Spotnik: Designing Distributed Machine Learning for Transient Cloud Resources," in *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '20)*, 2020.

[148] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini, "History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[149] Microsoft Azure, "Virtual Machine series." [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/

[150] Amazon AWS, "Amazon EC2 Instance Types." [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[151] Google Cloud, "Machine families resource and comparison guide." [Online]. Available: https://cloud.google.com/compute/docs/machine-resource

[152] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms,” in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP ’17)*, 2017.

[153] Kernel Virtual Machine, “https://www.linux-kvm.org.”

[154] Felix Cloutier, “WBINVD — Write Back and Invalidate Cache,” 2024. [Online]. Available: https://www.felixcloutier.com/x86/wbinvd

[155] Intel, “Intel® CAT: Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family,” 2016. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html

[156] Software Freedom Conservancy, “QEMU: A generic and open source machine emulator and virtualizer,” April 2024. [Online]. Available: https://www.qemu.org/

[157] KVM Hypervisor, “Linux KVM Hypercalls,” 2024. [Online]. Available: https://www.kernel.org/doc/html/v5.9/virt/kvm/hypercalls.html

[158] KVM Hypervisor, “KVM Lock Overview,” 2024. [Online]. Available: https://docs.kernel.org/virt/kvm/locking.html

[159] KVM Hypervisor, “KVM/Linux Kernel Scheduler,” 2024. [Online]. Available: https://elixir.bootlin.com/linux/v4.14/source/kernel/sched/core.c#L479

[160] gRPC, “Completion Queue: Next(),” 2024. [Online]. Available: https://grpc.github.io/grpc/cpp/classgrpc_1_1_completion_queue.html#a86d9810ced694e50f7987ac90b9f8c1a

[161] Apache Thrift, “TServerSocket: Listen(),” 2024. [Online]. Available: https://github.com/apache/thrift/blob/1252cf3a2f3b1d942c8c4713ed7b2cf35c64e547/lib/cpp/src/thrift/transport/TServerSocket.cpp#L381

[162] DeathStarBench, “TextService Initialization.” [Online]. Available: https://github.com/delimitrou/DeathStarBench/blob/master/socialNetwork/src/TextService/TextService.cpp#L59

[163] DeathStarBench, “RateService Initialization.” [Online]. Available: https://github.com/delimitrou/DeathStarBench/blob/master/hotelReservation/services/rate/server.go#L81

[164] DeathStarBench, “TextService Handler.” [Online]. Available: https://github.com/delimitrou/DeathStarBench/blob/master/socialNetwork/src/TextService/TextHandler.h#L41

[165] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, “Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors,” in *International Symposium on Microarchitecture (MICRO)*, November 2002.

[166] L. Hammond, M. Willey, and K. Olukotun, “Data Speculation Support for a Chip Multiprocessor,” in *Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[167] I. Cutress, “The Ice Lake Benchmark Preview: Inside Intel’s 10nm.” [Online]. Available: https://www.anandtech.com/show/14664/testing-intel-ice-lake-10nm/2

[168] I. Cutress, “Examinining Intel’s Ice Lake Processors: Taking a Bite of the Sunny Cove Microarchitecture.” [Online]. Available: https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3

[169] Wikipedia, “Sunny Cove.” [Online]. Available: https://en.wikipedia.org/wiki/Sunny_Cove_(microarchitecture)

[170] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, “GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’15)*, 2015.

[171] J. Kim and K. Lee, “FunctionBench: A Suite of Workloads for Serverless Cloud Function Service,” in *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD ’19)*, 2019.

[172] T. Palit, Y. Shen, and M. Ferdman, “Demystifying Cloud Benchmarking,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS ’16)*, 2016.

[173] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, “BioBench: A Benchmark Suite of Bioinformatics Applications,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS ’05)*, 2005.

[174] C. Katsakioris, C. Alverti, K. Nikas, D. Siakavaras, S. Psomadakis, and N. Koziris, “FaaSRail: Employing Real Workloads to Generate Representative Load for Serverless Research,” in *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, 2024.

[175] D. Ustiugov, D. Park, L. Cvetković, M. Djokic, H. Hè, B. Grot, and A. Klimovic, “Enabling In-Vitro Serverless Systems Research,” in *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, 2023.

[176] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, “High performance cache replacement using re-reference interval prediction (RRIP),” in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA ’10)*, 2010.

[177] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*, 2016.

[178] K. Nguyen, "Code and Data Prioritization - Introduction and Usage Models in the Intel® Xeon® Processor E5 v4 Family," 2016. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-code-and-data-prioritization-with-usage-models.html

[179] Z. Luo, S. Fu, E. Amaro, A. Ousterhout, S. Ratnasamy, and S. Shenker, "Out of Hand for Hardware? Within Reach for Software!" in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS '23)*, 2023.

[180] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, "Accelerating Serverless Computing by Harvesting Idle Resources," in *Proceedings of the ACM Web Conference 2022 (WWW '22)*, 2022.

[181] A. Suresh and A. Gandhi, "ServerMore: Opportunistic Execution of Serverless Functions in the Cloud," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[182] G. Sadeghian, M. Elsakhawy, M. Shahrad, J. Hattori, and M. Shahrad, "UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '23)*, 2023.

[183] D. Tullsen and J. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture (MICRO '01)*, 2001.

[184] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*, 2019.

[185] A. Mirhosseini, H. Golestani, and T. F. Wenisch, "HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.

[186] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker, "How Does It Function? Characterizing Long-Term Trends in Production Serverless Workloads," in *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*, 2023.

[187] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming Serverless Functions Better with Heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[188] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, “SEUSS: Skip Redundant Paths to Make Serverless Fast,” in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys ’20)*, 2020.

[189] Z. Jia and E. Witchel, “Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’21)*, 2021.

[190] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, “WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows,” in *Proceedings of the 2022 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE ’22)*, 2022.

[191] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, “Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications,” in *Proceedings of the ACM Symposium on Cloud Computing (SOCC ’21)*, 2021.

[192] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, “FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’22)*, 2022.

[193] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, “Faastlane: Accelerating Function-as-a-Service Workflows,” in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC ’21)*, 2021.

[194] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic Ephemeral Storage for Serverless Analytics,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI ’18)*, 2018.

[195] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, “InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache,” in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST ’20)*, 2020.

[196] M. Shahrad, J. Balkind, and D. Wentzlaff, “Architectural Implications of Function-as-a-Service Computing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[197] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, “Lukewarm Serverless Functions: Characterization and Optimization,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA ’22)*, 2022.

[198] D. Schall, A. Sandberg, and B. Grot, “Warming Up a Cold Front-End with Ignite,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’23)*, 2023.

[199] Intel, “Products formerly SAPPHIRE RAPIDS,” April 2024. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/codename/126212/products-formerly-sapphire-rapids.html

[200] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, “rFaaS: Enabling High Performance Serverless with RDMA and Leases,” in *Proceedings of the 37th IEEE Interational Parallel and Distributed Processing Symposium (IPDPS ’23)*, 2023.

[201] The vHive Ecosystem, “vSwarm - Serverless Benchmarking Suite,” April 2024. [Online]. Available: https://github.com/vhive-serverless/vSwarm

[202] M. Yu, T. Cao, W. Wang, and R. Chen, “Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing,” in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’23)*, 2023.

[203] Serverless, “Use cases,” April 2024. [Online]. Available: https://www.serverless.com/learn/use-cases/

[204] S. Eismann, J. Scheuner, E. V. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, “A Review of Serverless Use Cases and their Characteristics,” *CoRR*, vol. abs/2008.11110, 2020. [Online]. Available: https://arxiv.org/abs/2008.11110

[205] Brittany King, “Top use cases for serverless computing,” April 2024. [Online]. Available: https://www.digitalocean.com/blog/top-use-cases-for-serverless-computing

[206] Firecracker, “Firecracker design,” 2022. [Online]. Available: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md

[207] W. Jordan, “The serverless server,” 2022. [Online]. Available: https://fly.io/blog/the-serverless-server/

[208] Intel, “Intel RDT Software Package,” 2024. [Online]. Available: https://github.com/intel/intel-cmt-cat/tree/master

[209] A. Seznec, “TAGE-SC-L Branch Predictors,” in *JILP - Championship Branch Prediction*, Minneapolis, United States, June 2014. [Online]. Available: https://inria.hal.science/hal-01086920

[210] R. Starc, T. Kuchler, M. Giardino, and A. Klimovic, “Serverless? RISC more!” in *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies (SESAME ’24)*, 2024.

[211] ARM, “ARM Cortex A15,” April 2024. [Online]. Available: https://developer.arm.com/Processors/Cortex-A15

[212] ARM, “Processing Architecture for Power Efficiency and Performance,” April 2024. [Online]. Available: https://www.arm.com/technologies/big-little

[213] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’19)*, 2019.

[214] MongoDB, “Serverless Application Development,” April 2024. [Online]. Available: https://www.mongodb.com/solutions/use-cases/serverless

[215] H. Xia, D. Zhang, W. Liu, I. Haller, B. Sherwin, and D. Chisnall, “A Secret-Free Hypervisor: Rethinking Isolation in the Age of Speculative Vulnerabilities,” in *Proceedings of the IEEE Symposium on Security and Privacy (SP ’22)*, 2022.

[216] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, “Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization,” in *Proceedings of the IEEE 25th Computer Security Foundations Symposium (CSF ’12)*, 2012.

[217] VMware vSphere, “Sharing Memory Across Virtual Machines,” April 2024. [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-F9111E35-E197-46EC-8350-77827A5A2DEC.html#GUID-F9111E35-E197-46EC-8350-77827A5A2DEC

[218] J. Lindemann and M. Fischer, “A memory-deduplication side-channel attack to detect applications in co-resident virtual machines,” in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC ’18)*, 2018.

[219] S.-H. Yang, M. Powell, B. Falsafi, and T. Vijaykumar, “Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay,” in *Proceedings of the Eighth International Symposium on High Performance Computer Architecture (HPCA ’02)*, 2002.

[220] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, “Drowsy caches: Simple techniques for reducing leakage power,” in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA ’02)*, 2002.

[221] J. H. Yahya, H. Volos, D. B. Bartolini, G. Antoniou, J. S. Kim, Z. Wang, K. Kalaitzidis, T. Rollet, Z. Chen, Y. Geng, O. Mutlu, and Y. Sazeides, “AgileWatts: An Energy-Efficient CPU Core Idle-State Architecture for Latency-Sensitive Server Applications,” in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO ’22)*, 2022.

[222] Python Scikit-learn, “RandomForestClassifier,” April 2024. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.htmll

[223] Python Scikit-learn, “MultiOutputClassifier,” April 2024. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputClassifier.htmll

[224] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, “Hermod: principled and practical scheduling for serverless functions,” in *Proceedings of the 13th Symposium on Cloud Computing (SoCC ’22)*, 2022.

[225] K. Seemakhupt, B. E. Stephens, S. Khan, S. Liu, H. Wassel, S. H. Yeganeh, A. C. Snoeren, A. Krishnamurthy, D. E. Culler, and H. M. Levy, “A Cloud-Scale Characterization of Remote Procedure Calls,” in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP ’23)*, 2023.

[226] Azure, “Azure Public Dataset.” [Online]. Available: https://github.com/Azure/AzurePublicDataset

[227] AWS, “AWS Samples: AWS Serverless Workshops.” [Online]. Available: https://github.com/aws-samples/aws-serverless-workshops/

[228] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, “Characterizing Serverless Platforms with ServerlessBench,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC ’20)*, 2020.

[229] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, “SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing,” in *Proceedings of the 22nd International Middleware Conference (Middleware ’21)*, 2021.

[230] S. Chen, C. Delimitrou, and J. F. Martinez, “PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’19)*, 2019.

[231] R. Chen, H. Shi, Y. Li, X. Liu, and G. Wang, “OLPart: Online Learning Based Resource Partitioning for Colocating Multiple Latency-Critical Jobs on Commodity Computers,” in *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys ’23)*, 2023.

[232] Y. Zhang, J. Chen, X. Jiang, Q. Liu, I. M. Steiner, A. J. Herdrich, K. Shu, R. Das, L. Cui, and L. Jiang, “LIBRA: Clearing the Cloud Through Dynamic Memory Bandwidth Management,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA ’21)*, 2021.

[233] R. B. Roy, T. Patel, and D. Tiwari, "SATORI: Efficient and Fair Resource Partitioning by Sacrificing Short-Term Benefits for Long-Term Gains," in *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021.

[234] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[235] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*, 2020.

[236] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*, 2020.

[237] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, "CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.

[238] H. Wang, I. Koren, and C. M. Krishna, "An adaptive resource partitioning algorithm for SMT processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, 2008.

[239] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "L1-bandwidth aware thread allocation in multicore SMT processors," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*, 2013.

[240] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*, 1996.

[241] S. Raasch and S. Reinhardt, "The impact of resource partitioning on SMT processors," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, 2003.

[242] J. Ahn, C. H. Park, and J. Huh, "Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '14)*, 2014.

[243] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015.

[244] Google, "Protocol Buffers." [Online]. Available: https://protobuf.dev/

[245] A. Gonzalez, A. Kolli, S. Khan, S. Liu, V. Dadu, S. Karandikar, J. Chang, K. Asanovic, and P. Ranganathan, "Profiling Hyperscale Big Data Processing," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[246] J. Boo, Y. Chung, E. Baek, S. Na, C. Kim, and J. Kim, "F4T: A Fast and Flexible FPGA-based Full-stack TCP Acceleration Framework," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[247] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL Acceleration with Commodity Processors," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, 2011.

[248] X. Hu, C. Wei, J. Li, B. Will, P. Yu, L. Gong, and H. Guan, "QTLS: high-performance TLS asynchronous offload framework with Intel® QuickAssist technology," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*, 2019.

[249] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A Hardware Accelerator for Protocol Buffers," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021.

[250] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus Prime: Accelerating Data Transformation in Servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[251] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee, "A Specialized Architecture for Object Serialization with Applications to Big Data Analytics," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*, 2020.

[252] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: towards zero-copy serialization," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, 2021.

[253] S. Karandikar, A. N. Udipi, J. Choi, J. Whangbo, J. Zhao, S. Kanev, E. Lim, J. Alakuijala, V. Madduri, Y. S. Shao, B. Nikolic, K. Asanovic, and P. Ranganathan, "CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[254] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke, H. Myneni, and C. Wang, "Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*, 2020.

[255] Intel, "Intel® QAT: Performance, Scale, and Efficiency." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html

[256] S. Kanev, S. L. Xi, G.-Y. Wei, and D. Brooks, "Mallacc: Accelerating Memory Allocation," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[257] Intel, "Intel® Dynamic Load Balancer." [Online]. Available: https://www.intel.com/content/www/us/en/download/686372/intel-dynamic-load-balancer.html

[258] H. Seyedroudbari, S. Vanavasam, and A. Daglis, "Turbo: SmartNIC-enabled Dynamic Load Balancing of µs-scale RPCs," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '23)*, 2023.

[259] M. Hill and V. J. Reddi, "Gables: A Roofline Model for Mobile SoCs," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*, 2019.

[260] S. Gupta and S. Dwarkadas, "RELIEF: Relieving Memory Pressure In SoCs Via Data Movement-Aware Accelerator Scheduling," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '24)*, 2024.

[261] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture Support for Domain-Specific Accelerator-Rich CMPs," *ACM Transactions on Embedded Computing Systems*, 2014.

[262] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "VIP: Virtualizing IP chains on handheld platforms," in *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015.

[263] T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, and J. Balkind, "Cohort: Software-Oriented Acceleration for Heterogeneous SoCs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23)*, 2023.

[264] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[265] OpenSSL, "Cryptography and and SSL/TLS Toolkit." [Online]. Available: https://www.openssl.org/

[266] Facebook, "Zstandard." [Online]. Available: https://facebook.github.io/zstd/

[267] Google, "Snappy, a fast compressor/decompressor." [Online]. Available: https://github.com/google/snappy

[268] N. Sun and P. Bhattacharya, "Using the Cryptographic Accelerator of the UltraSPARC T1 Processor," March 2006. [Online]. Available: https://www.oracle.com/technetwork/server-storage/solaris/documentation/819-5782-150147.pdf

[269] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, 2012.

[270] Y. Yuan, J. Hu, R. Wang, N. Ranganathan, R. Kuper, I. Jeong, and N. S. Kim, "On-chip Accelerators in 4th Gen Intel Xeon Scalable Processors: Features, Performance, Use Cases, and Future!" ISCA'23 Tutorial, 2023.

[271] Intel, "Intel® Xeon® Processors," April 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/processors/xeon.html

[272] Y. Gan, Y. Qiu, L. Chen, J. Leng, and Y. Zhu, "Low-Latency Proactive Continuous Vision," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*, 2020.

[273] MongoDB, "Explaining BSON with examples." [Online]. Available: https://www.mongodb.com/basics/bson

[274] A. Kwatra, "Intel Labs' Contributions to Latest Intel® Xeon® Scalable Processor," 2024. [Online]. Available: https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/Intel-Labs-Contributions-to-Latest-Intel-Xeon-Scalable-Processor/post/1441731

[275] R. Kuper, I. Jeong, Y. Yuan, R. Wang, N. Ranganathan, N. Rao, J. Hu, S. Kumar, P. Lantz, and N. S. Kim, "A Quantitative Analysis and Guidelines of Data Streaming Accelerator in Modern Intel Xeon Scalable Processors," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24), Volume 2*, 2024.

[276] ARM, "Learn the Architecture - SMMU Software Guide." [Online]. Available: https://developer.arm.com/documentation/109242/0100/Operation-of-an-SMMU/Address-Translation-Services

[277] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2024. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[278] S. Kim, J. Zhao, K. Asanovic, B. Nikolic, and Y. S. Shao, “AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’23)*, 2023.

[279] Intel, “Introduction to Memory Bandwidth Allocation.” [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html

[280] S.-T. Wang, H. Xu, A. Mamandipoor, R. Mahapatra, B. H. Ahn, S. Ghodrati, K. Kailas, M. Alian, and H. Esmaeilzadeh, “Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA’24)*, 2024.

[281] The Structural Simulation Toolkit, “ariel,” 2024. [Online]. Available: https://sst-simulator.org/sst-docs/docs/elements/ariel/intro

[282] T. Benz, M. Rogenmoser, P. Scheffler, S. Riedel, A. Ottaviano, A. Kurth, T. Hoefler, and L. Benini, “ A High-Performance, Energy-Efficient Modular DMA Engine Architecture ,” *IEEE Transactions on Computers*, 2024.

[283] A. Mirhosseini and T. Wenisch, “μSteal: A Theory-Backed Framework for Preemptive Work and Resource Stealing in Mixed-Criticality Microservices,” in *Proceedings of the ACM International Conference on Supercomputing (ICS ’21)*, 2021.

[284] C. Delimitrou and C. Kozyrakis, “Amdahl’s Law for Tail Latency,” *Commun. ACM*, vol. 61, no. 8, jul 2018.

[285] G. Jeong, B. Sharma, N. Terrell, A. Dhanotia, Z. Zhao, N. Agarwal, A. Kejariwal, and T. Krishna, “Characterization of Data Compression in Datacenters,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS ’23)*, 2023.

[286] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, “A cloud-scale acceleration architecture,” in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’16)*, 2016.

[287] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA ’14)*, 2014.

[288] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, “The mondrian data engine,” in *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA ’17)*, 2017.

[289] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: the architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, 2014.

[290] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '13)*, 2013.

[291] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing SoC accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, 2013.

[292] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015.

[293] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing Server Efficiency in the Face of Killer Microseconds," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*, 2019.

[294] A. Amarnath, S. Pal, H. T. Kassa, A. Vega, A. Buyuktosunoglu, H. Franke, J.-D. Wellman, R. Dreslinski, and P. Bose, "Heterogeneity-Aware Scheduling on SoCs for Autonomous Vehicles," *IEEE Computer Architecture Letters*, 2021.

[295] S. Baskaran, M. T. Kandemir, and J. Sampson, "An architecture interface and offload model for low-overhead, near-data, distributed accelerators," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*, 2022.

[296] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh, "Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*, 2018.

[297] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "CHARM: a composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '12)*, 2012.

[298] P. Lin and A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach," *CoRR*, vol. abs/1903.12221, 2019. [Online]. Available: http://arxiv.org/abs/1903.12221

[299] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[300] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.

[301] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.

[302] "Serverless Train Ticket." [Online]. Available: https://github.com/FudanSELab/serverless-trainticket

[303] AWS, "AWS Lambda Anti-Patterns: Synchronous waiting within a single Lambda function." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/synchronous-waiting.html

[304] AWS, "AWS Lambda Anti-Patterns: Lambda functions calling Lambda functions." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/functions-calling-functions.html

[305] M. Khairy, A. Alawneh, A. Barnes, and T. G. Rogers, "SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*, 2022.

[306] Linux Manual Page, "ld.so(8)." [Online]. Available: https://man7.org/linux/man-pages/man8/ld.so.8.html

[307] "OpenWhisk Invoker." [Online]. Available: https://github.com/apache/openwhisk/tree/master/core/invoker

[308] "OpenWhisk Load Balancer." [Online]. Available: https://github.com/apache/openwhisk/tree/master/core/controller/src/main/scala/org/apache/openwhisk/core/loadBalancer

[309] "KNative Serving Activator." [Online]. Available: https://github.com/knative/serving/tree/main/pkg/activator

[310] "KNative Serving Autoscaler." [Online]. Available: https://github.com/knative/serving/tree/main/pkg/autoscaler

[311] AWS, "Security Overview AWS Lambda." [Online]. Available: https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-overview-aws-lambda/security-overview-aws-lambda.pdf

[312] Z. Li, Q. Chen, S. Xue, T. Ma, Y. Yang, Z. Song, and M. Guo, "Amoeba: QoS-Awareness and Reduced Resource Usage of Microservices with Serverless Computing," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS '20)*, 2020.

[313] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, Predicting and Scheduling Serverless Workloads under Partial Interference," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC '21)*, 2021.

[314] L. Ao, G. Porter, and G. M. Voelker, "FaaSnap: FaaS Made Fast Using Snapshot-Based VMs," in *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*, 2022.

[315] "Redis." [Online]. Available: https://redis.io/

[316] Q. Liu and Z. Yu, "The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View from Alibaba Trace," in *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC '18)*, 2018.

[317] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC '12)*, 2012.

[318] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces," in *Proceedings of the International Symposium on Quality of Service (IWQoS '19)*, 2019.

[319] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC '21)*, 2021.

[320] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling Resource Underutilization in the Serverless Era," in *Proceedings of the 21st International Middleware Conference (Middleware '20)*, 2020.

[321] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling Quality-of-Service in Serverless Computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*, 2020.

[322] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A Scalable Low-Latency Serverless Platform," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[323] Linux Manual Page, "perf-stat(1)." [Online]. Available: https://man7.org/linux/man-pages/man1/perf-stat.1.html

[324] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments," in *Proceedings of the 21st International Middleware Conference (Middleware '20)*, 2020.

[325] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.

[326] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "Micro" Back in Microservice," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.

[327] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-Aware Thread Management," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, 2018.

[328] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *Proceedings of the VLDB Endowment*, 2020.

[329] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[330] H. Li, Y. Yuan, R. Du, K. Ma, L. Liu, and W. Hsu, "DADI: Block-Level Image Service for Agile and Elastic Application Deployment," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.

[331] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, 2016.

[332] Amazon AWS, "Implementing statelessness in functions." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/statelessness-functions.html

[333] Microsoft, "Serverless architecture considerations." [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/serverless/serverless-architecture-considerations

[334] Microsoft Azure, "Azure Blob Storage." [Online]. Available: https://azure.microsoft.com/en-us/products/storage/blobs

[335] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware Data Passing for Chained Serverless Applications," in *2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[336] M. Wawrzoniak, I. Müller, G. Alonso, and R. Bruno, "Boxer: Data Analytics on Network-enabled Serverless Platforms," in *Proceedings of the 11th Annual Conference on Innovative Data Systems Research (CIDR '21)*, 2021.

[337] D. Ustiugov, S. Jesalpura, M. B. Alper, M. Baczun, R. Feyzkhanov, E. Bugnion, B. Grot, and M. Kogias, "Expedited Data Transfers for Serverless Clouds," *CoRR*, vol. abs/2309.14821, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2309.14821

[338] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, "OFC: An Opportunistic Caching System for FaaS Platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*, 2021.

[339] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications," in *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[340] L. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE transactions on computers*, vol. 100, no. 12, pp. 1112–1118, 1978.

[341] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A primer on memory consistency and cache coherence. Second edition.* Springer Nature, 2020.

[342] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, 1990.

[343] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '91)*, 1991.

[344] Engineering at Meta, "Cache made consistent," 2024. [Online]. Available: https://engineering.fb.com/2022/06/08/core-infra/cache-made-consistent/

[345] Amazon AWS, "Saga pattern." [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html

[346] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[347] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker, "How Does It Function? Characterizing Long-Term Trends in Production Serverless Workloads," in *Proceedings of the 14th ACM Symposium on Cloud Computing (SoCC '23)*, 2023.

[348] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's Time to Revisit LRU vs. FIFO," in *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '20)*, 2020.

[349] Amazon AWS, "Developing for retries and failures." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/retries-failures.html

[350] Google Cloud, "Enable event-driven function retries." [Online]. Available: https://cloud.google.com/functions/docs/bestpractices/retries

[351] Microsoft Azure, "Designing Azure Functions for identical input." [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-idempotent

[352] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC '97)*, 1997.

[353] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, "On-demand Container Loading in AWS Lambda," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '23)*, 2023.

[354] T. Haynes and P. Noveck, "Network file system (NFS) version 4 protocol." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7530

[355] Microsoft Azure, "Create a function in Azure that's triggered by Blob storage," 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-create-storage-blob-triggered-function

[356] R. Fernández-Pascual, J. M. García, M. E. Acacio, and J. Duato, "Fault-Tolerant Cache Coherence Protocols for CMPs: Evaluation and Trade-Offs," in *Proceedings of the 15th International Conference on High Performance Computing (HiPC '08)*, 2008.

[357] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, 2010.

[358] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman, 2022.

[359] Amazon AWS, "Implement the serverless saga pattern by using AWS Step Functions." [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/implement-the-serverless-saga-pattern-by-using-aws-step-functions.html

[360] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed. Morgan and Claypool Publishers, 2010.

[361] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica, "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*, 2023.

[362] A. Patil, V. Nagarajan, N. Nikoleris, and N. Oswald, "Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS," in *Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '23)*, 2023.

[363] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching," in *17th USENIX Conference on File and Storage Technologies (FAST '19)*, 2019.

[364] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[365] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger, "The CacheLib Caching Engine: Design and Experiences at Scale," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[366] J. Yang, Y. Yue, and K. V. Rashmi, "A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter," *ACM Transactions on Storage*, vol. 17, no. 3, 2021.

[367] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports, "Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[368] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger, "Kangaroo: Caching Billions of Tiny Objects on Flash," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021.

[369] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, "RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, 2018.

[370] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, "Assise: Performance and availability via client-local NVM in a distributed file system," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[371] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[372] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, ser. SOSP '89, 1989.

[373] A. Fuerst and P. Sharma, "Locality-Aware Load-Balancing For Serverless Clusters," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22)*, 2022.

[374] M. Abdi, S. Ginzburg, X. C. Lin, J. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, 2023.

[375] "How long does AWS Lambda keep your idle functions around before a cold start?" [Online]. Available: https://www.pluralsight.com/resources/blog/cloud/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start

[376] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," in *Proceedings of 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2018.

[377] Amazon AWS, "AWS Step Functions." [Online]. Available: https://aws.amazon.com/step-functions/

[378] Microsoft Azure, "Durable orchestrations," April 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-orchestrations?tabs=csharp-inproc

[379] IBM, "IBM Cloud Composer." [Online]. Available: https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer

[380] Google Cloud, "Workflows." [Online]. Available: https://cloud.google.com/workflows

[381] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella, "Archipelago: A Scalable Low-Latency Serverless Platform," *CoRR*, vol. abs/1911.09849, 2019. [Online]. Available: http://arxiv.org/abs/1911.09849

[382] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Serverless Workflows with Durable Functions and Netherite," *CoRR*, vol. abs/2103.00033, 2021. [Online]. Available: https://arxiv.org/abs/2103.00033

[383] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the Gap Between Serverless and Its State with Storage Functions," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*, 2019.

[384] Alibaba, "Function Compute." [Online]. Available: https://www.alibabacloud.com/product/function-compute

[385] F. Lardinois, "Platform9's Fission Workflows makes it easier to write complex serverless applications," 2017. [Online]. Available: https://techcrunch.com/2017/10/03/platform9s-fission-workflows-makes-it-easier-to-write-complex-serverless-applications/?guccounter=1

[386] O. Tardieu, “Serverless Composition of Serverless Functions.” [Online]. Available: https://s3.us.cloud-object-storage.appdomain.cloud/res-files/2516-debs19.pdf

[387] Apache Software Foundation, “Open Whisk Composer.” [Online]. Available: https://github.com/apache/openwhisk-composer-python

[388] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, 2014.

[389] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, “My VM is Lighter (and Safer) than Your Container,” in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP ’17)*, 2017.

[390] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, “Characterizing Serverless Platforms with ServerlessBench,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC ’20)*, 2020.

[391] C. Delimitrou and C. Kozyrakis, “Quasar: Resource-Efficient and QoS-Aware Cluster Management,” in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’14)*, 2014.

[392] Apache Software Foundation, “Annotations on OpenWhisk assets.” [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/annotations.md

[393] K.-T. A. Wang, R. Ho, and P. Wu, “Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment,” in *Proceedings of the Fourteenth EuroSys Conference (EuroSys ’19)*, 2019.

[394] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, “BabelFish: Fusing Address Translations for Containers,” in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA ’20)*, 2020.

[395] Y. Zhang, I. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, “Faster and Cheaper Serverless Computing on Harvested Resources,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP ’21)*, 2021.

[396] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, “Agile Cold Starts for Scalable Serverless,” in *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud ’19)*, 2019.

[397] Amazon AWS, “AWS Serverless Workshops.” [Online]. Available: https://github.com/aws-samples/aws-serverless-workshops

[398] Amazon AWS, “AWS Serverless Airline Booking.” [Online]. Available: https://github.com/aws-samples/aws-serverless-airline-booking

[399] Amazon AWS, “AWS Orchestration Documentation.” [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/orchestration.html

[400] National Centers for Environmental Information, "Weather Data." [Online]. Available: https://www.ncei.noaa.gov/pub/data/uscrn/products/subhourly01/2021/

[401] University of California Irvine - Machine Learning Repository, "Online Retail Data Set II." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Online+Retail+II

[402] N. Antonio, A. de Almeida, and L. Nunes, "Hotel Booking Demand Datasets," *Data in Brief*, 2019.

[403] Bureau of Transportation Statistics, "Origin and Destination Survey." [Online]. Available: https://www.transtats.bts.gov/Fields.asp?gnoyr_VQ=FHK

[404] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic Management of Data and Computation in Datacenters," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[405] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *Proceedings of the VLDB Endowment*, 2010.

[406] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-Based Approximation for Data Parallel Applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, 2014.

[407] H. Rito and J. Cachopo, "Memoization of Methods Using Software Transactional Memory to Track Internal State Dependencies," in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*, 2010.

[408] A. Fuchs and D. Wentzlaff, "Scaling Datacenter Accelerators with Compute-Reuse Architectures," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*, 2018.

[409] V. Vassiliadis, M. A. Johnston, and J. L. McDonagh, "Fast, Transparent, and High-Fidelity Memoization Cache-Keys for Computational Workflows," in *2022 IEEE International Conference on Services Computing (SCC '22)*, 2022.

[410] W. Lee, E. Slaughter, M. Bauer, S. Treichler, T. Warszawski, M. Garland, and A. Aiken, "Dynamic Tracing: Memoization of Task Graphs for Dynamic Task-Based Runtimes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, 2018.

[411] CB Insights, "Why Serverless Computing Is The Fastest-Growing Cloud Services Segment." [Online]. Available: https://www.cbinsights.com/research/serverless-cloud-computing/

[412] Preeti Wadhwani, "Serverless architecture market size to cross $90 Bn by 2032." [Online]. Available: https://www.gminsights.com/pressrelease/serverless-architecture-market

[413] E. R. Masanet, A. Shehabi, N. Lei, S. J. Smith, and J. G. Koomey, "Recalibrating global data center energy-use estimates," *Science*, 2020.

[414] A. Andrae and T. Edler, "On Global Electricity Usage of Communication Technology: Trends to 2030," *Challenges*, vol. 6, 2015.

[415] U. Gupta, M. Elgamal, G. Hills, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "ACT: designing sustainable computer systems with an architectural carbon modeling tool," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*, 2022.

[416] U. Gupta, Y. Kim, S. Lee, J. Tse, H. S. Lee, G. Wei, D. Brooks, and C. Wu, "Chasing Carbon: The Elusive Environmental Footprint of Computing," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*, 2021.

[417] S. Chen, A. Jin, C. Delimitrou, and J. F. Martínez, "ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*, 2022.

[418] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan, "Gemini: Learning to Manage CPU Power for Latency-Critical Search Engines," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.

[419] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '15)*, 2015.

[420] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting Heterogeneity for Tail Latency and Energy Efficiency," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*, 2017.

[421] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*, 2015.

[422] J. Pont-Tuset, F. Perazzi, S. Caelles, P. Arbelaez, A. Sorkine-Hornung, and L. V. Gool, "The 2017 DAVIS challenge on video object segmentation," *CoRR*, vol. abs/1704.00675, 2017. [Online]. Available: http://arxiv.org/abs/1704.00675

[423] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[424] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, “Learning word vectors for sentiment analysis,” in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies.* Portland, Oregon, USA: Association for Computational Linguistics, June 2011. [Online]. Available: http://www.aclweb.org/anthology/P11-1015 pp. 142–150.

[425] S. Lahiri, “Complexity of Word Collocation Networks: A Preliminary Structural Analysis,” in *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics.* Gothenburg, Sweden: Association for Computational Linguistics, April 2014. [Online]. Available: http://www.aclweb.org/anthology/E14-3011 pp. 96–105.

[426] Y.-G. Jiang, J. Liu, A. Roshan Zamir, G. Toderici, I. Laptev, M. Shah, and R. Sukthankar, “THUMOS challenge: Action recognition with a large number of classes,” 2014. [Online]. Available: http://crcv.ucf.edu/THUMOS14/

[427] Google Cloud, “Serverless Computing.” [Online]. Available: https://cloud.google.com/serverless/

[428] M. Alian, A. H. M. O. Abulila, L. Jindal, D. Kim, and N. S. Kim, “NCAP: Network-Driven, Packet Context-Aware Power Management for Client-Server Architecture,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA ’17)*, 2017.

[429] K.-D. Kang, G. Park, H. Kim, M. Alian, N. S. Kim, and D. Kim, “NMAP: Power Management Based on Network Packet Processing Mode Transition for Latency-Critical Workloads,” in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’21)*, 2021.

[430] C.-H. Chou, L. N. Bhuyan, and D. Wong, “μDPM: Dynamic Power Management for the Microsecond Era,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA ’19)*, 2019.

[431] Y. Liu, S. C. Draper, and N. S. Kim, “SleepScale: Runtime joint speed scaling and sleep states management for power efficient data centers,” in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA ’14)*, 2014.

[432] Amazon AWS, “Configuring Lambda function options.” [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

[433] Microsoft Azure, “Available Instance SKUs.” [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal#available-instance-skus

[434] Google Cloud, “Cloud Functions Pricing.” [Online]. Available: https://cloud.google.com/functions/pricing

[435] KNative, "Configure resource requests and limits." [Online]. Available: https://knative.dev/docs/serving/services/configure-requests-limits-services/

[436] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*, 2023.

[437] MathWorks, "Mixed-Integer Linear Programming (MILP) Algorithms," April 2024. [Online]. Available: https://www.mathworks.com/help/optim/ug/mixed-integer-linear-programming-algorithms.html

[438] J. M. Lucas and M. S. Saccucci, "Exponentially weighted moving average control schemes: Properties and enhancements," *Technometrics*, vol. 32, no. 1, pp. 1–12, 1990.

[439] C. Holt, "Forecasting seasonals and trends by exponentially weighted moving averages," *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, 2004.

[440] J. van Winkel, "The Production Environment at Google, from the Viewpoint of an SRE." [Online]. Available: https://sre.google/sre-book/production-environment/

[441] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, 2013. [Online]. Available: http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[442] Intel, "Intel® Xeon® Processor E5-2660 v3." [Online]. Available: https://intel.com/content/www/us/en/products/sku/81706/intel-xeon-processor-e52660-v3-25m-cache-2-60-ghz/specifications.html

[443] Intel, "Intel® Xeon® Gold 6142 Processor." [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/120487/intel-xeon-gold-6142-processor-22m-cache-2-60-ghz.html

[444] Intel, "Intel® Xeon® Processor E5-2640 v4," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/92984/intel-xeon-processor-e52640-v4-25m-cache-2-40-ghz/specifications.html

[445] D. Beyer and P. Wendler, "CPU Energy Meter." [Online]. Available: https://github.com/sosy-lab/cpu-energy-meter

[446] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars, "HaPPy: Hyperthread-aware Power Profiling Dynamically," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '14)*, 2014.

[447] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe, "Process-Level Power Estimation in VM-Based Systems," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*, 2015.

[448] G. Fieni, R. Rouvoy, and L. Seinturier, “SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers,” in *Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID ’20)*, 2020.

[449] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, “RAPL in Action: Experiences in Using RAPL for Power Measurements,” *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, 2018.

[450] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati, “Portable, scalable, per-core power estimation for intelligent resource management,” in *Proceedings of the International Conference on Green Computing*, 2010.

[451] Amazon AWS, “Tune a Machine Learning Model.” [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/sample-hyper-tuning.html

[452] Python PuLP, “Optimization with PuLP.” [Online]. Available: https://coin-or.github.io/pulp/

[453] OSDev, “Model Specific Registers.” [Online]. Available: https://wiki.osdev.org/Model_Specific_Registers

[454] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, “Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis,” in *Proceedings of the 3rd Symposium on Cloud Computing (SoCC ’12)*, 2012.

[455] Microsoft Azure, “Overview of autoscale in Azure,” April 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview

[456] Google Cloud, “Autoscaling groups of instances,” 2023. [Online]. Available: https://cloud.google.com/compute/docs/autoscaler/

[457] Amazon AWS, “AWS Auto Scaling,” April 2024. [Online]. Available: https://aws.amazon.com/autoscaling/

[458] S. I. Abrita, M. Sarker, F. Abrar, and M. A. Adnan, “Benchmarking VM Startup Time in the Cloud,” in *Benchmarking, Measuring, and Optimizing: First BenchCouncil International Symposium*, 2019.

[459] IBM Cloud, “”Scaling stateful and stateless services”,” April 2024. [Online]. Available: https://www.ibm.com/docs/en/cloud-app-management/2019.3.0?topic=sizing-scaling-stateless-stateful-services

[460] Microsoft Azure, “Introduction to Auto Scaling,” April 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-autoscaling

[461] M. Jalili, I. Manousakis, I. Goiri, P. A. Misra, A. Raniwala, H. Alissa, B. Ramakrishnan, P. Tuma, C. Belady, M. Fontoura, and R. Bianchini, "Cost-Efficient Overclocking in Immersion-Cooled Datacenters," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021.

[462] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin, "Computational Sprinting," in *Proceedings of the IEEE 18th International Symposium on High-Performance Comp Architecture (HPCA '12)*, 2012.

[463] S. Fan, S. M. Zahedi, and B. C. Lee, "The Computational Sprinting Game," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, 2016.

[464] H. Cai, X. Zhou, Q. Cao, H. Jiang, F. Sheng, X. Qi, J. Yao, C. Xie, L. Xiao, and L. Gu, "GreenSprint: Effective Computational Sprinting in Green Data Centers," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '18)*, 2018.

[465] W. Zheng, X. Wang, Y. Ma, C. Li, H. Lin, B. Yao, J. Zhang, and M. Guo, "SprintCon: Controllable and Efficient Computational Sprinting for Data Center Servers," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '19)*, 2019.

[466] S. Kondguli and M. Huang, "A Case for a More Effective, Power-Efficient Turbo Boosting," *ACM Transactions on Architecture and Code Optimization (TACO '18)*, vol. 15, no. 1, 2018.

[467] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the Intel® Core™ i7 Turbo Boost feature," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '09)*, 2009.

[468] N. Morris, C. Stewart, L. Chen, R. Birke, and J. Kelley, "Model-Driven Computational Sprinting," in *Proceedings of the 13th ACM European Conference on Computer Systems (EuroSys '18)*, 2018.

[469] P. Ranganathan, P. Leech, D. Irwin, and J. Chase, "Ensemble-level Power Management for Dense Blade Servers," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*, 2006.

[470] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini, "Statistical profiling-based techniques for effective power provisioning in data centers," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, 2009.

[471] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song, "Dynamo: Facebook's Data Center-Wide Power Management System," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA '16)*, 2016.

[472] V. Sakalkar, V. Kontorinis, D. Landhuis, S. Li, D. De Ronde, T. Blooming, A. Ramesh, J. Kennedy, C. Malone, J. Clidaras, and P. Ranganathan, "Data Center Power Oversubscription with a Medium Voltage Power Plane and Priority-Aware Capping," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[473] S. Li, X. Wang, X. Zhang, V. Kontorinis, S. Kodakara, D. Lo, and P. Ranganathan, "Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[474] A. G. Kumbhare, R. Azimi, I. Manousakis, A. Bonde, F. Frujeri, N. Mahalingam, P. A. Misra, S. A. Javadi, B. Schroeder, M. Fontoura, and R. Bianchini, "Prediction-Based Power Oversubscription in Cloud Platforms," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[475] C. Zhang, A. G. Kumbhare, I. Manousakis, D. Zhang, P. A. Misra, R. Assis, K. Woolcock, N. Mahalingam, B. Warrier, D. Gauthier, L. Kunnath, S. Solomon, O. Morales, M. Fontoura, and R. Bianchini, "Flex: High-Availability Datacenters with Zero Reserved Power," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021.

[476] H. Rui, "amd-pstate CPU Performance Scaling Driver," April 2024. [Online]. Available: https://docs.kernel.org/admin-guide/pm/amd-pstate.html

[477] AMD, "CPU warranty terms," April 2024. [Online]. Available: https://www.amd.com/system/files/documents/processor-warranty-update.pdf

[478] Intel, "CPU warranty terms," April 2024. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/processors/Limited_Warranty_8.5x11_for_Web_English.pdf

[479] L. Piga, I. Narayanan, A. Sundarrajan, M. Skach, Q. Deng, M. C. B. Maity, A. Huang, A. Dhanotia, and P. Malani, "Expanding Datacenter Capacity with DVFS Boosting: A Safe and Scalable Deployment Experience," in *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*, 2024.

[480] J. Lyu, M. You, C. Irvene, M. Jung, T. Narmore, J. Shapiro, L. Marshall, S. Samal, I. Manousakis, L. Hsu, P. Subbarayalu, A. Raniwala, B. Warrier, R. Bianchini, B. Schroeder, and D. S. Berger, "Hyrax: Fail-in-Place Server Operation in Cloud Platforms," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*, 2023.

[481] G. Wang, L. Zhang, and W. Xu, "What Can We Learn from Four Years of Data Center Hardware Failures?" in *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '17)*, 2017.

[482] Amazon Web Services, "Processor state control for your EC2 instance," April 2024. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html

[483] Microsoft Azure, "Virtual Machine series," April 2024. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series

[484] Google Compute Platform, "CPU platforms," April 2024. [Online]. Available: https://cloud.google.com/compute/docs/cpu-platforms

[485] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *Proceedings of the ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED '10)*, 2010.

[486] Intel, "What Is Intel® Turbo Boost Technology?" April 2024. [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html

[487] AMD, "Turbo Core Technology," April 2024. [Online]. Available: https://www.amd.com/en/technologies/turbo-core

[488] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next Generation," in *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys '20)*, 2020.

[489] P. Alcorn, "How to Overclock Your CPU: CPU Overclocking Impact on Lifespan and Reliability," May 2023. [Online]. Available: https://www.tomshardware.com/how-to/how-to-overclock-a-cpu#section-cpu-overclocking-impact-on-lifespan-and-reliability

[490] D. Marcon, T. Kauerauf, F. Medjdoub, J. Das, M. Van Hove, P. Srivastava, K. Cheng, M. Leys, R. Mertens, S. Decoutere, G. Meneghesso, E. Zanoni, and G. Borghs, "A comprehensive reliability investigation of the voltage-, temperature- and device geometry-dependence of the gate degradation on state-of-the-art GaN-on-Si HEMTs," in *Proceedings of the 2010 International Electron Devices Meeting*, 2010.

[491] E. Wu, J. Sune, W. Lai, E. Nowak, J. McKenna, A. Vayshenker, and D. Harmon, "Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate oxides," *Solid-State Electronics*, vol. 46, no. 11, 2002.

[492] A. Yassine, H. Nariman, M. McBride, M. Uzer, and K. Olasupo, "Time dependent breakdown of ultrathin gate oxide," *IEEE Transactions on Electron Devices*, vol. 47, no. 7, 2000.

[493] D. DiMaria and J. Stathis, "Non-arrhenius temperature dependence of reliability in ultrathin silicon dioxide films," *Applied Physics Letters*, 1999.

[494] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg, "Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems," in *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '20)*, 2020.

[495] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, “MICA: A Holistic Approach to Fast In-Memory Key-Value Storage,” in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’14)*, 2014.

[496] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda, “Protean: VM Allocation Service at Scale,” in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’20)*, 2020.

[497] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin, “Computational Sprinting on a Hardware/Software Testbed,” in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’13)*, 2013.

[498] Y.-H. Lee, N. R. Mielke, W. McMahon, Y.-L. R. Lu, and S. Pae, “Thin-gate-oxide breakdown and cpu failure-rate estimation,” *IEEE Transactions on Device and Materials Reliability*, vol. 7, no. 1, pp. 74–83, 2007.

[499] W. R. Davis, C. Shaw, and A. R. Hassan, “How to write a compact reliability model with the open model interface (omi),” in *2020 IEEE International Reliability Physics Symposium (IRPS)*, 2020, pp. 1–2.

[500] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: Flexible, Scalable Schedulers for Large Compute Clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys ’13)*, 2013.

[501] Xen Project, “XenStore,” April 2024. [Online]. Available: https://wiki.xenproject.org/wiki/XenStore

[502] Microsoft Azure, “Data Exchange: Using key-value pairs to share information between the host and guest on Hyper-V,” April 2024. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/dn798287(v=ws.11)

[503] R. Russell, “Virtio: Towards a de-facto standard for virtual i/o devices,” *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, p. 95–103, jul 2008.

[504] Google Cloud, “About VM metadata,” April 2024. [Online]. Available: https://cloud.google.com/compute/docs/metadata/overview

[505] Amazon AWS, “Instance metadata and user data,” April 2024. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html

[506] Microsoft Azure, “Azure Instance Metadata Service,” April 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/instance-metadata-service

[507] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, “Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices,” in *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’21)*, 2021.

[508] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, “The power of prediction: microservice auto scaling via workload learning,” in *Proceedings of the 13th Symposium on Cloud Computing (SoCC ’22)*, 2022.

[509] A. F. Baarzi and G. Kesidis, “SHOWAR: Right-Sizing And Efficient Scheduling of Microservices,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC ’21)*, 2021.

[510] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, “Sage: practical and scalable ML-driven performance debugging in microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’21)*, 2021.

[511] Intel, “Intel Platform Analysis Technology,” April 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/platform-analysis-technology/overview.html

[512] Intel, “Platform Monitoring Technology Telemetry (PMT),” April 2024. [Online]. Available: https://github.com/intel/Intel-PMT

[513] AMD, “Host System Management Port (HSMP),” April 2024. [Online]. Available: https://github.com/amd/amd_hsmp

[514] ACPI Specification Revision Committee, “Advanced configuration and power interface specification,” 2022. [Online]. Available: https://uefi.org/specifications

[515] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-Scale Cluster Management at Google with Borg,” in *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys ’15)*, 2015.

[516] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, “Understanding Silent Data Corruptions in a Large Production CPU Population,” in *SOSP*, 2023.

[517] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, “Silent Data Corruptions at Scale,” *CoRR*, vol. abs/2102.11245, 2021. [Online]. Available: https://arxiv.org/abs/2102.11245

[518] P. H. Hochschild, P. J. Turner, J. C. Mogul, R. K. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, “Cores that don’t count,” in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS ’21)*, 2021.

[519] W. Zheng and X. Wang, “Data Center Sprinting: Enabling Computational Sprinting at the Data Center Level,” in *Proceedings of the IEEE 35th International Conference on Distributed Computing Systems (ICDCS ’15)*, 2015.

[520] H. Cai, Q. Cao, F. Sheng, Y. Yang, C. Xie, and L. Xiao, "ESprint: QoS-Aware Management for Effective Computational Sprinting in Data Centers," in *Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '19)*, 2019.

[521] D. Lo and C. Kozyrakis, "Dynamic management of TurboMode in modern multi-core chips," in *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*, 2014.

[522] R. P. Pothukuchi, J. L. Greathouse, K. Rao, C. Erb, L. Piga, P. G. Voulgaris, and J. Torrellas, "Tangram: Integrated Control of Heterogeneous Computers," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*, 2019.

[523] B. Greskamp and J. Torrellas, "Paceline: Improving Single-Thread Performance in Nanoscale CMPs through Core Overclocking," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*, 2007.

[524] A. Bacha and R. Teodorescu, "Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, 2013.

[525] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing Voltage Margins for Energy Efficiency in Multicore CPUs," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*, 2017.

[526] R. Bertran, A. Buyuktosunoglu, P. Bose, T. J. Slegel, G. Salem, S. Carey, R. F. Rizzolo, and T. Strach, "Voltage Noise in Multi-Core Processors: Empirical Characterization and Optimization Opportunities," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '14)*, 2014.

[527] K. K. Chang, A. G. Yağlıkçı, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," *Proceedings of the ACM on Measurements and Analysis of Computer Systems*, 2017.

[528] M. Kaliorakis, A. Chatzidimitriou, G. Papadimitriou, and D. Gizopoulos, "Statistical Analysis of Multicore CPUs Operation in Scaled Voltage Conditions," *IEEE Computer Architecture Letters*, 2018.

[529] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '03)*, 2003.

[530] A. A. Bhattacharya, D. Culler, A. Kansal, S. Govindan, and S. Sankar, "The need for speed and stability in data center power capping," in *Proceedings of the International Green Computing Conference (IGCC '12)*, 2012.

[531] V. Kontorinis, L. E. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, D. M. Tullsen, and T. S. Rosing, "Managing Distributed Ups Energy for Effective Power Capping in Data Centers," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*, 2012.

[532] C.-H. Hsu, Q. Deng, J. Mars, and L. Tang, "SmoothOperator: Reducing Power Fragmentation and Improving Power Utilization in Large-Scale Datacenters," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*, 2018.

[533] Z. Chen, J. Hu, G. Min, A. Y. Zomaya, and T. El-Ghazawi, "Towards Accurate Prediction for High-Dimensional and Highly-Variable Cloud Workloads with Deep Learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, 2020.

[534] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, 2021.

[535] J. Stojkovic, E. Choukse, C. Zhang, I. Goiri, and J. Torrellas, "DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '25)*, 2025.

[536] J. Stojkovic, C. Zhang, I. Goiri, E. Choukse, H. Qiu, R. Fonseca, J. Torrellas, and R. Bianchini, "TAPAS: Thermal- and Power-Aware Scheduling for LLM Inference in Cloud Platforms," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*, 2025.

[537] J. Stojkovic, D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, March 2022.

[538] J. Stojkovic, N. Mantri, D. Skarlatos, T. Xu, and J. Torrellas, "Memory-Efficient Hashed Page Tables," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA '23)*, February 2023.