OPTIMIZING FOUNDATIONAL SYSTEM BUILDING BLOCKS OF DATACENTER APPLICATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Zhuangzhuang Zhou

August 2024

OPTIMIZING FOUNDATIONAL SYSTEM BUILDING BLOCKS OF DATACENTER APPLICATIONS

Zhuangzhuang Zhou, Ph.D.

Cornell University 2024

Cloud computing has become the prevailing computing infrastructure for the majority of the world's computation. Computing platforms for cloud computing and large internet services are hosted in datacenters, and optimizing the performance of datacenter applications can result in significant cost savings. Given the diversity of datacenter workloads, optimizing a single application may not yield substantial improvements in the total system efficiency, as costs are spread across numerous independent workloads. In contrast, optimizing the foundational system building blocks of datacenter applications, including high-level system infrastructures to underlying system software libraries, can significantly improve the productivity of the datacenter fleet, since entire classes of datacenter applications can benefit from such optimizations.

This dissertation proposes a series of optimizations in foundational system building blocks of datacenter applications. Applications running in datacenter are often built as collections of loosely coupled services that are deployed and executed through high-level system building blocks such as serverless workflow engines and microservice frameworks. First, we focus on optimizing such a system building block at the top of the computing stack, the serverless computing framework. Despite the benefits of ease of programming, fast elasticity, and fine-grained billing, serverless computing suffers from resource inefficiency. We designed Aquatope, a QoS-and-uncertainty-aware resource sched-

uler for end-to-end serverless workflows that takes into account the inherent uncertainty present in FaaS platforms, and improves performance predictability and resource efficiency. Aquatope uses a set of scalable and validated Bayesian models to create prewarmed containers ahead of function invocations, and to allocate appropriate resources at function granularity to meet a complex workflow's end-to-end QoS, while minimizing resource cost.

Aquatope demonstrates that a joint solution to cold start and resource management, taking into account uncertainty, can effectively improve the resource efficiency of serverless applications. However, serverless workflows still suffer from significant control plane and inter-function communication overheads, which make them unsuitable for latency-critical applications. We also designed Meteion, a fast and efficient serverless workflow engine for latency-critical interactive applications. Meteion decouples the control plane from the workflow execution, and leverages lightweight per-function engines to enable decentralized workflow orchestration and direct inter-function communication. Meteion's DAG scheduler utilizes the workflow's latency distribution and graph structure to provision containers promptly, ensuring that functions can execute seamlessly on worker servers without falling back to the control plane.

Second, we delve into a foundational system library, the memory allocator. Datacenter applications typically share the usage of certain low-level software libraries, and memory allocation constitutes a substantial component of datacenter computation. Optimizing the memory allocator can improve application performance, leading to significant cost savings. We present the first comprehensive characterization of TCMalloc at warehouse scale. Our characterization reveals a profound diversity in the memory allocation patterns, allocated object sizes and lifetimes, for large-scale datacenter workloads, as well as in their

performance on heterogeneous hardware platforms. Based on these insights, we optimize TCMalloc for warehouse-scale environments. Specifically, we propose optimizations for each level of its cache hierarchy that include usage-based dynamic sizing of allocator caches, leveraging hardware topology to mitigate inter-core communication overhead, and improving allocation packing algorithms based on statistical data. Evaluation results show that these optimizations significantly improve the productivity of the datacenter fleet.

BIOGRAPHICAL SKETCH

Zhuangzhuang Zhou received his Bachelor of Science in Engineering degree from Shanghai Jiao Tong University in 2019, majoring in Electrical and Computer Engineering. During his undergraduate studies, Zhuangzhuang worked on approximate computing and electronic design automation under the supervision of Professor Weikang Qian. He then joined the School of Electrical and Computer Engineering at Cornell University as a Ph.D. student, where he was supervised by Professor Christina Delimitrou at the Computer Systems Laboratory. During his doctorate studies, Zhuangzhuang focused on improving the performance and efficiency of cloud computing systems, including optimizing serverless computing systems, and memory allocator tax in datacenter. He worked as a research intern at Intel in 2021 and at Google in 2023. Zhuangzhuang received a Master of Science degree in Electrical and Computer Engineering from Cornell University in May 2023.

This document is dedicated to my parents.

ACKNOWLEDGEMENTS

As I get closer to the destination of my doctoral journey, I sincerely realize that pursuing a Ph.D. is much more than my efforts alone. First and foremost, I would like to thank my advisor, Prof. Christina Delimitrou for her support and advice throughout my Ph.D. studies. Christina gave me great freedom to choose my research direction and explore topics of interest. She guided me to think critically about my research project and, most importantly, inspired me to recognize the nature of good systems research. I am also grateful to the other two committee members, Prof. José Martínez and Prof. Hakim Weatherspoon, for their invaluable advice, insights, and feedback.

My internship at Google has inspired me to think about real-world problems in large-scale optimization of datacenter workloads. I would like to thank my collaborators at Google, including Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, and Parthasarathy Ranganathan, for their guidance, feedback, and support. Vaibhav and Nilay are brilliant internship hosts that are always helpful, and they have made this internship a valuable and rewarding experience in my doctoral studies.

I would like to thank members of the SAIL group, including Yanqi Zhang, Yu Gan, Shuang Chen, Neeraj Kulkarni, Mingyu Liang, Nikita Lazarev, Varun Gohil, for numerous inspiring research discussions and valuable suggestions. Specifically, I would like to express my gratitude to Yanqi Zhang for his expertise and guidance in co-authoring papers. It has been a great pleasure to collaborate with him. Furthermore, I would like to thank all my friends at CSL and Cornell, including Yanghui Ou, Peitian Pan, Zichao Yue, Jie Liu, Philip Bedoukian, Lei Li, Zhongyuan Zhao, Yichi Zhang, Mulong Luo, Chenhui Deng, Yaohui Cai, Yixiao Du, Hongzheng Chen for all the fun memories and wonderful time we

have shared together. My years at Cornell have been made even better by the support of my friends back in China and around the world. I have certainly missed listing some below. Thanks to Qun Song, Zhi Lin, Yichen Hu, Wenyao Zhu, Weihong Song, Peiyuan Qi, Guohao Li, Zitao Zhang, Xiteng Liu, Yue Yao, Liliang Ren, Jiayao Wu, Yixuan Chen, Yifan Zhao, Xiaohan Fu, Shengyi Qian for keeping in touch with me over the years despite the spatial and temporal sparsity. I hope we can meet often in the future.

Finally, I would like to express my deepest gratitude to my parents, Qiang Zhou and Qian Zhu, for their unwavering love and support throughout my life. Without their support, I could not have made it through all the ups and downs to finish my Ph.D. journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background

Cloud computing has become the prevailing computing infrastructure for the majority of the world's computation, with multiple benefits, such as scalability to large systems, fine-grained elasticity, fast deployment, and protection against data loss [39,40,46,88,142]. Computing platforms for cloud computing and large internet services are often hosted in large datacenters. Optimizing the performance of datacenter applications can result in significant savings in resources, maintenance, and server costs. However, datacenter workloads are extremely diverse, and optimizing a single application may not yield substantial improvements in the total system efficiency, as costs are spread across numerous independent workloads.

In contrast, optimizing the foundational system building blocks of datacenter applications, ranging from high-level system infrastructures to underlying software libraries, can significantly improve the productivity of the datacenter fleet. Datacenter applications are often built as collections of loosely coupled services encapsulated in containers or virtual machines (VMs). These applications are deployed and executed through high-level system infrastructures such as serverless workflow engines and microservice frameworks, which also provide opportunities to optimize performance bottlenecks for distributed datacenter applications at scale. Moreover, applications running in the datacenter typically share the usage of certain low-level software libraries, and improving these libraries can yield substantial performance gain, since entire classes of

datacenter applications can benefit from such optimization.

Serverless computing [88] is one of such foundational building blocks at the top of the computing stack with great optimization opportunities. Serverless platforms provide a programming paradigm known as Function as a Service (FaaS) that simplifies cloud programming. Users can build their applications as functions, and upload them to the cloud provider's serverless platform, where functions are executed in response to external events. This event-driven interface makes FaaS a suitable candidate for applications with high data-level parallelism and/or intermittent activity. Moreover, FaaS frees users from the burden of managing virtual machines (VMs) or containers to run functions, which makes serverless computing increasingly popular. In major public clouds [5,8,14], serverless adoption continues to rise, with over 49 - 70% organizations adopting serverless solutions [38]. Large internet service providers have also built their serverless platforms in their hyperscale private clouds [130].

As serverless applications become more representative in datacenters, improving the performance of serverless systems has become a pressing need. The ease of use of FaaS for users comes at the expense of extra hardware costs for cloud providers. To accommodate the function initialization overhead and ensure application performance, the serverless provider must pre-provision a large amount of VMs or containers to provide fast elasticity and the illusion of unlimited resources [130, 133, 155]. While determining the optimal number of containers and their sizes for a serverless application is challenging, optimizing the resource management of the FaaS platform can significantly improve resource utilization in datacenter without sacrificing application performance. Furthermore, serverless applications are increasingly built as workflows con-

sisting of multiple loosely coupled functions that coordinate with each other to execute application logic. [6, 7, 15]. For serverless workflows, the resource management challenges are amplified due to cascading cold starts and varied resource needs across functions [158]. Serverless workflows also introduce new performance challenges such as increased orchestration, control plane, and inter-function communication overheads [52], which can be further optimized to meet the performance goals of latency-critical applications.

In addition to high-level system infrastructures, low-level software libraries also serve as foundational building blocks for datacenter applications. These software components, including serialization, remote procedure calls (RPCs), compression, hashing, compression, data movement, and memory allocation, are known as components of the *datacenter tax* [89, 141]. These libraries comprise a significant portion of computation in the datacenter fleet, and offer optimization opportunities to achieve substantial performance and efficiency gains. Memory allocator is one of the most important software libraries used by datacenter applications, since it not only impacts the time spent in memory allocation, but also directly affects the data locality of allocated objects. Optimizing a memory allocator can reduce cache and data Translation Lookaside Buffer (dTLB) misses, reduce back-end stalls, and ultimately improve fleet productivity, i.e., fulfilling more requests with the same or fewer hardware resources.

1.2 Contributions

This dissertation focuses on improving the performance of cloud computing systems by optimizing the foundational building blocks of datacenter applica-

tions, from the high-level system infrastructure (i.e., serverless computing) to the underlying system software library (i.e., memory allocator).

First, we focus on optimizing a high-level system building block, the serverless computing framework. Specifically, we aim to allocate the optimal amount of resources to serverless applications under quality of service (QoS) constraints. We present **Aquatope, a QoS-and-uncertainty-aware scheduler for multi-stage serverless workflows**. Aquatope jointly tackles the two main challenges contributing to degraded performance and resource inefficiency in FaaS: cold starts and function-level resource allocation. Aquatope consists of two major components, *a dynamic pre-warmed container pool* and *a container resource manager*. The dynamic pre-warmed container pool uses a hybrid Bayesian neural network to adjust the number of pre-warmed containers. The container resource manager leverages Bayesian Optimization to search for a near-optimal resource configuration for each execution stage in a workflow. Aquatope uses a Bayesian approach to account for the noise and uncertainty that are prevalent in FaaS platforms due to stochasticity inherent to function execution, load fluctuation, and interference from colocated applications. Aquatope is a centralized controller, operates online, transparently to the user, and introduces marginal overheads.

Aquatope demonstrates that a joint solution to cold start and resource management, taking into account uncertainty, can effectively improve the resource efficiency of serverless applications. However, serverless workflows still suffer from significant control plane and inter- function communication overheads, which make them unsuitable for latency-critical applications. We present **Meteion, a fast and efficient serverless workflow framework for latency-critical interactive applications**. Meteion decouples the control plane from

the workflow execution, and leverages lightweight per-function engines to enable decentralized workflow orchestration and direct inter-function communication. Meteion's DAG scheduler utilizes the workflow's latency distribution and graph structure to provision containers in a timely manner, ensuring that functions can execute seamlessly on the worker servers without falling back to the control plane.

Second, we delve into a foundational system library, the memory allocator. Memory allocation constitutes a substantial component of datacenter computation. Optimizing the memory allocator can improve application performance, leading to significant cost savings. We present the first comprehensive **characterization of TCMalloc at warehouse scale**. Our characterization reveals a profound diversity in the memory allocation patterns, allocated object sizes and lifetimes, for large-scale datacenter workloads, as well as in their performance on heterogeneous hardware platforms. Based on these insights, we optimize TCMalloc for warehouse-scale environments. Specifically, we propose optimizations for each level of its cache hierarchy that include usage-based dynamic sizing of allocator caches, leveraging hardware topology to mitigate inter-core communication overhead, and improving allocation packing algorithms based on statistical data. Evaluation results show that these optimizations significantly improve the productivity of the datacenter fleet.

1.3 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes the related work. Chapter 3 introduces Aquatope, a QoS and uncertainty-

aware resource management framework for multi-stage serverless workflows. Chapter 4 presents Meteion, a fast and efficient serverless workflow engine for latency-critical interactive applications. Chapter 5 presents the characterization and optimization of TCMalloc, a low-level system memory allocator used in the production datacenter fleet. Finally, Chapter 6 concludes the dissertation.

CHAPTER 2

RELATED WORK

We now review related work on foundational building blocks of datacenter applications, including both serverless frameworks and memory allocators.

2.1 Serverless

Mitigating cold starts: Cold starts can cause severe performance degradation in serverless execution. There are several strategies to mitigate cold start overheads, including pre-crafting virtual network interfaces [113], restoring a function from a well-formed checkpoint image to skip initialization [62], and prefetching a function's working set of memory pages [145]. Most FaaS providers keep container instances loaded in memory for a fixed amount of time after a function terminates [146]. AWS Lambda offers a *provisioned concurrency* [23] configuration to pre-load a fixed number of containers to accelerate function startup. FaaSCache [68] uses a caching-inspired container eviction policy to terminate containers when the server is saturated, but does not pre-warm containers. Shahrad et al. [133] proposed a histogram-based policy to adjust a container's keep-alive time. Similarly, IceBreaker [128] uses a Fourier-transformation-based model to predict future invocation patterns, and pre-warms function containers accordingly. Orion [111] samples workflow execution time under different resource configurations and pre-warms function containers in a workflow based on sampled execution time.

Resource scheduling for FaaS: Many FaaS resource schedulers focus on the storage side of serverless. Pocket [92] uses user-provided workload hints to

rightsize storage resources. Pu et al. [122] build application-specific performance models to select the storage configuration that achieves the desired cost-performance trade-off. There are a few systems that address the compute side of serverless management. Saha et al. [129] and Suresh et al. [143] use autoscaling to adjust a container's memory to satisfy a function's latency requirements. These systems are again designed for single-stage applications, and do not handle the diverse needs of different execution stages.

QoS-aware cloud management: There has been extensive work on resource managers that meet QoS for latency-critical cloud applications [59,61,71,73,149, 156]. PARTIES [54] showed that resources of interactive services are fungible, which simplifies resource partitioning when colocating multiple latency-critical jobs. CLITE [120], RAMBO [103], and SATORI [127] showed that Bayesian Optimization (BO) can identify resource configurations that meet QoS for latency-critical jobs, maximize throughput for batch workloads, and preserve fairness among colocated jobs. While these systems improve performance and resource efficiency, they are designed for long-running applications, and cannot be directly applied to multi-stage serverless applications built with transient function containers. Moreover, these approaches do not consider the noise and uncertainty present in FaaS infrastructures, which can greatly hinder traditional BO techniques.

Inter-function data transfer. Many prior proposals focus on optimizing data transfer overheads between functions in serverless workflows. SAND [43] leverages data locality, and transfers data between colocated functions through a local message bus. Pocket [92] uses user-provided workload hints to rightsize storage resources for exchanging intermediate data between execution stages

in a workflow. Locus [122] builds a performance model to select the optimal storage for analytics workflows to achieve the desired cost-performance trade-off. Pocket [92] and Locus [122] use user-provided workload hints and performance models to rightsize storage resources for exchanging intermediate data between execution stages in a workflow. SONIC [110] optimizes the data-passing method (e.g., direct passing, remote storage) for each stage of a serverless workflow, and performs communication-aware function placement. A recent characterization study of Azure Duration Functions shows that 50% of serverless workflows have less than 8KB of intermediate data transfers [112]. For many latency-critical workflows, data transfer overhead is not the major source of performance degradation, and we need to improve their performance by enabling in-situ workflow execution.

Serverless workflow engines: There has been extensive work on optimizing the performance of serverless workflow engines. Nightcore [87] and Faastlane [93] use local memory to accelerate inter-function communication. Atoll [138] partitions the workflow DAG into subgraphs and assigns them to semi-global schedulers and worker pools to reduce the scheduling overheads. Similarly, FaaSFlow [105] uses per-worker workflow engine to reduce workflow scheduling and control plane networking overheads. While all the aforementioned systems can improve the performance of serverless workflow, they still suffer from interleaved execution between the control plane and function containers.

2.2 Memory allocator

Profiling datacenter workload. Prior studies on profiling production workloads deployed in the datacenter demonstrate that low-level software building blocks are ideal optimization candidates to achieve performance gains at scale [61, 76, 89, 94, 141]. Kanev et al. [89] profiled thousands of Google services in the datacenter fleet, identified common building blocks in datacenter computation, and proposed architectural optimizations accordingly. Sriraman et al. [141] characterized diverse microservices in Meta to show system-level and architectural acceleration opportunities. One of the most important building blocks is the memory allocator, which not only affects the time spent in the allocation itself, but also directly affects data locality of allocated objects. While these studies investigate the behavior of memory allocators in datacenter workloads, they only address CPU cycles consumed by the allocator.

Modern memory allocator. To improve memory allocation performance, custom modern memory allocators [36, 42, 50, 65, 78, 95, 99, 104, 106, 121, 150] are used to replace the default `malloc` implementation (e.g., glibc [13]). Large-scale services use these allocators at scale. `jemalloc` [65] (used by Meta [66]) emphasizes fragmentation avoidance and scalable concurrency support. `mimalloc` [99] (used by Microsoft [24]) improves locality by providing users with objects from the same page. Hoard [50] focuses on reducing thread contention and false sharing. Mesh [121] uses remapping of virtual pages and randomized allocation to enable memory compaction. `snmalloc` [106] uses batched message-passing instead of per-thread caching to send batches of deallocations to the originating thread.

Hugepage support. There has been extensive work on optimizing memory management in the kernel to improve physical memory contiguity and provide better hugepage support [96, 118, 119, 123, 148, 157]. Ingens [96] manages contiguity as a first-class resource and tracks utilization and access frequency of memory pages. Contiguitas [157] proposes to separate movable allocations from unmovable ones by placing them into different memory regions. User-space memory allocators can also benefit from kernel optimizations because they rely on the system to provide contiguous memory.

CHAPTER 3

AQUATOPE: QOS-AND-UNCERTAINTY-AWARE RESOURCE MANAGEMENT FOR MULTI-STAGE SERVERLESS WORKFLOWS

3.1 Introduction

Serverless computing is gaining popularity as a high-level system infrastructure for deploying datacenter workloads, due to its ease of programming and maintenance, fast elasticity, and fine-grained billing. Serverless simplifies management for users, since its interface removes the need for users to explicitly configure virtual machines (VMs) or containers, and users only pay for the resources they use during execution. For applications with high data-level parallelism and intermittent activity, serverless can achieve much higher performance for the same or lower cost.

Despite these benefits, serverless introduces several challenges, especially when a service has to meet quality of service (QoS) requirements in terms of execution time or tail latency. A lot of prior work has focused on reducing the cold start overheads in serverless, i.e., overheads associated with instantiating new containers or VMs, and installing necessary dependencies [62, 68, 133, 145, 146]. While impactful, cold starts are not the sole reason behind degraded performance in serverless. Another crucial issue the system has to tackle is appropriate function-level resource management. Without a proper resource configuration, the function can suffer from performance degradation and increased execution cost [152]. More importantly, these two problems are closely correlated with each other, as cold and warm starts lead to different function performance, and require significantly different resources. For the system to minimize cost,

while satisfying QoS, we need to tackle both challenges jointly.

Furthermore, serverless providers are increasingly providing workflow programming model interfaces, where each serverless application consists of multiple loosely-coupled functions in pursuit of fine-grained scalability and modular development and deployment [6, 7, 152]. The challenges above are amplified for multi-stage serverless workflows, where cascading cold starts across dependent stages [57] and varied resource needs for each stage [152] make cold start elimination and resource management even more challenging. Finally, serverless is prone to high system-level noise due to the interference from colocated workloads in FaaS deployments, which further hinder performance predictability [133].

To improve the resource efficiency of serverless applications, we present Aquatope, a QoS-and-uncertainty-aware scheduler for multi-stage serverless workloads that jointly tackles the two main challenges contributing to degraded performance and inefficiency in Function-as-a-Service (FaaS): cold starts and function-level resource allocation. Aquatope consists of two major components, *a dynamic pre-warmed container pool* and *a container resource manager*. The dynamic pre-warmed container pool uses a hybrid Bayesian neural network to adjust the number of pre-warmed containers. The container resource manager leverages Bayesian Optimization to search for a near-optimal resource configuration for each execution stage in a workflow. Aquatope uses a Bayesian approach to account for the noise and uncertainty that are prevalent in FaaS platforms due to stochasticity inherent to function execution, load fluctuation, and interference from colocated applications. Aquatope is a centralized controller, operates online, transparently to the user, and introduces marginal overheads.

We implement Aquatope on OpenWhisk [3] and evaluate it across a wide set of analytics and interactive multi-stage serverless applications, including ML pipelines, video processing frameworks, and social networks. In all cases, Aquatope outperforms prior empirical and ML-driven approaches in performance and efficiency, reducing QoS violations by 5× compared to prior work [21, 120, 128], and execution cost by 34% on average and up to 52% compared to other QoS-meeting methods.

3.2 Background and Motivation

3.2.1 Problem Statement

Many real-world serverless applications are implemented as *multi-stage serverless workflow* in which incoming user requests invoke sets of serverless functions that coordinate with each other to execute a serverless workflow. Existing platforms provide various composition mechanisms to control a workflow and transfer intermediate state across functions [4, 6, 7]. By splitting a complex application into dependent but loosely-coupled functions, the application benefits from fine-grained scalability, parallel execution, and modular development [152]. At the same time, decoupling a serverless application into multiple stages also introduces challenges in resource management, including additional function instantiation overheads, data transfer overheads, and varied resource needs across execution stages. Without an appropriate framework in place, multi-stage serverless workflows can experience QoS violations and resource inefficiency.

Aquatope specifically targets such serverless workflows that must meet predefined QoS constraints. Aquatope tackles two correlated aspects of serverless resource management: ensuring that function instantiation overheads are minimal so that tasks do not suffer from cold starts, and optimizing the resource configuration of each stage to minimize cost while satisfying QoS. While Aquatope is geared towards multi-stage serverless workloads, it can also be applied to simpler applications with a single stage.

3.2.2 Challenges

Resource management for multi-stage serverless workflows faces the following challenges.

Cold starts: Cold starts are one of the most studied overheads associated with serverless [68,128,132,146,152]. A *cold start* invocation occurs when a serverless application is triggered, but its function instances are not yet loaded in memory. For the FaaS platform, a cold start involves launching a new container (and/or a new VM), setting up its runtime environment, and fetching and loading necessary libraries and dependencies. This process can take a long time relative to the short-lived function execution [68,146].

Diverse resource requirements: Serverless functions vary in functionality and are implemented with different libraries and runtimes. Their resource requirements also vary a lot [132, 152], and without proper resource management, both performance and cost can suffer. Existing FaaS platforms, including AWS Lambda [5], Google Cloud Functions [14], and IBM Cloud Functions [17] re-

quire users to specify a memory limit for serverless functions, and allocate CPU resources proportional to the amount of provisioned memory, which can lead to CPU or memory overprovisioning.

Correlation of cold start and resource allocation: Cold starts not only affect the function startup latency but also exacerbate runtime performance degradation, as they can prevent a function invocation from reusing its execution context [9], which caches global variables (e.g., SDK clients, database connections, ML models, etc.). In this case, the function is forced to execute the user-provided initialization code to download data dependencies and initialize runtime packages, etc. [68]. This leads to different runtime performance and resource requirements for warm and cold starts, with cold start function invocations requiring more resources to meet the same performance target than warm start invocations. Without eliminating cold starts, the resource manager is forced to strike a balance between the performance behaviors of cold and warm starts, leading to degraded performance and excess resources.

Multi-stage serverless workflow overheads: Serverless application developers tend to decouple complex applications (e.g., ML inference, interactive web service) into workflows of loosely-coupled functions. Despite the advantages of fine-grained scalability and modular development, the performance of such applications can suffer for multiple reasons. First, the startup overhead is amplified by cascading cold starts across dependent functions [57, 152]. Second, resource requirements can vary a lot across the execution stages of the same workflow. Without proper resource management for each execution stage, the application would either fail to satisfy its QoS and/or suffer from increased cost. Additionally, different function composition methods (e.g., asynchronous

invocation, function callback, function chaining, fan-in/fan-out, etc.) introduce more performance unpredictability, which makes finding a near-optimal resource configuration for the whole application more challenging.

Uncertainty in FaaS: Noise and uncertainty are inherent to FaaS platforms. Serverless is well-suited for applications with fluctuating workloads due to their fine-grained scalability and pay-as-you-go pricing model [5]. A large fraction of serverless applications have significant variability in invocation patterns, making it difficult to provision appropriate resources for them in advance [68,133]. In addition, due to their short execution time and fine granularity, cloud providers tend to colocate serverless functions to higher degrees than traditional cloud services. As a result, functions can suffer from interference from colocated workloads and lead to unpredictable performance [146, 152], which causes biased observations and impairs the performance of sampling-based resource management approaches [103,120,127].

3.3 Aquatope Design Overview

Aquatope is a QoS-and-uncertainty-aware resource scheduler for end-to-end, multi-stage serverless workflows. The design objective of Aquatope is to meet the user-defined QoS of a multi-stage serverless application, while using the minimum amount of resources. To this end, Aquatope jointly tackles two correlated challenges in serverless: (1) it maintains a *dynamic pre-warmed container pool* to minimize cold starts and ensure most of the function invocations are handled by warm containers, and (2) it employs a *container resource manager* that allocates appropriate resources to each function, based on its warm-start

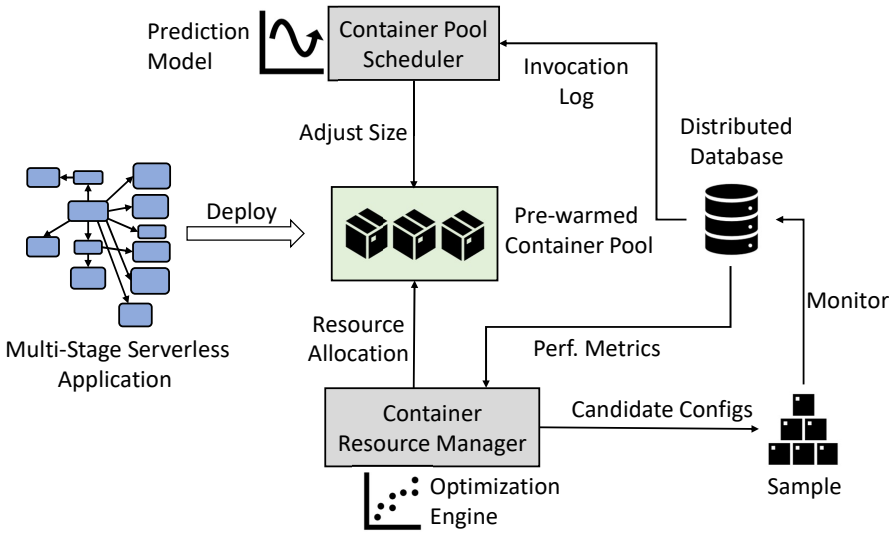


Figure 3.1: System overview of Aquatope.

performance behavior.

Both components are implemented with Bayesian approaches to overcome the uncertainties present in FaaS platforms, including workload fluctuations, performance unpredictability, and interference from colocated applications. The dynamic pre-warmed container pool uses a hybrid Bayesian neural network [153], which provides accurate and high-confidence predictions of function invocation rates, allowing Aquatope to adjust the number of pre-warmed containers ahead of function invocations. The container resource manager then builds surrogate models to approximate the relationship between resource configurations and end-to-end performance and cost. The engine uses customized Bayesian optimization to efficiently explore the resource allocation space to find a configuration that meets the end-to-end QoS with minimal overprovisioning. It arrives at a suitable configuration by balancing exploration and exploitation, while adapting to noise in the cloud.

By integrating these two components, Aquatope jointly tackles both chal-

lenges, and allows multi-stage serverless applications to operate in a performant and efficient manner. The following two sections describe each of Aquatope's components in detail.

3.4 Eliminating Cold Starts

Aquatope maintains a pool of pre-warmed containers to handle incoming function invocations. Aquatope sizes the pre-warmed container pool at runtime such that there are just enough warm containers to handle incoming function invocations. Aquatope also determines when to terminate a function's container to reclaim unused resources.

Since multi-stage serverless applications are built with diverse runtimes and topologies, the optimal number of pre-warmed containers is application-specific. Aquatope uses a set of machine learning (ML) models to infer the total number of required containers for each active serverless application over the next time interval, and adjusts the number of different types of containers accordingly. While several models can be applied towards this purpose, Aquatope uses a hybrid Bayesian neural network to infer future invocation rates, which achieves high accuracy, fast inference, and agility to load fluctuations.

3.4.1 Time Series Prediction

The problem of predicting function invocation patterns can be formalized as follows: given a number of different types of active function containers for a serverless workflow in the past t time windows $\{x_1, x_2, ..., x_t\}$, we need to predict

the invocation pattern for the next time window $\{x_{t+1}\}$. The time window size is configurable, and is set to 1 minute by default, which is the typical timescale for container keep-alive times in FaaS platforms [3, 133]. External features, which are the time of day, time of week, and function trigger types (HTTP, object storage, event hub, etc.), also need to be integrated into the prediction model to improve accuracy. Aquatope also accounts for the dependencies between functions in a multi-stage workflow, by predicting the invocation pattern of downstream containers in x_{t+1}, when it sees their upstream containers invoked in $\{x_1, x_2, ..., x_t\}$. This captures both probabilistic and deterministic dependencies between execution stages, by predicting the expected and exact number of containers respectively. Since load fluctuates and invocation patterns may change, it is also important to incorporate uncertainty estimation to improve the robustness of the model, and to ensure that the scheduler makes reliable decisions and can recover from anomalies.

3.4.2 Hybrid Bayesian Neural Network Model

Classic timeseries prediction models (e.g., exponential smoothing, ARIMA models, Theta method) usually require manual tuning to configure the model and uncertainty parameters [85]. Moreover, it is difficult to incorporate external features into these models, which can be impactful to accuracy. Long Short Term Memory (LSTM) models [82] have also gained popularity in timeseries prediction. LSTM can capture long-term sequential dependencies in the data and outperform traditional methods [97]. However, conventional LSTM models cannot easily embed non-temporal external features or incorporate noise and uncertainty into their predictions, which is important for handling fluctuating

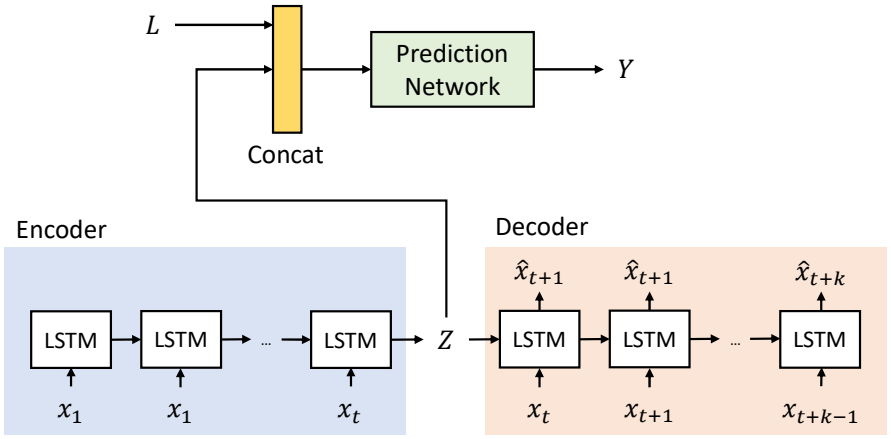


Figure 3.2: Aquatope's hybrid Bayesian model for the dynamic pre-warmed function container pool, consisting of a LSTM encoder-decoder and a prediction network.

workloads.

To overcome these problems and achieve generalizable and scalable prediction, we build a hybrid Bayesian neural network model. The novelty of our Bayesian model is twofold. First, it can utilize external features, such as time of day, to forecast function invocations. Second, it takes system noise into account when making predictions, allowing it to provide reliable uncertainty estimation, which is critical for fluctuating workloads. As shown in Fig. 3.2, the model consists of two parts: (i) the Long Short-Term Memory (LSTM) encoder-decoder, which serves as a feature-detection blackbox that extracts a latent variable from the input timeseries; and (ii) the prediction network, which infers the invocation pattern in the next time window using the latent variable and external features. We use Monte Carlo (MC) dropout [69] to approximate Bayesian inference and quantify the prediction uncertainty.

LSTM encoder-decoder: Before training the prediction model, we first construct and train the LSTM encoder-decoder to extract latent features from a serverless trace, which contains information of the historical invocation pat-

terns. The LSTM encoder-decoder consists of two LSTMs modules. The encoder processes the input workload sequence ($X = \{x_1, x_2, ..., x_t\}$), and generates the latent variable (Z), which summarizes its information. The decoder uses the extracted latent variable to produce the output workload sequence for the upcoming k windows $\{x_{t+1}, x_{t+2}, ..., x_{t+k}\}$.

The LSTM encoder-decoder is constructed using stacked LSTM cells with two layers. The encoder and decoder have 64 and 16 features in the hidden states respectively; the network's configuration is discussed below.

Prediction network: After training the LSTM encoder-decoder, we use the LSTM encoder as an automatic feature-extraction blackbox. The last hidden state of the encoder is the latent variable Z. Then, we train a prediction network to forecast the number of active containers (Y) in the next time window, using Z as features. To further increase the prediction accuracy, we concatenate the external feature vector (L) with Z, then feed it into the prediction network. We build the prediction network using a multi-layer perceptron, which consists of tanh activation functions and three fully connected layers. The model parameters of the LSTM and prediction network are selected based on the validation accuracy.

Bayesian inference: Incorporating noise and uncertainty into the model is essential for accurate timeseries forecasting under fluctuating load. To enable this, we leverage approximate Bayesian inference. Due to its simplicity, generality and scalability, we use MC dropout [69] to approximate Bayesian neural networks and achieve epistemic uncertainty estimates, rather than training a deterministic model. We apply variational dropout to the encoder [70], and regular dropout to the prediction network. By applying stochastic dropouts to each hid-

den layer of the encoder and prediction network, we can obtain the predictive mean and variance through T forward passes using different samples of model weights ($\{W_t\}_{t=1}^{T}$).

3.4.3 Prediction-Based Container Pool Manager

Aquatope adjusts the number of pre-warmed containers for the next time window based on model predictions, by creating warm containers in advance to accommodate incoming invocations, and shutting down idle containers in time to save resources. The adjustment interval of the container pool is 1 minute, which is long enough to hide the container instantiation overhead, and is the typical time-scale for container keep-alive times in production FaaS platforms [133]. The latency of the prediction model is below 10ms, which is negligible compared to the adjustment interval of the container pool.

3.5 Optimizing Per-Function Resources

Cold starts are not the sole reason for performance degradation in FaaS platforms. It is also critical to ensure that the resources allocated to each function are appropriate. The pre-warmed container pool manager ensures that the majority of function invocations are handled by warm containers, which simplifies function-level resource allocation, narrowing it down to only considering the warm-start performance behavior of serverless workflows.

Aquatope needs to consider the diverse resource requirements of each function across execution stages. Manually deriving an analytical performance

model for a variety of applications is difficult. On the other hand, exhaustively searching the entire configuration space is time consuming and expensive, since the total number of available configurations grows exponentially with the number of stages in a workflow. Moreover, each configuration needs to be profiled multiple times to get around the noise in FaaS platforms.

Rather than relying on manually-derived analytical models or exhaustive profiling, Aquatope uses Bayesian Optimization (BO), a data-driven approach, to learn the mapping from resource configurations to performance and cost. BO has been effective in black-box resource optimization for long-running cloud workloads [44,120,127], where application behaviors are not known to the cloud provider in advance. However, previous BO-based resource managers did not take noise and uncertainty into account, leading to increased search time and cost, and degraded performance. Aquatope's container resource manager leverages an improved Bayesian Optimization (BO) approach that considers noise and uncertainty and is robust to biased observations and data outliers, resulting in fast convergence and lower search overheads. Aquatope also exploits the scalability of serverless workloads to accelerate exploration by enabling batch sampling, rather than using individual samples as in previous work.

We first describe the BO algorithm workflow, and then discuss the challenges that prevent conventional BO from being robust to noise in FaaS platforms. Finally, we discuss Aquatope's customized BO that overcomes these challenges.

3.5.1 Bayesian Optimization Workflow

Problem formulation: Formally, for a multi-stage serverless application, we want to find the resources (c) that minimize execution cost (f), while satisfying the end-to-end QoS (λ); the formula is shown in Eq. 3.1. The resource configuration includes the CPU, memory, and concurrency settings for all functions in the application, consistent with the interface of major FaaS providers [5, 8, 14]. The execution cost is linear to the CPU and memory time, consistent with cost models in production serverless platforms [5, 8, 14]. The optimization objective is:

$$\min_{c} f(c) \text{ subjects to } \ell(c) \leq \lambda \tag{3.1}$$

$f(c)$ and $\ell(c)$ are black-box functions, whose values (cost and execution time respectively) can be observed by sampling resource configuration c. Collecting more samples increases the probability of finding a good configuration, at the cost of increased exploration overheads. However, the search process is under both time and budget constraints, as shown in Eq. 3.2, in which T_{budget} denotes the budget towards sampling resource configurations $\{c_1, c_2, ..., c_k\}$, and T_{time} indicates the time constraint for the exploration process.

$$\sum_{k=1}^{K} f(c_k) \leq T_{budget} \quad \text{and} \quad \sum_{k=1}^{K} \ell(c_k) \leq T_{time} \tag{3.2}$$

Bayesian optimization: BO relies on two key components. First, BO relies on a model that captures the relationship between input and objective function to drive the optimization process, a model commonly referred to as surrogate

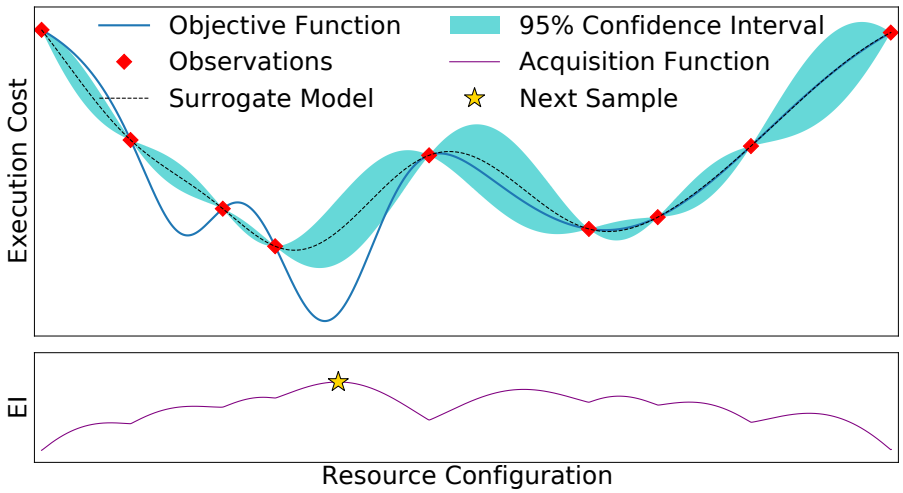


Figure 3.3: Iterative process in Bayesian Optimization (BO).

model in BO literature [134]. Second, BO leverages acquisition functions that determine the next data point to be sampled based on the predictions of the surrogate model. As shown in Fig. 3.3, the algorithm proceeds iteratively and in each epoch, the surrogate model is updated with the data (resource configuration and corresponding performance metrics) sampled in the previous epoch, and the acquisition function leverages the updated surrogate model to determine the next data point (candidate resource configuration) to be sampled in the current epoch.

3.5.2 Challenges for Conventional BO

Conventional BO-based resource managers can suffer from increased search time and cost, and degraded performance due to the following challenges:

- **Cloud noise:** Previous BO-based resource managers assume a noiseless setting [120] [127] [134]. However, the cloud is a noisy environment. For example, resource interference and workload fluctuation, can exacerbate

performance unpredictability, and result in biased observations of workload performance. Serverless applications can also suffer from interference, leading to misleading observations (outliers) in BO's sampling process. In this case, the naive BO workflow would suffer from model misspecifications caused by outliers, and the GP models would fail to characterize the performance of the workflow.

- **QoS constraint:** Adding black-box inequality constraints like QoS constraints to BO is challenging [134]. Prior BO-based resource managers [120, 127] rely on manually crafted objective functions with a penalty term that is triggered upon QoS violation, to guide the sampling process. However, manually crafted objective functions lack the flexibility to capture the behavior of complex serverless workflows and can lead to slow convergence and performance degradation.
- **Batch sampling:** Conventional BO samples and evaluates one configuration at a time [134], limiting the speed of convergence. For serverless applications, if we take advantage of the scalability of serverless by sampling multiple configurations at a time, the exploration can be greatly accelerated, improving the resource savings.

3.5.3 Customized Bayesian Optimization

We propose a customized BO which addresses the challenges above. The algorithmic novelty of this customized BO is threefold

- First, different from previous approaches that ignore or underestimate cloud noises, Aquatope takes noise and uncertainty into account by de-

sign, when searching for a near-optimal resource configuration. To capture various forms of cloud noise, Aquatope divides noises into two categories: one is inherent noise, which can be approximated well by a normal distribution; the other is irregular noise which does not follow a normal distribution. The latter includes noise caused by resource contention or networking instability. We refer to the first type of noise as *Gaussian noise* and to the second as *non-Gaussian noise*. Aquatope uses noise-aware surrogate models and acquisition functions to account for Gaussian noise, and builds diagnostic models to prune the non-Gaussian data outliers.

- Second, Aquatope effectively incorporates end-to-end QoS constraints into BO. Unlike conventional BO that relies on a manually crafted objective function with a reactive penalty term that is triggered when a QoS violation occurs, Aquatope takes a proactive approach by building a surrogate model that predicts end-to-end performance, and uses the predictions of the model to filter candidate configurations that may violate QoS.
- Finally, instead of sampling one configuration at a time, Aquatope employs batch sampling with customized acquisition functions, substantially reducing the exploration time, without sacrificing the quality of the selected resource allocation configuration.

Customized surrogate models: Aquatope uses Gaussian process (GP) [124] as the surrogate model. GP is a suitable surrogate model for resource exploration for several reasons. GP is non-parametric and does not make any assumptions over the target black-box function and is thus flexible enough to capture the relationship between resources and performance. GP is also computationally tractable and can be evaluated and updated cheaply and often [51]. Finally, GP can provide a measure of uncertainty for the predictions of unsampled data

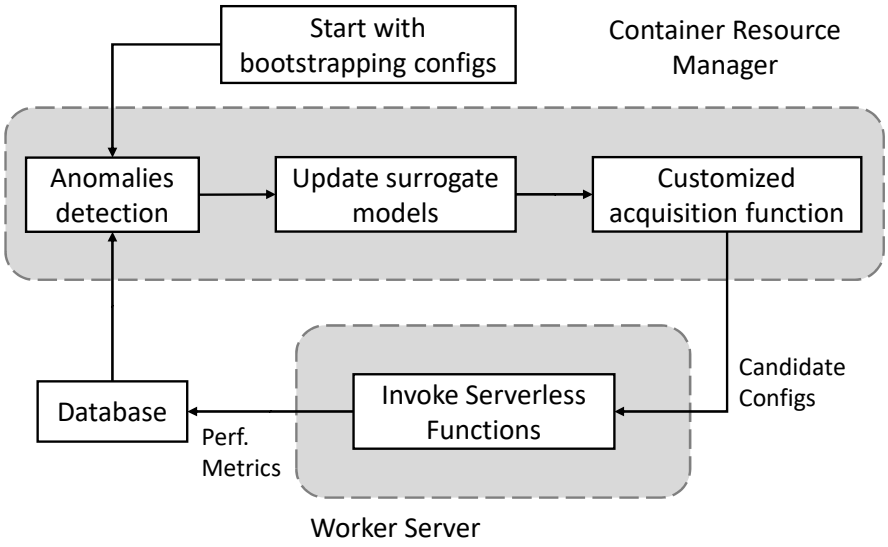


Figure 3.4: Workflow of Aquatope's container resource manager.

points and naturally captures Gaussian noise. Specifically, Aquatope uses fixed-noise GP models with Matérn(5/2) as the covariance kernel [124] to model the Gaussian noise.

More importantly, instead of combining the cost and performance targets with a manually crafted objective function [120, 127] and building a single GP model for it, Aquatope builds independent GP models for the cost target f and the QoS constraint ℓ. The intuition for separating the two is to allow the GP models to converge faster and more accurately. The cost GP model captures the cost reduction for an unsampled resource configuration, and the performance GP model narrows down the search space, by discerning the regions more likely to be feasible (i.e., satisfy QoS).

Customized acquisition function: Aquatope uses customized acquisition functions to select the next batch of candidate configurations, maximizing expectation of improvement (cost reduction) over the current best observation. The classic expected improvement (EI) acquisition function [139] provides a reason-

able balance between exploration and exploitation at a low computation cost. However, EI selects one candidate in each iteration and assumes noiseless observations. Instead, we leverage recent advances in BO to use constrained noisy expected improvement (NEI) with quasi-Monte Carlo integration (QMC) [102]. NEI takes Gaussian observation noise into consideration and does not assume the best observation is known, which would require noiseless observations.

We use the method in [74] to multiply NEI of reducing cost with the probability of satisfying QoS, which is derived from the performance GP model, to obtain the constrained NEI. The constrained NEI helps Aquatope to focus on the feasible configuration space, where QoS can be met. QMC provides an approximation of constrained NEI and its gradient, which do not have analytic expressions, and enables batch optimization by iteratively maximizing NEI integrated over pending unobserved samples. We use a batch size of 3, which speeds up the search without sacrificing quality.

Anomaly detection: We refer to data outliers from non-Gaussian noise as anomalies. Aquatope builds diagnostic models to prune anomalies in the sampling process. For each sampled configuration, we create a diagnostic GP model using data points other than the one under evaluation. The diagnostic GP model computes the predictive mean and confidence interval to identify a possible anomaly. If the observed value of that configuration falls outside the 95% predictive confidence interval, it is labeled as an anomaly. We evaluate all observed configurations and add potential anomalies to the list.

Batch evaluation: After obtaining a batch of candidate configurations, Aquatope sends requests to the pre-warmed container pool to launch the serverless workflow, to ensure warm starts. Then it profiles all candidate configura-

tions in parallel and evaluates their performance. We use both QoS-preserving and QoS-violating sample observations to update the surrogate models, because QoS-violating configurations help the GP models to identify which regions are more likely to meet QoS without actually sampling them.

Putting it all together: The complete workflow of the customized BO engine is shown in Fig. 3.4. The BO engine starts with a few randomly sampled configurations to warm up the surrogate models. Then the BO engine proceeds iteratively. In each iteration, the BO engine uses the customized acquisition functions to select a batch of candidate configurations to sample that are likely to preserve QoS. When the sampling finishes and performance metrics are retrieved, the observed performance metrics are first sent to the anomalies detection engine to filter misleading observations, which are then used to update both the performance and cost surrogate models.

Incremental retraining: The anomaly detection mechanism also allows Aquatope to detect changes in the performance behavior of serverless workflows, when the observed performance metrics deviate from the model predictions. These deviations can be caused by changes in the input workload, function updates, etc. In this scenario, Aquatope performs incremental retraining, and updates the model by collecting new samples using a sliding window, and gradually adapts to changes in the application behavior.

3.6 System Implementation

Aquatope is built over Apache OpenWhisk [3]; a widely-used open-source FaaS platform that powers IBM's Cloud Functions [17]. Fig. 3.5 shows Aquatope's

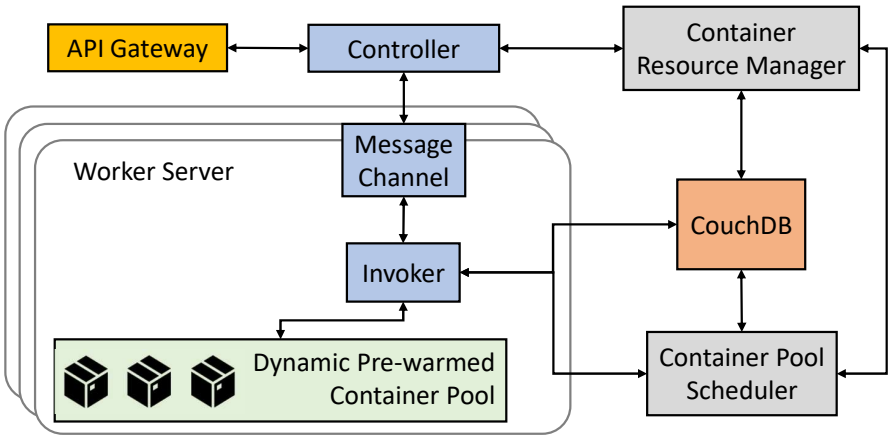


Figure 3.5: Architecture of Aquatope's implementation.

implementation.

OpenWhisk architecture: The API gateway of OpenWhisk is implemented with NGINX [27]. The backend of OpenWhisk consists of controllers and invokers that scale horizontally, with one invoker deployed per worker server. Function invocations are forwarded to a controller, which chooses an invoker to execute the invocation by considering invoker capacity and execution history. The invocation is sent to the invoker through a message channel implemented with Kafka [2]. Function implementations, invocation histories, execution results, and statistics are stored in CouchDB [1].

Resource scheduling: By default, OpenWhisk allocates a relative share of CPU proportional to the amount of memory provisioned for each function container. To implement Aquatope, we modified the resource scheduling mechanism of OpenWhisk to decouple CPU and memory resource allocations, and support CPU-limit-based resource scheduling.

Dynamic pre-warmed container pool: Similar to AWS Lambda's provisioned concurrency [23], OpenWhisk's invoker maintains a pool of pre-warmed con-

tainers (*stem cell*) for heavily-used functions. By default, the configuration of the pre-warmed container pool is static and pre-defined, and all worker servers share the same configuration. We modify the controller and invoker to support dynamic adjustment of the pre-warmed container pool, making it worker-server specific, and configured via the controller for all managed invokers (or via the invoker directly). The load balancer in the controller is aware of the pre-warmed containers and routes function invocation requests to the supporting invokers accordingly.

Container pool scheduler: Aquatope runs an independent service to control the pre-warmed containers. It fetches metadata for the serverless applications requiring pre-warmed resources, and their invocation histories from CouchDB. For each application, the scheduler trains the prediction model, and uses it to adjust the dynamic pre-warmed container pool. The hybrid Bayesian NN is implemented with PyTorch [29]. The scheduler makes decisions in each time interval and sends the updated container pool configurations to the invokers.

Container resource manager: Aquatope aims to find a near-optimal configuration for a serverless application. When a new application is registered, Aquatope obtains its metadata and QoS from CouchDB, and starts the optimization process. The GP models are implemented using GPyTorch [75] and the optimization workflow is implemented in BoTorch [49]. The engine samples the candidate resource configurations on the worker servers. The execution results and performance metrics are fetched from CouchDB. After selecting a near-optimal configuration, the engine sends messages to the controller to update the configuration of the application.

3.7 Methodology

3.7.1 Applications

Generic function workflows: We first implement several generic function workflows using the Apache OpenWhisk Composer [4], to combine multiple synthetic serverless functions into multi-stage workflows. We create a function generator to synthesize configurable resource-intensive functions that emulate varying CPU and memory workloads. We generate two workflows which are often present in multi-stage workflows: Chain and Fan-out/Fan-in. In Chain, a sequence of functions executes in a specific order. The output of one function is fed to its downstream function. In Fan-in/Fan-out, the workflow executes multiple functions in parallel, and only returns when the last child completes after some aggregation.

ML pipeline: We implement an ML pipeline that serves as the backend of a parking lot security system, trying to recognize humans and vehicles. Fig. 3.6 shows the architecture of the ML pipeline. The image the parking lot security camera records is uploaded to the object store and triggers the ML pipeline. The pipeline performs object detection [77] and the labeled images are uploaded to the object store, with vehicle and human recognition being invoked in parallel.

Video processing framework: We implement a serverless video processing framework similar to Sprocket [45]. The input video URLs are fetched and decoded into fixed-length frames. Then different function pipelines are invoked and the video chunks are processed in parallel. Fig. 3.7 shows the framework's architecture. We use MinIO [25] as the ephemeral storage for the video frames

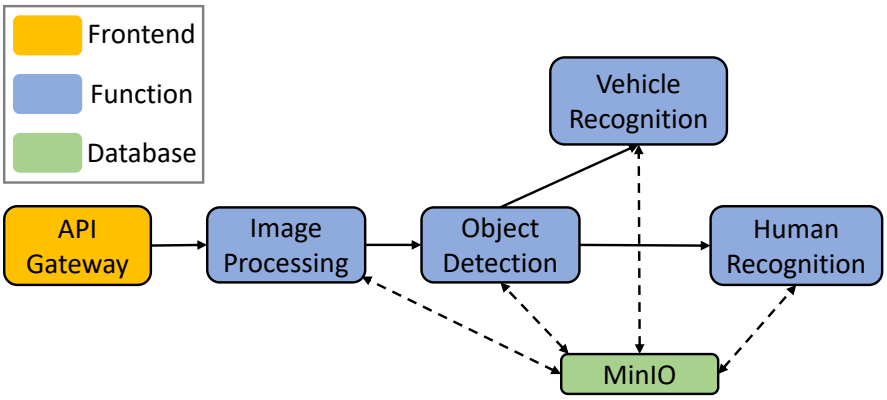


Figure 3.6: ML pipeline architecture.

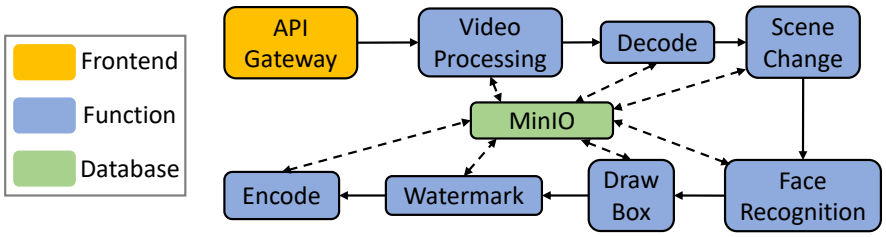


Figure 3.7: Video processing framework architecture.

for each stage.

Social network: We use a serverless implementation of the broadcast-style Social Network in DeathStarBench [72]. Fig. 3.8 shows the architecture of the serverless implementation of the service. Users can create posts embedded with text, media, links, and tags, which are then broadcast to all their followers. The texts and images uploaded by users go through the *text-filter* and *image-filter* functions. Contents violating the service's ethical guidelines are rejected. Users can also read posts on their timelines. The backend uses Memcached and Redis for caching, and MongoDB for persistent storage. We use the socfb-Reed98 Facebook network dataset [126] as the social graph, with 962 users and 18.8K follow relationships.

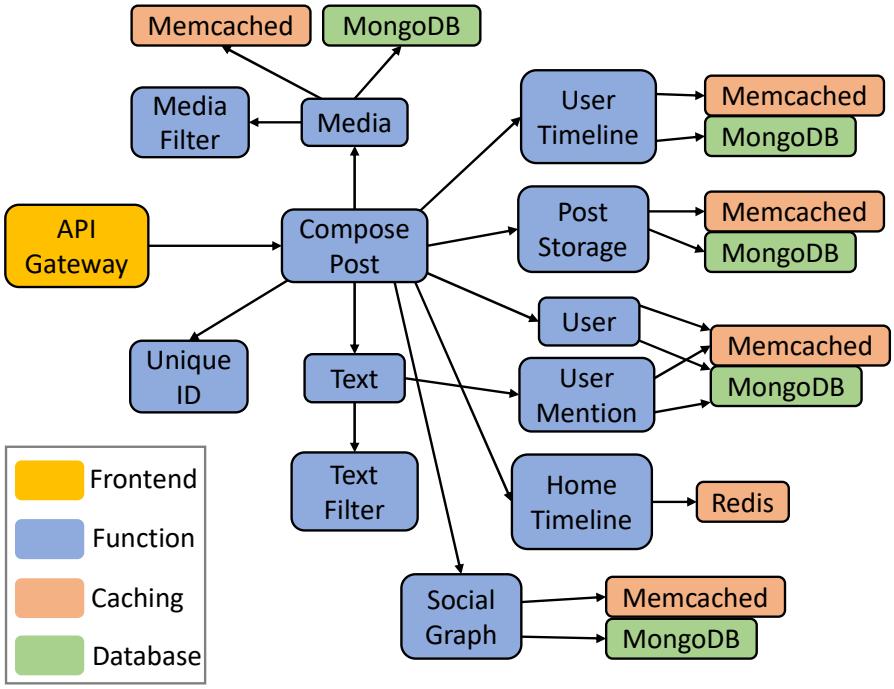


Figure 3.8: Serverless social network architecture [72].

3.7.2 Workload Generation

We use the Locust [22] load generator to emulate real user traffic. We generate custom-shaped loads based on scaled-down invocation pattern traces from the Azure Function Dataset [133]. Since the Azure dataset does not contain traces of Azure Durable Functions (the workflow engine for composing function logic) [7], we use the function invocation traces to emulate workflow invocation patterns. Within each one-minute interval provided in the trace, we use a Poisson process to generate workflow invocation traffic with an exponential distribution of inter-arrival times. We scale the invocation rate proportionally so that the maximum CPU utilization in the cluster does not exceed 70%, which is in accordance with the CPU utilization in the Google and Alibaba production clusters [135], and the Azure Function cluster [155]. This workload generation method is consistent with the methodology in [133]. The load generators and

functions are never physically co-located on a server.

3.7.3 Server Cluster

We deploy Aquatope to a dedicated local cluster with five, 2-socket, 40-core servers using Intel x86 Xeon E5s with 128GB RAM each, and two 2-socket, 88-core servers using Intel Gold 6152 processors with 188GB RAM each. Each server is connected to a 40Gbps ToR switch over 10Gbe NICs. All machines run Ubuntu 18.04.3 LTS. We use one of the 40-core servers to host the controller, API gateway, CouchDB and other system components, including Aquatope. Each of the remaining servers hosts an invoker and maintains a dynamic pre-warmed container pool to run the functions using Docker.

3.7.4 Comparison Baselines

We compare Aquatope with multiple strategies that mitigate cold starts and optimize resource allocations. In terms of reducing cold starts, we compare against (1) the fixed keep-alive policy used by most FaaS providers [5, 8]; (2) Apache OpenWhisk's reactive-stem-cell policy [16], which enables autoscaling for pre-warmed containers; (3) FaaSCache's container eviction and dynamic auto-scaling policy [68]; (4) histogram-based container keep-alive policy in [133], which uses historical function inter-arrival time to dynamically adjust the keep-alive time; (5) Icebreaker [128], which uses Fourier Transformation to predict and pre-warm function containers based on historical invocation patterns.

For resource management, we compare Aquatope with (a) autoscaling techniques [129, 143], that dynamically adjust a container's CPU and memory to match the function's latency requirements; (b) a random-search-based tuning system [81]; and (c) CLITE [120], which uses a BO-driven approach to search for a near-optimal resource configuration that minimizes the cost while satisfying QoS. We modify CLITE's score function to make it applicable to multi-stage serverless applications. Details are provided in Section 3.8.2.

3.8 Evaluation

We first evaluate Aquatope's two key components (dynamic pre-warmed container pool and container resource manager) separately, and then perform an end-to-end evaluation that includes both components.

3.8.1 Dynamic Pre-warmed Container Pool

Prediction model accuracy: We first evaluate the accuracy of the hybrid Bayesian NN used to predict the number of pre-warmed containers in Aquatope, by measuring its average accuracy across different serverless workflows and invocation patterns. We also compare Aquatope's model with three alternatives: (1) fixed Keep-Alive: A naïve model that uses the number of invoked containers in the last time window as the prediction for the next. (2) ARIMA [85]: Auto-Regressive Integrated Moving Average, a classic time-series prediction model used in Microsoft Azure's "Serverless in the Wild" system [133], and (3) LSTM [82]: a vanilla LSTM model with similar configuration

Table 3.1: Prediction accuracy measured in SMAPE.

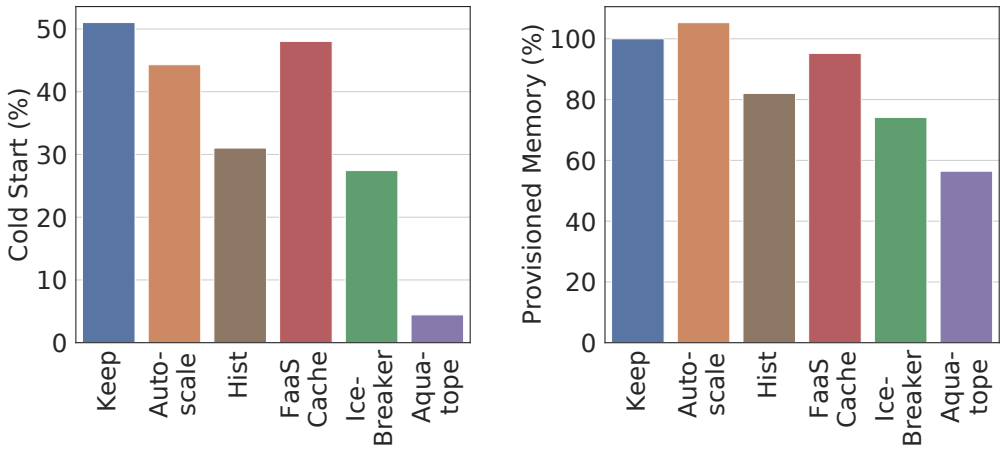
Prediction Error	**Prediction Models**			
	Fixed Keep-Alive	ARIMA	LSTM	**Aquatope**
SMAPE	24.5%	18.6%	9.5%	5.7%

as our hybrid model, but without considering external features, such as time of day/week and function types, or taking uncertainty into account. We use the same training dataset for all systems and evaluate performance on a separate test dataset.

Table 3.1 shows the Symmetric Mean Absolute Percentage Error (SMAPE) of the four models across all workflows in terms of pre-warmed vs. required containers, a widely used metric in time series prediction [159]. Aquatope's hybrid model significantly outperforms all other alternatives, with a 40% reduction in prediction error compared to the second best model, the vanilla LSTM. Our proposed Bayesian NN outperforms fixed Keep-Alive and ARIMA because these simple analytical models do not fully capture the dynamic invocation pattern, and it outperforms the vanilla LSTM because it uses information-rich external features as input, and also takes into account cloud noise and uncertainty when making predictions.

Eliminating cold starts: We now evaluate the cold start elimination approach in Aquatope compared to previous work. The results are shown in Fig. 3.9a.

The fixed *Keep-Alive* policy keeps containers alive for another 10 minutes after executing the last invocation, and the resulting the cold start rate is 51%. The autoscaling policy [16, 21] adjusts the number of pre-warmed containers based on utilization, and achieves cold start rate of 44%. However, autoscaling relies on reactive feedback control, and cannot adjust the containers fast enough,



(a) Function cold starts.

(b) Provisioned memory time.

Figure 3.9: Aquatope's dynamic pre-warmed container pool outperforms other empirical and data-driven approaches.

when load fluctuates rapidly. FaaSCache [68] performs similarly to autoscaling. This is expected since FaaSCache's container eviction policy is only triggered when server resources are exhausted, and is not designed for typical cloud deployments, where resources are plentiful, as is the case in the Azure function traces we use [133]. Therefore, FaaSCache falls back to a conservative dynamic auto-scaling policy. The histogram-based method in [133] and IceBreaker [128] use the function invocation inter-arrival time distribution to predict future invocations, and dynamically pre-warm and keep-alive containers. They outperform autoscaling and further eliminate 13%–17% of cold starts. However, neither the histogram model nor IceBreaker's Fourier-transformation-based model can capture complex timeseries patterns nor do they exploit external features, including time of day/week, to improve accuracy.

Aquatope uses the hybrid Bayesian model to account for both timeseries information and external features, and eliminates 24% more cold starts than IceBreaker, resulting in a cold start rate of less than 4%.

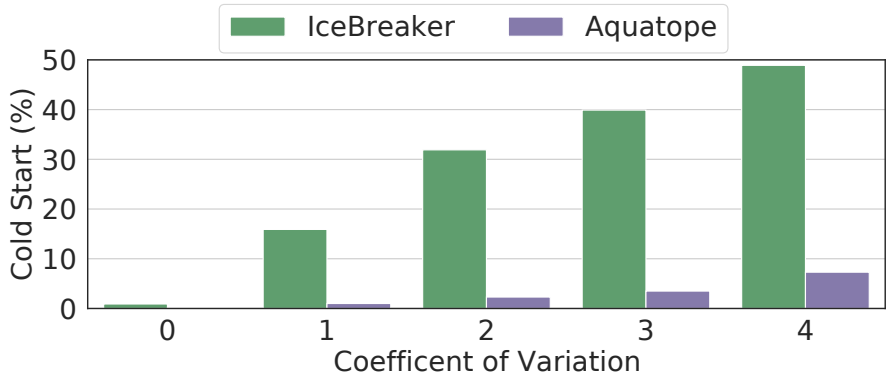


Figure 3.10: Aquatope outperforms IceBreaker, the best-performing previous work for cold start elimination, for input workloads with different coefficients of variation (CV). A CV greater than 1 suggests a large variation in the inter-arrival time of workflow invocations. Aquatope achieves higher benefits for highly fluctuating loads because it accounts for noise and uncertainty.

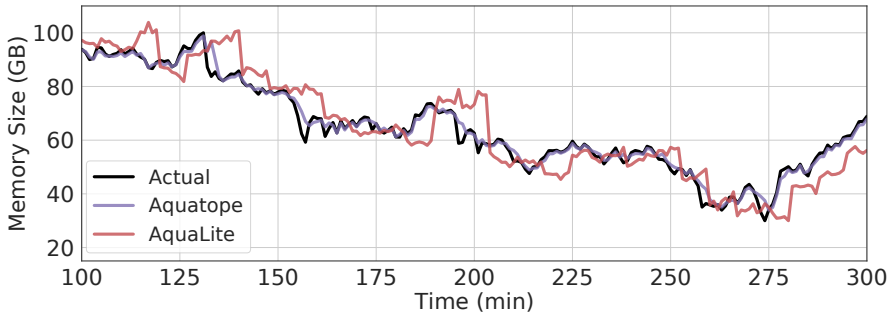


Figure 3.11: Aquatope's dynamic pre-warmed container pool adapts to fluctuating workload better than the AquaLite that does not account for uncertainty.

Reducing over-provisioned memory: Although pre-warming containers reduces cold starts, holding containers in memory for too long wastes resources. Fig. 3.9b shows the relative aggregate provisioned memory time for each approach. We use the same resource configuration for serverless containers across all approaches for a fair comparison.

Autoscaling increases the pre-warmed containers in large steps to satisfy performance, but reduces them in much smaller steps when container utilization is low. However, the temporal bursts common in serverless invocations

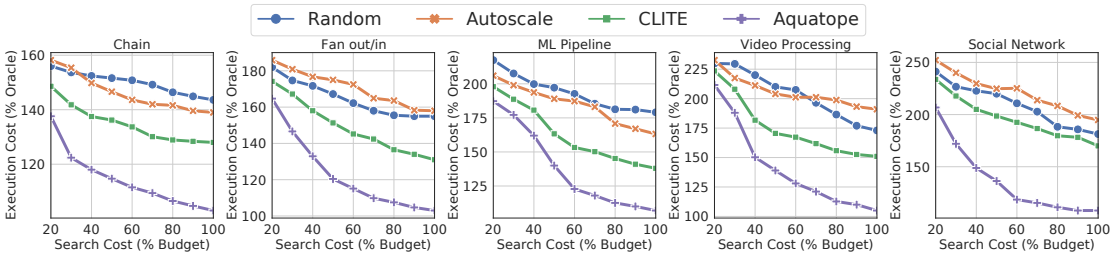


Figure 3.12: Iteratively searching for a near-optimal configuration across two synthetic and the three end-to-end workflows.

can lead to over-provisioning of pre-warmed containers, which can take a long time to reclaim resources. As a result, the provisioned memory time of autoscaling is 5% higher than for *Keep-Alive*. IceBreaker reduces memory time by 25% compared to *Keep-Alive* by terminating pre-warmed containers right after invocations complete. Aquatope's hybrid prediction model allows it to make fine-grained and timely adjustments to the container pool, and reduces memory time by 23% compared to IceBreaker.

Handling fluctuating load: Aquatope's dynamic pre-warmed container pool is designed to be noise-aware, making it robust to fluctuating workloads. For the Azure dataset we use, we look at the benefits of Aquatope compared to the best-performing previous work, IceBreaker [128], for loads with different coefficients of variation (CV) (standard deviation divided by the mean), as shown in Fig. 3.10. CV greater than 1 indicates significant variability in inter-arrival time [133]. For traces with CVs close to 0, Aquatope yields marginal improvement over IceBreaker. For traces with CV=1 to 4, Aquatope reduces 13%–41% more cold starts than IceBreaker, demonstrating the effectiveness of Aquatope's noise-aware approach. In the Azure dataset, more than 40% of invocation traces have CVs greater than 2, which highlights the high variability present in FaaS environments.

To further demonstrate the benefits of incorporating noise and uncertainty into the Bayesian prediction model, we also compare Aquatope with a simplified implementation without the uncertainty estimation of Sec. 3.4.2, referred to as AquaLite. The results are shown in Fig. 3.11, which shows the aggregate container memory provisioned by AquaLite and Aquatope over time, under a fluctuating load. Thanks to the uncertainty estimation, Aquatope is robust to fluctuating workloads and adjusts the pre-warmed container pool more accurately than AquaLite, reducing 3% more cold starts and saving 8% more provisioned memory.

Overhead: Aquatope's container pool scheduler makes adjustment to pre-warmed containers asynchronously, off the critical path, and does not impact the latency of function invocations. Training the hybrid model with a week's trace from Azure Function Dataset [133] takes 50s, which can easily accommodate retraining if needed. The latency of the prediction is below 10ms, which is marginal compared to the adjustment interval of the container pool.

3.8.2 Container Resource Manager

Resource efficiency of Aquatope: We first evaluate the resource efficiency of Aquatope's container resource manager, by comparing it with other resource mangers, including Random [81], Autoscaling [47,48] and CLITE [120], in which Random is the baseline policy that randomly selects sample configurations, Autoscaling is a widely adopted resource manager than adjusts resource allocation based on usage, and CLITE is the state-of-the-art BO-driven cloud resource manager that uses a manually crafted objective function to capture the goal of

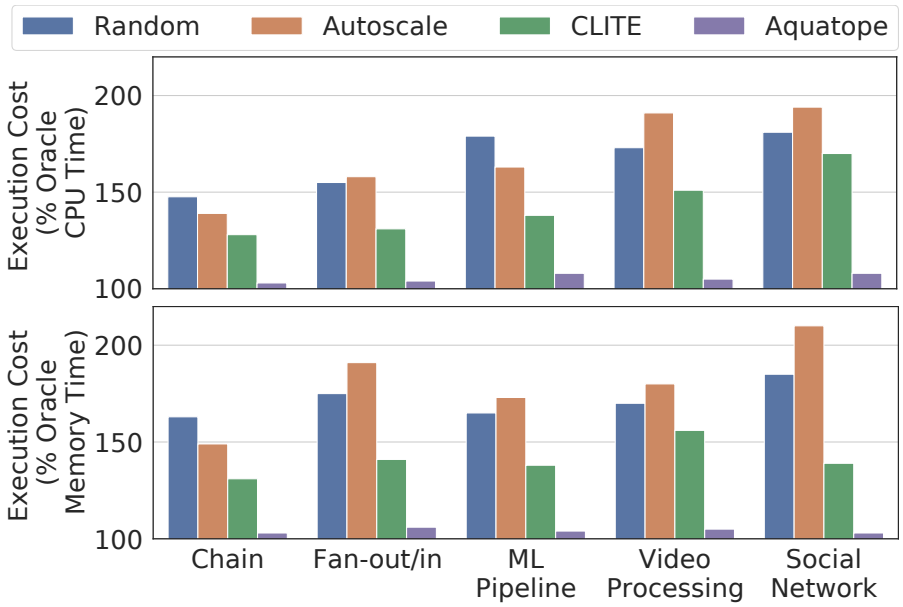


Figure 3.13: Aquatope's resource manager finds a near-optimal resource configuration across multi-stage serverless applications.

meeting QoS for latency-critical jobs, while maximizing performance for background jobs. We adopt CLITE to the FaaS setting by rewriting its objective function to minimize cost while satisfying QoS. In our experiments, a QoS violation is defined as failing to meet the end-to-end latency requirement of a serverless workflow. The QoS constraint is chosen to be the latency before saturation is reached, consistent with previous work [54, 156]. We have also conducted experiments with more or less conservative QoS settings and arrived at similar conclusions.

Fig. 3.13 shows the mean aggregated CPU and memory time of different serverless workflows, under different resource managers. Experiments are repeated 30 times, to account for system noise. For random search, we take the best of all 30 trials for evaluation, because each trial does not always find a QoS-satisfying configuration, consistent with how random search is used in prior work [44], and all the other resource managers successfully meet QoS. Under

the same time budget for resource exploration, Aquatope outperforms all other approaches across examined applications, and significantly reduces CPU and memory time. On average, Aquatope finds a near-optimal configuration with cost within 5% of the optimal configuration obtained by ORACLE, which exhaustively searches the entire allocation space. As shown in Fig. 3.13, Aquatope is not only capable of managing resources for simple applications (e.g., Chain), but can also find near-optimal configurations for complex applications (e.g., Social Network), whose functions vary widely in resource needs. Aquatope outperforms the second best resource managers, using 25%–62% less CPUs and 18%–51% less memory.

Specifically, Random selects a number of configurations to explore randomly for all stages and never learns from previous trials. In contrast, Aquatope uses a Gaussian process to model the performance of an application based on sampled configurations, and uses prior knowledge to explore the space. Autoscaling leads to increased cost for two reasons. First, it does not take into account the correlation between execution time and cost of serverless workflow. Adding resources can accelerate the computation but also raises the cost per unit of execution time. Second, it adds resources to all containers belonging to a serverless workflow, rather than only to those that need more resources, leading to overprovisioning. CLITE also results in sub-optimal cost because its manually crafted objective function does not capture the behavior of complex serverless workflows, and often gets trapped in local optima.

Fast and accurate convergence: With the customized surrogate models and acquisition functions, Aquatope is able to converge faster and more accurately than other BO-based resource managers, like CLITE, by proactively identify-

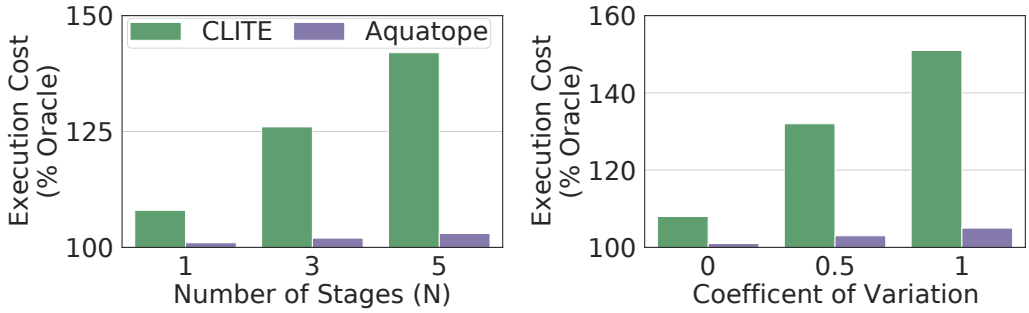


Figure 3.14: Aquatope's resource manager outperforms CLITE [120], the previous best-performing BO-driven approach, for (a) a function chain with varied number of stages, and (b) a single function workflow with varying degrees of execution time variability.

ing configurations that may violate QoS and avoiding sampling them. In addition, Aquatope's batch exploration also yields a substantial reduction in exploration time. As a result, compared to CLITE, Aquatope only spends 31% wall-clock time on average, and can find a configuration with 36% lower cost. Aquatope also converges more accurately, yielding better resource configurations. Fig. 3.12 shows the resulting cost of all evaluated resource managers for all serverless workflows at different budget levels, and Aquatope constantly converges to the most efficient resource configurations.

End-to-end QoS constraint: Aquatope handles end-to-end QoS constraints for complex workflows better than CLITE, which is the best-performing and most closely related previous work. CLITE is designed for colocated monolithic applications or multi-tier applications with defined per-tier QoS targets. However, defining per-tier QoS is a major challenge in real deployments, and most production services do not have per-tier targets. They instead define QoS only based on end-to-end latency. CLITE's hand-crafted objective function cannot capture the end-to-end performance behavior of complex workflows consist-

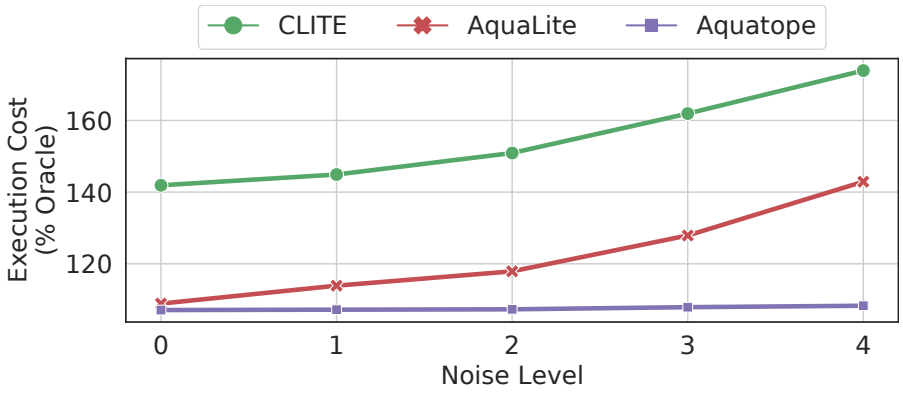


Figure 3.15: Aquatope's robustness to cloud noise.

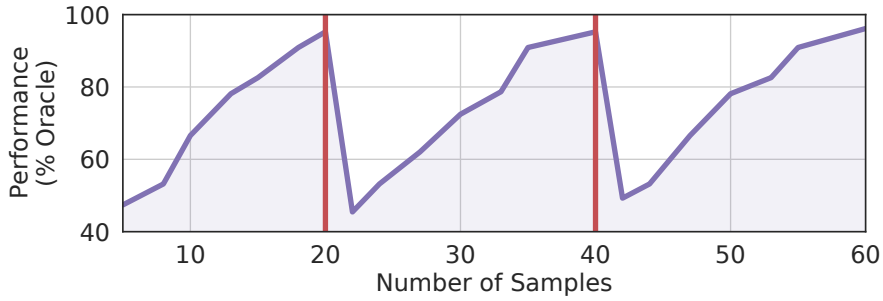


Figure 3.16: Aquatope adapts to changes in the performance model of the serverless workflow.

ing of multiple functions, whereas Aquatope's independent performance model treats the workflow as a whole, and converges faster and more accurately. As shown in Fig. 3.14a, when increasing the number of chained functions in a synthetic workflow, Aquatope outperforms CLITE in terms of execution cost by 7%–39%. This indicates that Aquatope is better at handling serverless workflows with complex topologies and end-to-end QoS constraints.

Resilience to cloud noise: A major challenge when applying Bayesian Optimization to FaaS is the noise in cloud environments, due to e.g., resource contention. If noise is not handled appropriately, resource managers can violate QoS and/or waste resources. While baseline BO can account for some noise, that is not sufficient to capture the variability of FaaS infrastructures.

Aquatope's resource manager uses customized noise-aware BO to find near-optimal resource configurations under noisy observations. We use a synthetic single function workflow with different degrees of execution time variability to evaluate the performance for Aquatope in a noisy environment. Fig. 3.14b shows that Aquatope outperforms CLITE in execution cost by 7%–45% as the inherent noise of the function increases.

As shown in Fig. 3.15, we further evaluate the robustness of Aquatope to irregular system noise by introducing intermittent background jobs [67, 116] on the same worker servers, causing noise and data outliers in the sampling process of the ML pipeline. The noise level represents the frequency and intensity of the background jobs. As the noise level increases, the number of data outliers increases and the resource manager is more likely to suffer from biased observations. Fig. 3.15 illustrates that Aquatope is still able to achieve a near-optimal configuration in the presence of noise and outliers, while CLITE experiences 37–64% increase in cost. We also compare Aquatope with AquaLite, a simplified version of Aquatope without the noise-aware components, and find that AquaLite experiences a 10–33% higher cost compared to Aquatope. This demonstrates the effectiveness of Aquatope by incorporating uncertainty into the performance model and proactively pruning data outliers.

Automatic retraining: Aquatope can detect and adapt to changes in performance behavior, which can be caused, for example, by changes in the function inputs or function updates. As shown in Fig. 3.16, Aquatope detects the change in performance behavior when the format and size of the inputs for the video processing pipeline change (marked by red lines), and updates the model dynamically by collecting new samples using a sliding window ap-

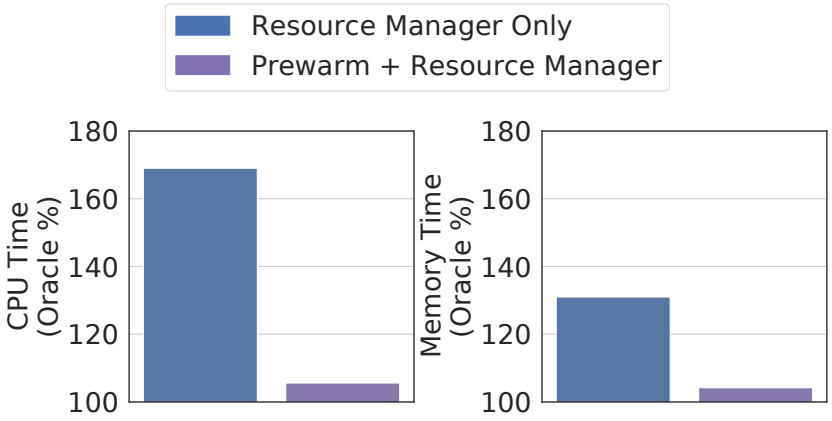


Figure 3.17: The performance impact of not having the pre-warmed container pool.

proach. Aquatope adapts to changes quickly with around 20 new samples within 2 minutes, and is always able to find a new near-optimal resource configuration.

Overhead: Aquatope's container resource manager is not in the critical path of function invocations. Functions continue to execute using their previous resource allocation configuration until Aquatope updates them. The computational overhead of Aquatope is negligible. The time to find the next batch of candidate configurations is less than 100ms, which can be masked by the time needed to evaluate the current samples.

3.8.3 End-to-End Performance

We first demonstrate that cold starts and resource usage are correlated, and therefore, cold start elimination and resource management need to be tackled jointly. Then we perform an end-to-end evaluation of Aquatope, including both the pre-warmed container pool and resource manager.

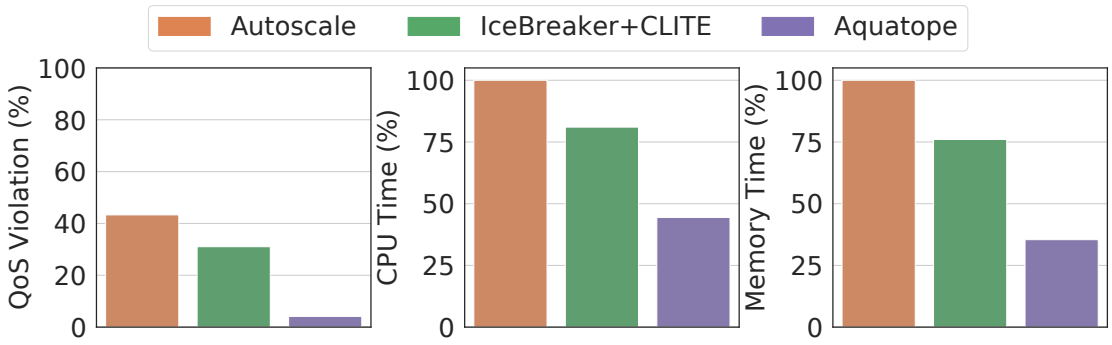


Figure 3.18: End-to-end performance analysis for Aquatope compared to autoscaling policies and a combination of the best prior work.

We demonstrate the aforementioned correlation, by showing that the resource manager cannot achieve the desired performance without reducing the resource allocation search space to correspond only to warm start containers. Fig. 3.17 shows the resulting average CPU and memory time of a fully fledged Aquatope with both the pre-warmed container pool and resource manager, and a simplified Aquatope with only the resource manager in place, compared to the offline oracle. Compared to the fully fledged Aquatope, the simplified version experiences a 64% increase in CPU time and 28% increase in memory time. This is due to the diverse behavior of cold and warm starts leading to different resource requirements, and the simplified version of Aquatope being forced to strike a balance between them, leading to degraded performance. This indicates the necessity of jointly tackling cold starts and resource management in FaaS.

We then perform an end-to-end analysis of the full-fledged Aquatope. Specifically, we compare Aquatope to a framework using the autoscaling-based FaaS resource manager [16, 21, 143] that scales both pre-warmed containers and allocated resources, and a framework combining the container pre-warming mechanism in IceBreaker [133] with the BO-based resource manager in CLITE [120] (IceBreaker+CLITE), which are the best-performing alternatives

based on Section 3.8.1-3.8.2. In our experiments, the average CPU utilization is 43% and the average memory utilization is 29%, which is consistent with the resource utilization of production clusters [135,155]. Fig. 3.18 shows the total CPU and memory time of all evaluated frameworks. IceBreaker+CLITE outperforms autoscaling by reducing 13% of QoS violations, 19% of CPU time, and 25% of memory time. In contrast, Aquatope:

1. Outperforms other approaches, eliminating another 27%–39% of the QoS violations, and bringing the total to below 3%.
2. Significantly reduces CPU and memory usage, reducing CPU time by 37%–55%, and memory time by 41%–64%.

Aquatope achieves these benefits by jointly tackling cold start elimination and resource management, and using Bayesian models that adjust to the behavior of a given application, while remaining general and robust to cloud noise.

3.9 Conclusion

We have presented Aquatope, a QoS-and-uncertainty-aware resource manager for multi-stage serverless workflows. Aquatope jointly tackles the challenges of cold starts and resource management; the former through the use of a hybrid Bayesian neural network and the latter using customized Bayesian Optimization. Across a diverse set of real-world serverless applications, Aquatope meets QoS, while significantly reducing the amount of required resources.

CHAPTER 4

METEION: FAST AND EFFICIENT SERVERLESS WORKFLOWS FOR LATENCY-CRITICAL INTERACTIVE APPLICATIONS

4.1 Introduction

Despite efficient resource management approaches, serverless workflows still suffer from significant control plane and communication overheads, making them a poor fit for latency-critical interactive applications. In a serverless framework, the control plane is responsible for workflow orchestration, load balancing, scheduling, and function communication, while functions encapsulated in containers or VMs are executed on the worker servers. Existing serverless workflow frameworks rely on a centralized workflow engine to perform workflow orchestration [4, 6, 7, 87, 105], and use the control plane as the medium for inter-function communication, resulting in constant interleaving between the control plane and functions during workflow execution. This leads to substantial control plane and communication overheads, which remain on the critical path and significantly impact the end-to-end latency of serverless workflows, further limiting their applicability to latency-critical interactive applications.

We present *Meteion*, a serverless workflow framework that enables fast and efficient execution of latency-critical interactive applications. Instead of relying on a centralized orchestrator to handle each function invocation in a workflow, Meteion enables decentralized workflow orchestration by deploying a lightweight per-function workflow engine within the runtime environment of each function. This helps Meteion effectively decouple the control plane from the workflow execution process, which allows for direct inter-function commu-

nication and fast in-situ workflow execution. To ensure fault tolerance with decentralized orchestration, Meteion persists the workflow state in a causal manner to maintain consistency, and reserves a control plane workflow engine that monitors the workflow execution state and intervenes in the critical path only when a failure or straggler task occurs and triggers the failure recovery mechanism. Finally, Meteion's DAG scheduler utilizes the workflow's latency distribution and graph structure to provision function containers in a timely manner, ensuring that the workflow can seamlessly execute on the worker servers without falling back to the control plane.

We evaluate Meteion on two real-time data processing pipelines ported from AWS Step Functions [33, 34] and two interactive web applications from DeathStarBench [72]. In all cases, Meteion outperforms the baseline workflow engines [4, 87, 105], reducing the end-to-end latency by 37%–75% and the provisioned memory time by 9%—23% compared to prior work [87,105], significantly improves the performance of latency-critical interactive workflows.

This chapter makes the following main contributions:

- Meteion is a fast and efficient serverless workflow engine that enables latency-critical, multi-tier services to run as serverless workflows with minimal system overhead, while adhering to the serverless abstraction.
- Meteion's system design effectively decouples the control plane overheads from the workflow execution process, enables decentralized workflow orchestration, direct inter-function communication, and fast in-situ workflow execution, while remaining fault tolerant.
- Meteion's scheduler leverages the latency distribution of a workflow to perform delay workflow scheduling, which provisions function contain-

ers in a timely manner to minimize the end-to-end latency and improve resource efficiency.

4.2 Background and Motivation

4.2.1 Problem Statement

Many real-world serverless applications are implemented as *serverless workflows*, in which incoming user requests invoke sets of serverless functions that coordinate with each other to execute application logic. By splitting a complex application into dependent but loosely-coupled functions, the application benefits from elastic scaling, load-based billing, and modular development [52, 152]. However, existing serverless workflow frameworks [4, 6, 7] suffer from the control plane and inter-function communication overheads, which both stay on the critical path and contribute to the end-to-end latency of a workflow. For current frameworks, a single interaction between functions can incur latency ranging from tens to hundreds of milliseconds [52, 87, 152], making them a poor choice for latency-critical interactive applications.

Meteion's target is to reduce these control plane and communication overheads. It is a serverless workflow engine that provides fast and efficient execution for applications with stringent latency requirements. Meteion decouples the control plane from the workflow execution process, and enables fast and efficient in-situ workflow execution with direct inter-function communication, significantly reducing the function interaction latency. These capabilities empower Meteion to implement high-performance serverless workflows, making

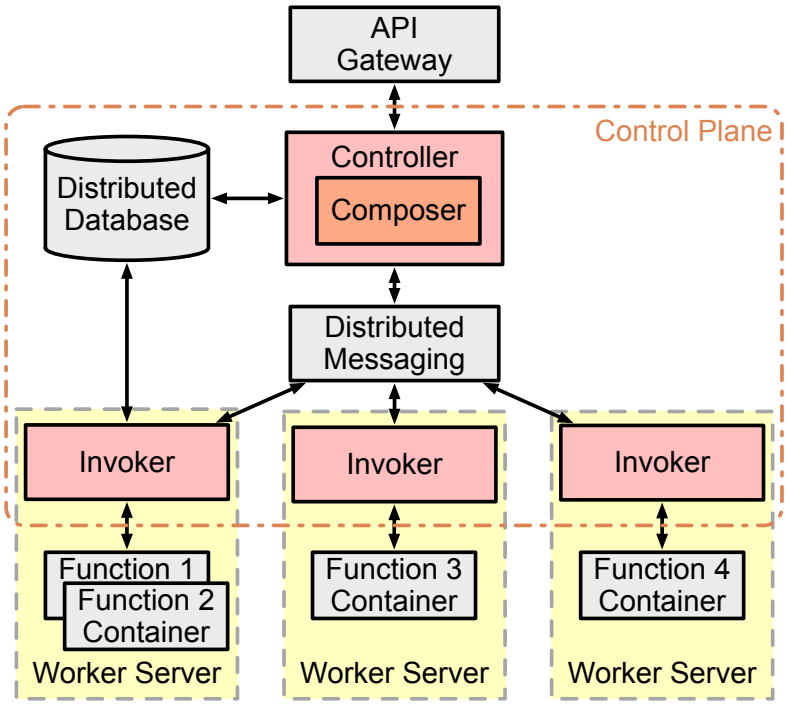


Figure 4.1: Architecture of Apache OpenWhisk. The control plane consists of the controller and invokers.

it suitable for latency-critical interactive applications.

4.2.2 Workflow Engine Architecture

In a serverless workflow engine, the *control plane* is responsible for updating and persisting the workflow execution state, handling the workflow logic, invoking functions in a workflow, and routing invocation requests to the underlying containers or VMs. Function instances running on the worker servers are responsible for the actual execution of functions, using sandboxes, such as containers and VMs.

We use Apache OpenWhisk [3], a widely-used, open-source FaaS platform that powers IBM's Cloud Functions [17], as an example to demonstrate a work-

flow's execution process. Figure 4.1 shows the architecture of OpenWhisk [3]. The control plane of OpenWhisk consists of the Controller and a number of Invokers, and functions instances are encapsulated with Docker containers.

OpenWhisk exposes an API gateway, implemented using Nginx [27], for users to interact with the FaaS platform. The API gateway forwards user requests to the Controller, which performs load balancing and selects an Invoker to execute the function invocation. The workflow orchestration is handled by the Composer layer [4] in the Controller. The Composer monitors and updates the execution state of a workflow, handles the workflow control logic (e.g., function chain, fan-out/in, conditional branch), and invokes functions in a workflow after their predecessors have all completed. The function invocation requests are sent to the selected Invoker via the distributed messaging system, implemented using Kafka [2]. The Invoker uses Docker containers to fulfill the function invocation requests, and stores the invocation results to the distributed database, implemented using CouchDB [1]. The Composer waits for the function invocation response from the message bus and polls the data store to ensure state persistence, and then invokes the function of the next workflow stage accordingly.

In OpenWhisk, functions in a workflow can only interact and communicate with each other indirectly through the control plane. This leads to the interleaved execution shown in Figure 4.5. After execution, each function instance always needs to contact the control plane to trigger the next function. Consequently, the control plane and communication overheads become part of the critical path of each function invocation. The interleaving of the control plane and functions during workflow execution is universal in state-of-the-practice

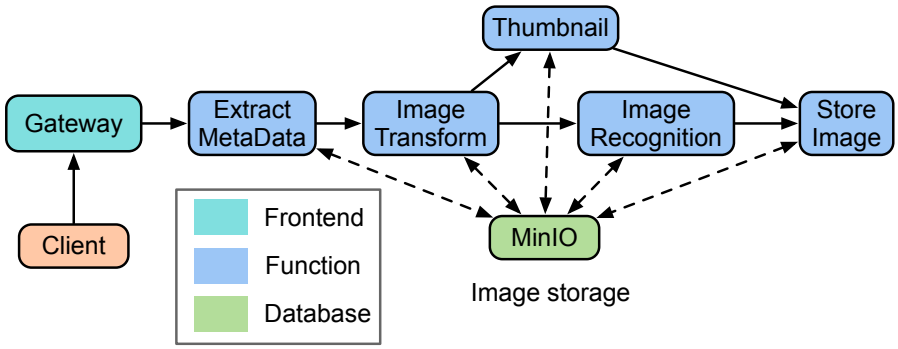


Figure 4.2: Image pipeline [33].

serverless workflow engines [4, 6, 7, 15, 52, 87, 105, 138].

4.2.3 Serverless Workflow Benchmarks

To investigate the performance bottlenecks of serverless workflow execution, we select and implement two representative serverless workflow applications from AWS Step Functions [6], and port three interactive web applications from DeathStarBench [72] as our benchmarks.

File processing: We implement an event-driven, real-time file processing application based on AWS Lambda samples [34]. This application delivers notes in Markdown format from the database and then converts them to HTML and performs sentimental analysis [147] in parallel.

Image pipeline: We implement a serverless image processing pipeline based on AWS Step Functions samples [33]. Figure 4.2 shows the architecture of the image pipeline workflow. It processes photos uploaded to the MinIO [25] object store and extracts metadata from the image, uses image recognition [83] to tag objects in the photo, and produces a thumbnail of the photo.

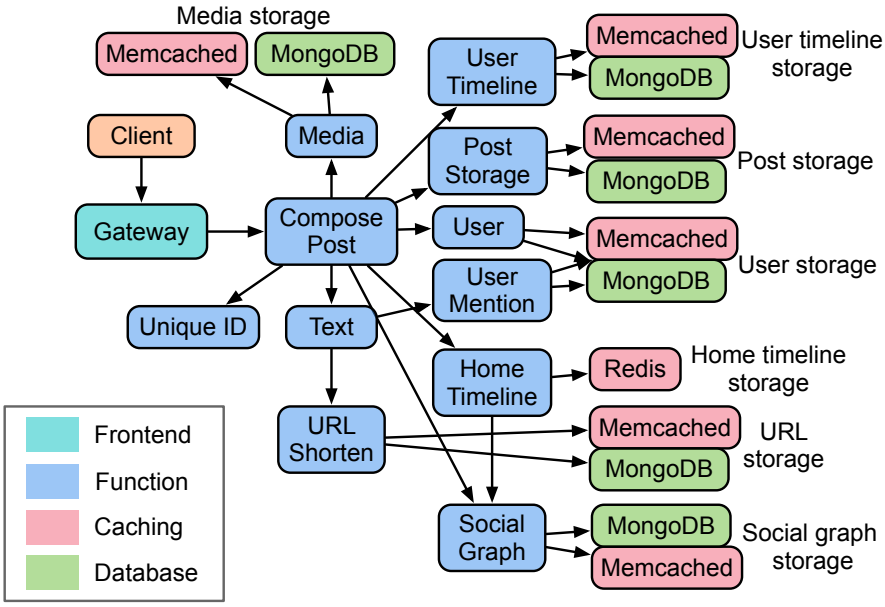


Figure 4.3: Serverless social network [72].

Interactive web applications: We use serverless implementations of the *Hotel Reservation* and *Social Network* applications in DeathStarBench [72]. *Hotel Reservation* is an online hotel reservation site for browsing hotel information and making reservations. Figure 4.3 shows the architecture of *Social Network*. Users can create posts embedded with text, media, links, and tags, which are then broadcast to all their followers. Users can also read posts on their timelines. The backend uses Memcached and Redis for caching, and MongoDB for persistent storage.

4.2.4 Control Plane and Communication Overheads

We deploy Apache OpenWhisk [3] in a dedicated local cluster with 7 nodes and use the benchmarks in Section 4.2.3 to study the control plane and communication overheads in the serverless workflow execution. The cluster specification is described in Section 4.4.3. We adjust the size of the pre-warmed container

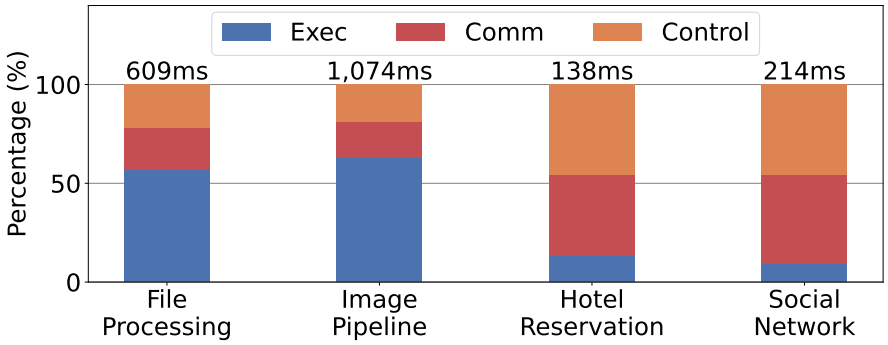


Figure 4.4: Breakdown of the mean latency of each serverless workflow to execution time, communication time, and control plane time.

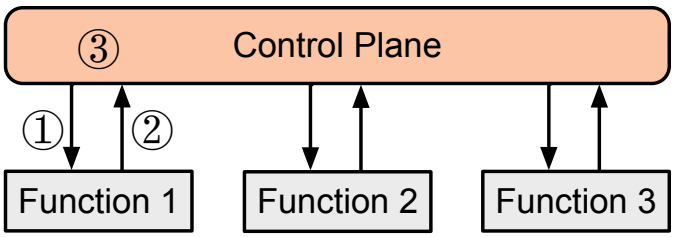


Figure 4.5: Interleaved execution of serverless workflows. Functions can only communicate with each other indirectly through the control plane.

pool [16] so that warm containers handle all function invocations to eliminate cold start overheads.

As shown in Figure 4.4, we decompose the mean end-to-end latency of workflow execution into the time spent in the control plane, network communication, and function execution. Across all benchmarks, the control plane and communication overheads account for a significant portion of the workflow latency: 37%–43% for real-time data processing workflows including File Processing and Image Pipeline; and up to 91% for interactive web applications that are less compute-intensive, including Hotel Reservation, Social Network, and Movie Review. The control plane and communication overheads are the dominating factors of the end-to-end latency in interactive serverless workflows, where function execution times are typically short [133].

The high control plane and communication overheads stem from the interleaved workflow execution pattern in existing serverless workflow engines, as shown in Figure 4.5, where functions can only indirectly communicate with each other via the control plane. For each function in the workflow, the control plane needs to first invoke that function (①), and then wait for the invocation results to be retrieved (②). Thus, each function invocation is accompanied by two interactions with the control plane.

The communication between the function instance and the control plane typically consists of multiple network hops and indirect communications via the message queue. In addition to the network communication overhead, the control plane also performs function scheduling, load balancing, handling workflow orchestration, and persisting workflow state (③), all of which are part of the critical path of function invocations and contribute to the end-to-end latency of a serverless workflow. To enable fast workflow execution for latency-critical interactive applications, the control plane must be decoupled from the workflow execution process, and functions should be allowed to interact and communicate with each other directly.

4.3 Meteion Design

Meteion is a fast and efficient serverless workflow engine for latency-critical, interactive applications. Figure 4.6 shows the architecture of Meteion's workflow framework. It contains three major components: (1) a *per-function workflow engine* embedded in the function runtime, which enables decentralized workflow orchestration, direct inter-function communication, and fast in-situ workflow

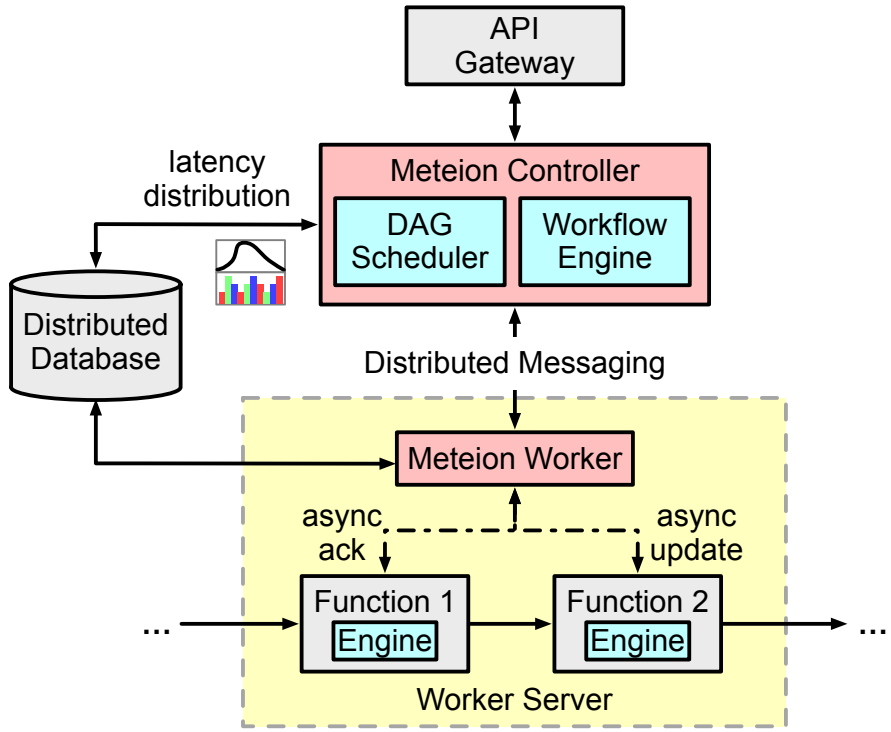


Figure 4.6: Meteion system architecture.

execution with minimal interventions from the control plane; (2) a *control plane workflow engine* that serves as a supervisor that monitors the state of workflow execution, and only intervenes in the critical path when failures or straggler tasks occur and trigger fault recovery mechanisms; (3) a *DAG scheduler* that leverages the latency distribution of a workflow to perform delay workflow scheduling, which provisions function containers in a timely manner to minimize the end-to-end latency and improve resource efficiency.

4.3.1 Design Principles

Meteion's design is based on the following principles:

- **Be serverless and be scalable.** Meteion follows the serverless abstraction of using fine-grained container-level isolation to handle function invocation requests, and imposing no additional restrictions on the placement of containers (e.g., enforcing functions to execute on the same worker server or VM). This allows Meteion to provide fast and efficient workflow execution, while embracing the benefits of FaaS, including fined-grained scheduling, better load balancing, per-invocation billing, and scalability.
- **Decouple control plane from workflow execution process.** As discussed in Section 4.2.2, existing serverless workflow frameworks rely on a centralized workflow engine to perform workflow orchestration, which makes control plane operations part of the critical path of function invocations in a workflow. Meteion implements decentralized workflow orchestration, effectively decoupling the control plane from the critical path of workflow execution. This allows Meteion to eliminate the control plane overhead and enable fast in-situ workflow execution.
- **Enable efficient direct inter-function communication.** Conventional serverless workflow engines rely on the control plane as the medium for inter-function communication, resulting in interleaved workflow execution. Instead of suffering from excessive communications between the control plane and function instances, Meteion enables direct inter-function communication, which can significantly reduce the network communication overhead, and allows functions to seamlessly execute on the worker servers, without falling back to the control plane.
- **Optimize the common case and make the uncommon case no worse.** Meteion optimizes deterministic workflows where the majority of functions execute without causing faults. If a workflow is non-deterministic,

i.e., the function DAG changes depending on some inputs, for the workflow's subgraphs that are non-deterministic, Meteion falls back to the control plane for scheduling decisions. In practice, such cases happen rarely across the large number of applications we examined. Even in those cases, Meteion's performance for those functions will be no worse than the default control plane.

4.3.2 In-Situ Workflow Execution

Meteion enables decentralized workflow orchestration, direct inter-function communication, and removes most of the control plane overhead from the critical path of workflow execution. We first introduce Meteion's components that enable fast in-situ execution of a serverless workflow that is asynchronous from the control plane, and then describe the end-to-end in-situ workflow execution process.

Function runtime. The architecture of Meteion's function runtime is shown in Figure 4.7, which is inspired by the design of existing FaaS runtimes [3, 28] that separate function proxy and function process. The function proxy is an HTTP server that acts as a reverse proxy of the function process. When the function proxy starts, it forks to create the function process that runs the application code provided by the user. The function proxy interacts with the function process through a full-duplex message channel consisting of two Linux pipes in opposite directions. Function initialization and invocation requests received by the function proxy are forwarded to the function process, which then initializes and executes the function accordingly. In conventional FaaS platforms, only the

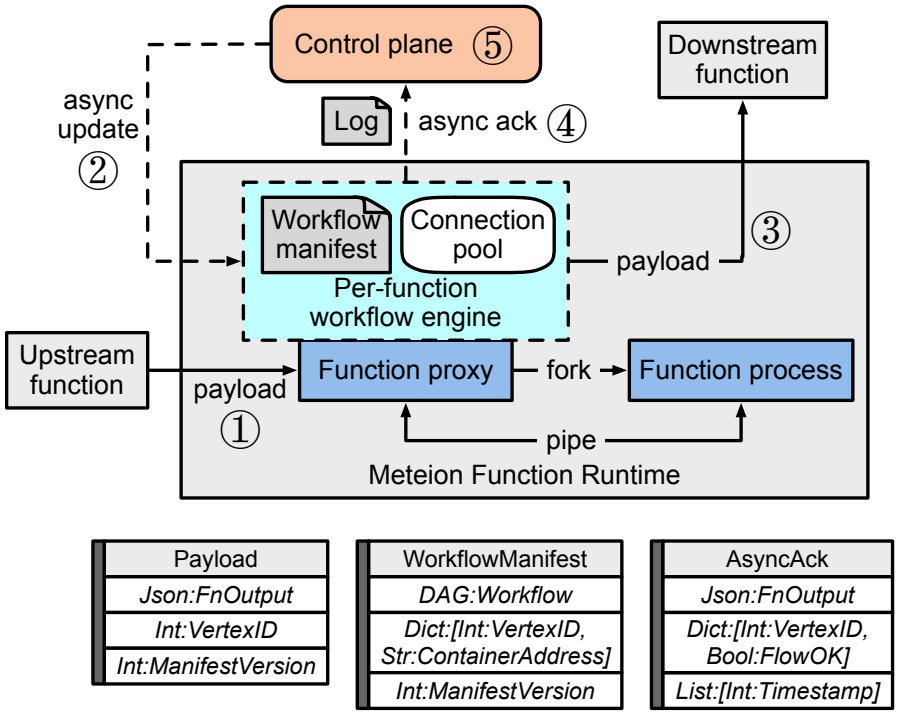


Figure 4.7: Meteion function runtime.

control plane can send invocation requests to the function proxy, and functions in a workflow can only interact with each other indirectly through the control plane. Meteion's function runtime uses a built-in per-function workflow engine to bypass the control plane, and enable direct, low-latency function interaction.

Per-function workflow engine. The rationale of inserting a lightweight per-function workflow engine into the function runtime is that communication between functions does not need to be implemented as indirect message exchanges with the control plane, nor does workflow orchestration need to be synchronous with function execution, which causes it to be on the critical path of workflow execution.

Instead of returning to the control plane after each function invocation, Meteion enables direct inter-function communication between function in-

stances by passing the *workflow manifest*, which includes a subset of workflow's DAG and corresponding function container addresses, to the per-function workflow engine. Container addresses are obtained by the control plane, which instantiates functions containers and performs container orchestration. With Meteion's runtime, a function can be directly invoked by its upstream function. The upstream payload (Figure 4.7 ①) consists of the output of the upstream function, its vertex id in the workflow DAG, and the version number of the workflow manifest. The per-function workflow engine uses the vertex id and workflow manifest information to process the control logic and determine which downstream function to connect with.

To ensure that the workflow can seamlessly execute across function containers, the control plane sends asynchronous update (Figure 4.7 ②) requests to each per-function workflow engine to keep workflow manifest up-to-date. The per-function engine always uses the latest version of a workflow's manifest for workflow execution. If the version number of the workflow manifest stored in the engine is outdated or the downstream function is not available, the transaction fails, and the engine yields control back to the control plane. When the function finishes execution, the per-function workflow engine directly sends invocation requests to downstream functions running on the worker servers (Figure 4.7 ③). Inter-function connections are cached in the connection pool of the per-function workflow engine to reduce inter-function communication latency.

Finally, the engine sends an asynchronous ack payload (Figure 4.7 ④), which contains the function's invocation result, downstream connection result, and tracing timestamps, to the control plane to update and persist the workflow state (Figure 4.7 ⑤). A workflow manifest does not necessarily include the ad-

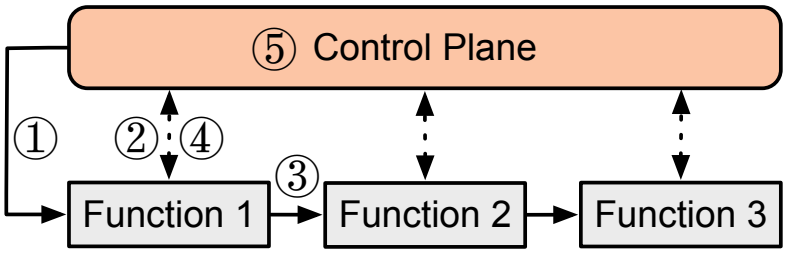


Figure 4.8: In-situ workflow execution of Meteion. Most control plane orchestration operations are asynchronous and off the critical path.

dresses of all function containers. If, when the function finishes, the containers of the downstream functions are not available, the data plane workflow engine forwards the request to the control plane, which performs the necessary orchestration to continue the workflow execution.

Control plane workflow engine. By decoupling the control plane from the workflow execution process, Meteion's control plane no longer acts as the medium of inter-function communication, and is mostly absent from the critical path.

After initiating the execution of a workflow, the control plane workflow engine only needs to asynchronously monitor the execution state of the workflow, handle workflow orchestration and update the workflow manifest in the function runtime. The control plane workflow engine uses the per-invocation tracing log in the async ack (Figure 4.7 ④) to update the workflow state and restart functions in case of failures. Tracing logs and workflow states are stored and persisted in the distributed database. The control plane workflow engine relies on the DAG scheduler (Section 4.3.4) to provision function containers and update the workflow manifest in a timely manner so that the workflow continues to execute in the data plane without falling back to the control plane.

In-situ workflow execution. The joint design of the control and per-function workflow engines allows for fast in-situ execution of serverless workflows. For each workflow invocation request, the control plane workflow engine provisions the startup containers and initiates workflow execution (Figure 4.8 ①). The workflow then continues its execution among function containers running on the worker servers, based on the workflow manifest, which can be updated asynchronously by the control plane (Figure 4.8 ②). Each function in the workflow can directly communicate with its downstream function using the per-function workflow engine (Figure 4.8 ③). The function invocation result is sent to the control plane via async ack (Figure 4.8 ④), and the control plane workflow engine updates and persists the workflow execution state accordingly (Figure 4.8 ⑤).

Overheads. For the most complex of the examined applications, the social network workflow composed of tens of functions, it takes at most 306 bytes to encode its workflow manifest, which adds only a small amount of data to the function payload. Parsing the workflow manifest and processing a workflow's control logic in the per-function workflow engine only takes 13 µs, which is negligible compared to the control plane overhead.

4.3.3 Fault Tolerance and Speculation

A crucial reality of cloud environments is the ubiquity of failures: functions may fail or time out, worker servers may crash and restart, and connections between services may be temporarily disrupted. To make the workflow system resilient to faults and errors, and to allow for faults recovery, we need to guaran-

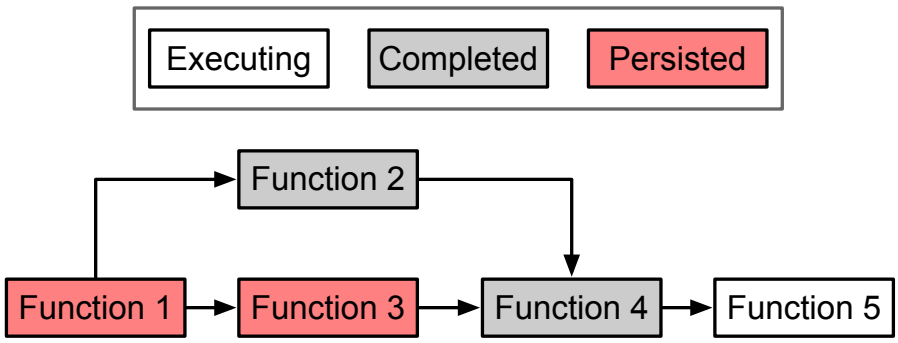


Figure 4.9: Meteion's control plane persists workflow state with causal consistency guarantees.

tee *exactly-once* semantics for the system-internal state, so that functions appear to execute exactly-once, even if individual functions in the workflow can crash and rerun multiple times.

Causally consistent state persistence. A centralized serverless workflow engine (Section 4.2.2) can naturally provide this exactly-once guarantee by choosing the persisted invocation result from the execution of the same function, and using it as the input for all downstream functions. Meteion employs a per-function workflow engine in each function runtime, which enables decentralized workflow orchestration and fast in-situ workflow execution. A key challenge for Meteion is to provide the same execution semantics without falling back to a centralized workflow engine.

As shown in Figure 4.9, Meteion's control plane updates and persists the workflow state with causal consistency guarantees. Meteion's worker only commits and persists a function's execution state if all its predecessors have been persisted. Functions in executing or completed state can crash and get aborted. In such case, Meteion aborts all functions that causally depend on the aborted function, ensuring that all system-internal effects that causally depend on the previous execution are aborted. In a complete execution, each output message

produced by an upstream function is consumed by exactly one non-aborted function.

Speculative workflow execution. Having this causal consistency guarantee allows us to enable speculation. Instead of conservatively waiting for the state of each function invocation to be persisted to the data store, Meteion can speculatively execute a workflow. The downstream function can proceed even if it causally depends on a not-yet-persisted upstream function. This can significantly boost the performance of a workflow since state persistence overhead can be effectively removed from the critical path, while the workflow remains fault-tolerant. For non-deterministic workflow (e.g. conditional branch) whose control logic relies on function output, Meteion pauses speculative execution and waits for the control plane to persist function output and process workflow logic.

Limitations on external effects. It should be noted that functions can produce external effects that are beyond the control of the cloud provider. For example, functions can perform non-idempotent operations on external databases. This issue is universal and is independent of the workflow engine architecture. Meteion focuses on maintaining causal consistency for system-internal states, and does not forbid functions from using user-level libraries [86, 154] that can resolve this problem.

4.3.4 DAG Scheduler

To ensure fast in-situ workflow execution and to bypass the control plane overhead, it is crucial to provision containers ahead of function invocation, and pro-

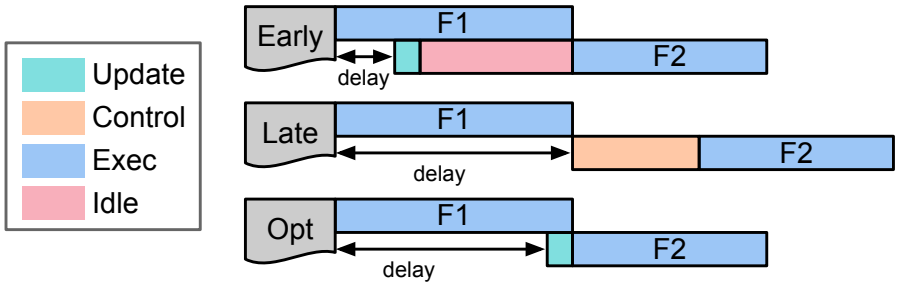


Figure 4.10: Performance of different provisioning decisions. By selecting the near-optimal provisioning delay, Meteion allows for low end-to-end latency with high resource utilization.

vide timely updates to the workflow manifest, so that the per-function engine can seamlessly interact with the downstream functions without falling back to the control plane.

Figure 4.10 illustrates the performance implications of different provisioning decisions. We define *UpdateTime* as the time it takes to update the workflow manifest in the function runtime. *ControlTime* represents the overhead incurred by the control plane when the downstream function is unavailable, and the per-function workflow engine delegates workflow orchestration to the control plane. *IdleTime* denotes the duration between the completion of function container provisioning and the initiation of function execution. Early provisioning of the downstream function (F2) results in a longer *IdleTime* and lower resource utilization. On the other hand, late provisioning of the downstream function forbids the upstream function (F1) from directly communicating with its downstream function, forcing workflow execution to fall back to the control plane, leading to increased end-to-end latency.

Meteion's DAG scheduler leverages the latency distribution and the DAG structure of the workflow to determine when to provision function containers and update the workflow manifest in the function runtime accordingly. Meteion

Input: workflow graph G, functions $f_1, \ldots, f_n$, resource utilization threshold U, latency threshold T.
Output: provisioning delay vector $\vec{d} = [d_1, \ldots, d_n]$.

$\vec{d} \leftarrow InitProvisionDelayVector(G)$;
$pq \leftarrow PriorityQueue()$;
$s_0 \leftarrow InitConfig(\vec{d})$;
$pq.insert(s_0)$;
while *pq is not empty* **do**
  $s_{current} \leftarrow pq.pop()$;
  for $f_i \in TopologicalSort(G)$ **do**
    $s_{next} \leftarrow s_{current}$;
    $s_{next}.\vec{d}[i] \leftarrow s_{next}.\vec{d}[i] + timestep$;
    $s_{next}.latency \leftarrow GetLatency(s_{next})$;
    $s_{next}.util \leftarrow GetUtil(s_{next})$;
    $s_{next}.priority \leftarrow (T - s_{next}.latency) \times s_{next}.util$;
    if $s_{next}.latency \leq T$ *and* $s_{next}.util \geq U$ **then**
      $\vec{d} \leftarrow s_{next}.\vec{d}$;
      break;
    end
    else if $s_{next}.latency \leq T$ *and* $s_{next}.util < U$ **then**
      $pq.push(s_{next})$;
    end
  end
end
return $\vec{d}$;

Algorithm 1: Meteion's Delay Workflow Scheduling

uses *delay workflow scheduling* to make provisioning decisions, minimizing the end-to-end latency, while reducing provisioned resources.

Distribution-based workflow performance modeling. The DAG scheduler relies on the performance model of the workflow to make accurate provisioning decisions for each function in the workflow. Since the variance in execution time among invocations of the same workflow can be significant [112], it is insufficient to rely on a single numerical value (e.g., mean or 99th percentile latency) to capture the performance characteristics of the workflow. To accurately measure the performance impact of different provisioning decisions, Meteion's DAG scheduler uses timestamps in tracing logs to model the latency of each

function in a workflow, the control plane overhead, and the workflow manifest update time as separate distributions, and derives the end-to-end latency accordingly.

We define the *provisioning delay* of a function as the duration between the start of the workflow invocation and the beginning of container provisioning for that specific function. Given a workflow graph G with n stages, the scheduler needs to select the provisioning delay vector $\vec{d} = [d_1, \ldots, d_n]$ that can provide a timely update to the workflow manifest for each stage in the workflow. Given the delay vector $\vec{d}$, we employ the Monte-Carlo method on the distribution-based performance model to determine the distribution of the end-to-end latency.

$$Util(G) = \frac{\sum_{f \in G} [Busy(f) \cdot Resource(f)]}{\sum_{f \in G} \{[Busy(f) + Idle(f)] \cdot Resource(f)\}} \tag{4.1}$$

To avoid early provisioning, we also need to consider resource utilization. Resource configurations can vary a lot across different execution stages of the same workflow. We use Eq. 4.1 to estimate the resource utilization of a workflow graph G, where $BusyTime$ includes both provisioning and execution times of function f. Similarly, we can get the average resource utilization for a given delay vector $\vec{d}$ based on the performance model.

Note that this is a different problem from eliminating cold starts, where the system predicts future load to pre-warm containers. Meteion utilizes the DAG structure and invocation time of a workflow to predict when downstream functions will be invoked, instead of when user requests will arrive, which improves container utilization and reduces the number of containers that need to be pre-

warmed. To further improve performance, Meteion can be coupled with mechanisms predicting load patterns [68, 111, 128, 158].

Delay workflow scheduling. Meteion's DAG scheduler provides just-in-time container provisioning and manifest updates for each function in the workflow to ensure in-situ workflow execution. The scheduler's pseudo-code is shown in Algorithm 1. For a given workflow graph G, the scheduler needs to choose a provisioning delay vector $\vec{d}$ that can meet the user-specified latency objective T, while satisfying the resource utilization target U. We take a heuristic approach to find a configuration in the configuration space that can satisfy the performance targets. The algorithm starts with a priority queue containing an initial configuration $\vec{d}$, where d_i is set as the lower bound of the provisioning delay for f_i. For each candidate configuration, we extend it by adding a timestep to the provisioning delay of each stage in the workflow in topological order. The priority is calculated as the difference between the target end-to-end latency threshold and the predicted latency, multiplied by the resource utilization. The intuition of our heuristic approach is that both the end-to-end latency and the resource utilization monotonically increase as the provision delay increases, and we need to strike a balance between them in the search process, i.e., maximize utilization while meeting the end-to-end latency target. These distributions get updated during runtime, if observed data deviate from estimations.

The DAG scheduler provisions containers in a workflow based on the provisioning delay vector. For each container, we maintain a list of its most-recently-used downstream containers. If there are multiple free containers in the container pool, we give priority to the one in the list of most-recently-used containers of its upstream function's container, so that the cached inter-function

connection can be reused to lower communication latency. If there are no free containers available, we will pre-warm a new container. Finally, Meteion updates the workflow manifest in the per-function workflow engine of the target function's predecessor.

Overheads. The pay-as-you-go pricing model of FaaS dictates that cloud providers need to collect and record performance metrics for function invocations. Meteion's DAG scheduler leverages existing latency data to build distribution-based performance models and does not incur additional storage overhead. Like other control plane components, the scheduler stays off the critical path and does not impact the latency of function invocations.

4.4 Methodology

4.4.1 System Implementation

Figure 4.6 shows the architecture of the Meteion workflow engine. We use Kubernetes [20] as the underlying container orchestration framework to handle container scheduling. We implement the distributed messaging system using Redis message broker [30]. The distributed database is implemented using MongoDB [26]. Meteion function runtime and data plane workflow engine are implemented in Go. Control plane components (i.e., control plane workflow engine, DAG scheduler, Meteion worker) are implemented in Python and the gevent library [12].

4.4.2 Benchmark Applications

We use two real-time data processing serverless workflows (*File Processing*, *Image Pipeline*) and two interactive multi-tier, web application workflows (*Hotel Reservation*, *Social Network*) described in Section 4.2.3. For the interactive web applications, we evaluate the performance gap between serverless workflows and microservices in DeathStarBench [152], using their microservice implementations based on remote procedure calls (RPCs). We also create a synthetic workflow generator to synthesize customized workflow DAGs (e.g., a function chain with an arbitrary number of stages) to study the performance of workflows of different topologies. For workload generation, we use the same methodology described in Section 3.7.2.

4.4.3 Server Cluster

We deploy Meteion on a dedicated, local Kubernetes cluster. The cluster setup is the same as described in Section 3.7.3.

4.4.4 Comparison Baselines

We compare the performance of Meteion with multiple serverless workflow engines.

OpenWhisk Composer [4]: A widely-used, open-source FaaS workflow engine that powers IBM's Cloud Functions [17]. The architecture of OpenWhisk Composer is described in Section 4.2.2. The control plane of OpenWhisk Composer

is closely coupled with function containers in the workflow execution process, which is described in Section 4.2.4.

Nightcore [87]: Nightcore runs all functions of a workflow on the same worker server and uses shared memory to bypass the network stack and accelerate communication between local functions. However, Nightcore lacks an efficient cross-server communication mechanism: when function containers are scheduled on multiple worker servers, Nightcore falls back to the API gateway for inter-function communication, resulting in interleaved execution between the control plane and functions, increasing control plane overhead.

FaaSFlow [105]: A workflow engine designed for scientific FaaS workflows. FaaSFlow bundles functions in a workflow into groups based on memory consumption and assigns them to per-worker workflow engines to reduce the scheduling overhead. Colocated functions communicate with each other indirectly through each worker's workflow engine. For cross-worker communication, the local engine needs to pass the execution state of the workflow to the remote worker engine.

4.5 Evaluation

4.5.1 End-to-End Latency Breakdown

We first study the breakdown of the end-to-end latency of workflows running on Meteion. We deploy each benchmark on our cluster, we invoke each workflow 1000 times to account for system noise, and we record the latency of each

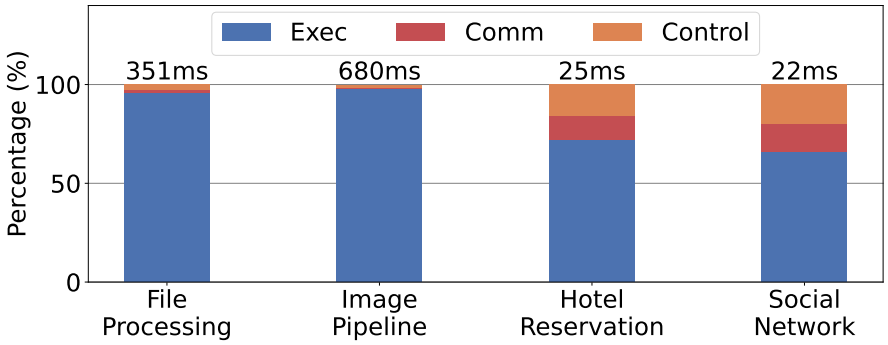


Figure 4.11: Meteion's mean latency breakdown.

invocation. Figure 4.11 shows the mean latency breakdown of Meteion, which consists of function execution time, inter-function communication time, and control plane overhead. For the real-time data processing workflows, the function execution time dominates end-to-end latency, while inter-function communication and control plane overheads account for only a marginal percentage of the end-to-end latency (1.0%–1.3% and 2.1%–3.2%, respectively). For the interactive web applications, control plane overheads account for 9%–20% of the end-to-end latency, which is significantly lower compared to the 83%–91% fraction in the OpenWhisk Composer (Section 4.2.4). In-situ function execution corresponds to 66%–80%, and inter-function communication accounts for 11%–14% of the end-to-end latency.

Compared to OpenWhisk Composer, Meteion achieves 37%–43% lower latency for real-time data processing workflows, and 78%–90% lower latency for interactive web applications. The underlying reason is that Meteion significantly reduces the function interaction overheads. By decoupling the control and data planes, Meteion manages to avoid falling back to the control plane for all function interactions, and removes most of the overheads associated with it from the critical path. By enabling direct inter-function communication, Meteion allows for fast in-situ data plane workflow execution, which substan-

tially improves the performance of latency-critical workflows.

4.5.2 Performance in Distributed Environments

We now focus on Meteion's performance in distributed settings with varied numbers of worker servers, and compare it to previous systems. We use a synthetic hello-world function sequence with 10 stages, each with minimal computation, to emphasize the overheads of workflow orchestration and inter-function communication. To ensure functions are distributed across the cluster, we enforce the placement of function containers in the workflow so that each worker server has at least one function container in the sequence.

Figure 4.12 shows the 99th percentile end-to-end latency of the function sequence running on different numbers of worker servers. As the number of worker servers increases from 1 to 7, the performance of Meteion and OpenWhisk Composer stays stable, while the tail latencies of FaaSFlow and Nightcore increase by 4× and 24× respectively. For OpenWhisk Composer, the functions in a workflow always communicate with each other indirectly through the centralized workflow engine and message queues, so the end-to-end latency of the workflow is not affected by the number of worker servers.

Both FaaSFlow and Nightcore rely on function colocation to accelerate the workflow execution. FaaSFlow manages to reduce the control plane network and scheduling overheads by using a per-worker workflow engine to handle the logic of local functions, so that functions do not need to update their execution states in the master workflow engine. Although FaaSFlow can alleviate some of the control plane overheads, it still frequently falls back to the control

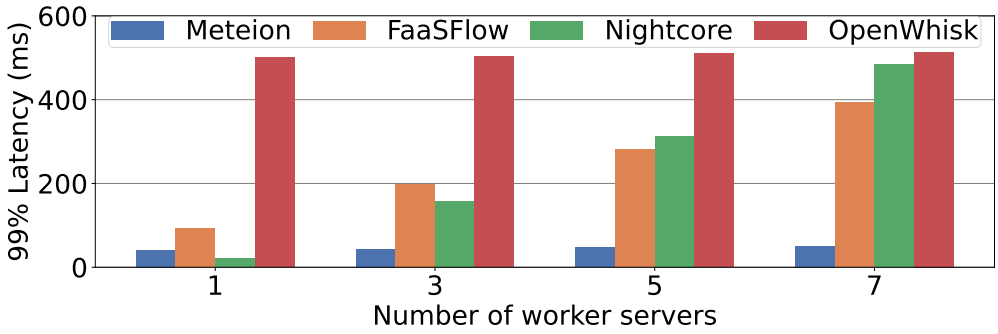


Figure 4.12: Performance of a function sequence with 10 stages running on multiple worker servers.

plane between function execution. When functions in a workflow are deployed on multiple worker servers, the per-worker workflow engines need to transfer and synchronize the state, resulting in a dramatic increase in tail latency. Nightcore also leverages function colocation, and uses local memory to enable fast direct inter-function communication. Nightcore is the best-performing approach when all functions in a workflow are colocated on a single worker server. However, it does not scale well in a distributed environment. When it is not possible to schedule function containers on the same node, Nightcore falls back to the API gateway for inter-function communication, which leads to higher control plane overheads and severe performance degradation.

Meteion is designed to be efficient in distributed environments and does not impose additional restrictions on container placement or rely on function colocation to reduce end-to-end latency. Meteion effectively separates the control plane from the workflow execution plane, and uses per-function workflow engines to enable direct inter-function communication across the cluster. Unlike the baseline approaches that suffer from performance degradation in distributed settings, Meteion remains efficient on multiple worker servers, and outperforms the baseline approaches with up to 10× reduction in tail latency.

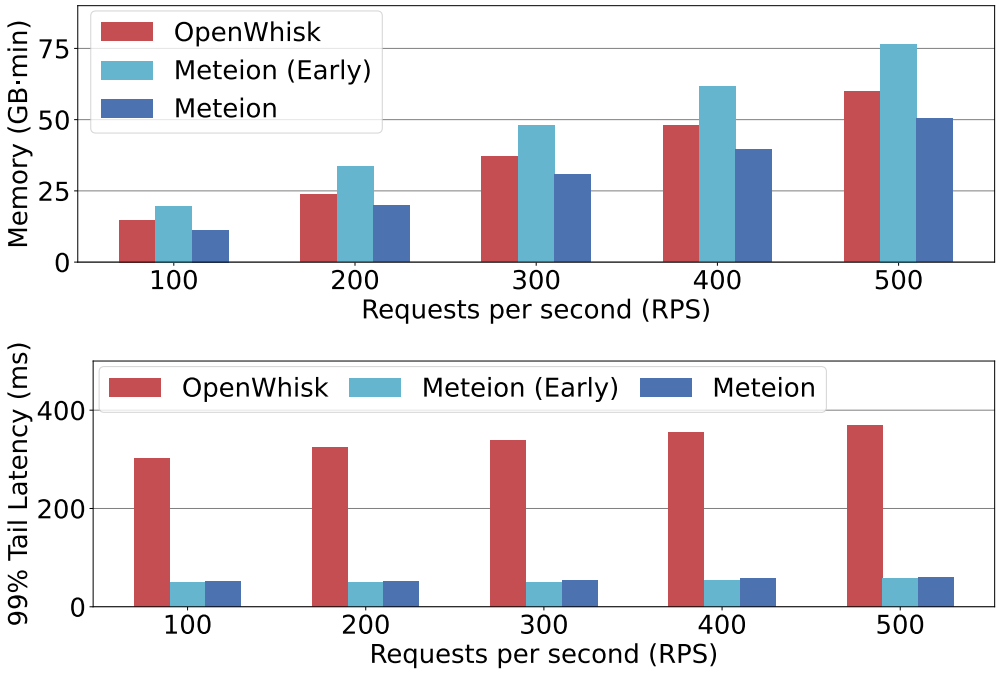


Figure 4.13: Aggregated provisioned memory time and tail latency with different scheduling policies.

4.5.3 Delay Workflow Scheduling

Performance of DAG scheduler. To avoid control plane overheads, Meteion's DAG scheduler needs to provision function containers in advance, and update the workflow manifests in the per-function workflow engines in a timely manner. We now show the performance of Meteion's DAG scheduler with the Social Network workflow, which consists of multiple stages with varied execution times. Figure 4.13 shows the aggregated provisioned memory time and 99th percentile latency of the workflow with different scheduling policies. OpenWhisk Composer provisions one container at a time at function invocation, while Meteion's early provisioning policy sets the provisioning delays of all stages in the workflow to zero, i.e., provisioning all containers upon workflow invocation. Early provisioning provides the full workflow manifest, and guar-

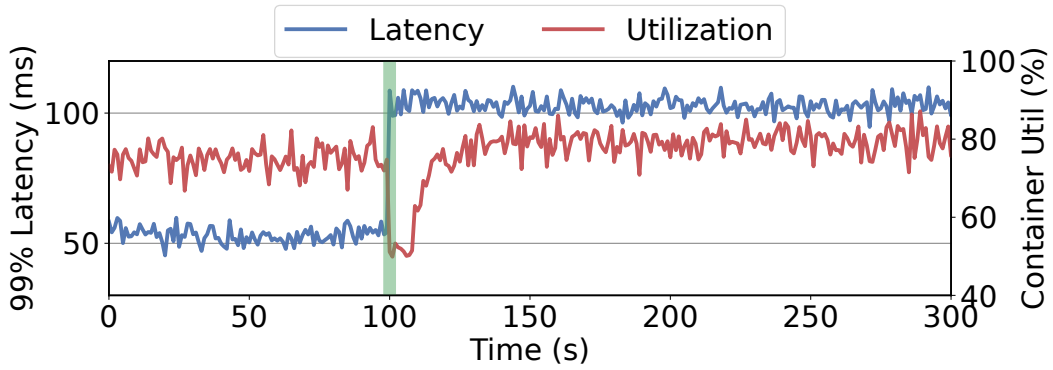
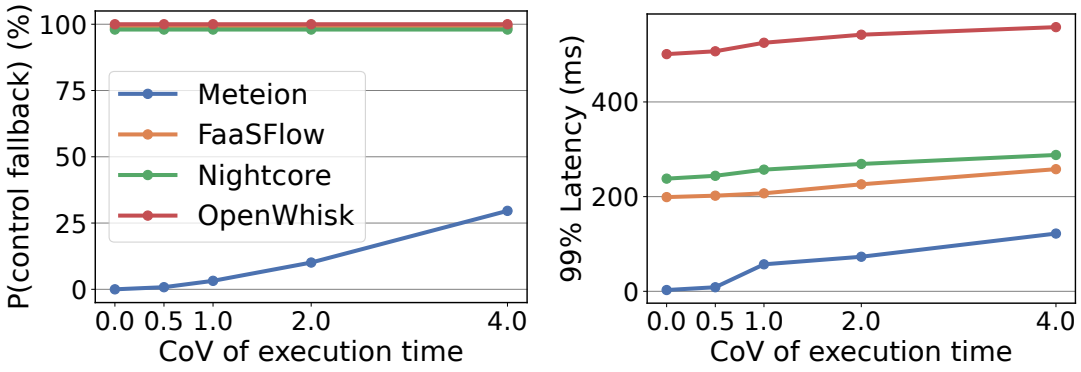


Figure 4.14: Meteion's DAG scheduler can adapt to distribution shifts and satisfies the resource utilization target.

antees in-situ execution across all stages, significantly reducing the end-to-end latency. This does, however, lead to some overprovisioning, with 27%–33% more provisioned memory time used in Meteion with early provisioning compared to OpenWhisk Composer.

By modeling the workflow latency as a distribution, and provisioning containers in each stage just in time, Meteion's delay workflow scheduling effectively reduces the overprovisioned memory time by 17%–24% compared to OpenWhisk Composer. Meteion determines the provisioning delay for each function based on the latency distribution, minimizing the probability of suffering from high control plane overheads while satisfying the resource utilization target. As a result, Meteion only yields a marginal increase (2%–3%) in tail latency compared to early provisioning. The DAG scheduler helps Meteion keep workflows executing within the worker servers without overprovisioning.

Distribution shift adaptation. Meteion's DAG scheduler can detect and adapt to distribution shifts in the workflow's performance behavior, which can be caused by changes in function inputs or function updates. As shown in Figure 4.14, the latency of the Social Network workflow increases as we enable



(a) Control plane fallback.

(b) Latency.

Figure 4.15: Probability of falling back to the control plane and end-to-end latency for workflows with varying degrees of execution time variability.

image compression in the compose-post function (marked by the green line), leading to early provisioning and lower container utilization. Meteion can detect the change in performance behavior and update the performance model accordingly by collecting new tracing logs using a sliding window approach, allowing it to satisfy the resource utilization target within 30 seconds.

Variation in workflow execution time. For workflows with highly variable execution times, Meteion's DAG scheduler prioritizes maintaining resource utilization targets to avoid early provisioning of downstream function containers based on long-tailed execution times of upstream functions. We use a synthetic workflow with different degrees of execution time variability to evaluate the performance of Meteion, as shown in Figure 4.15. We can see that a higher coefficient of variance in execution time can lead to increased control plane fallbacks. But under the same resource utilization constraint, Meteion's latency still outperforms alternative workflow engines that always fall back to the control plane.

4.5.4 End-to-End Evaluation

Finally, we perform an end-to-end analysis of Meteion by evaluating it on real-time data processing workflows and interactive web applications. We deploy each workflow to the cluster and evaluate its performance under different loads. Figure 4.16 shows the 99th percentile latency of each serverless workflow managed with different workflow engines. For the interactive web applications, we also show the performance of their microservice implementations to investigate any performance gaps between the two deployment strategies. Since Nightcore does not have a cluster-level container scheduling mechanism, to ensure a fair comparison, we use the Kubernetes scheduler (i.e., the same container scheduler used by Meteion) to schedule Nightcore's function containers across the cluster. It should be noted that Meteion does not impose any restrictions on the scheduling or placement of containers, so other container schedulers can be easily integrated into Meteion.

As shown in Figure 4.16, the OpenWhisk Composer suffers from high control plane overhead, since it relies on a centralized control plane workflow engine to handle all inter-function communication. FaaSFlow and Nightcore can mitigate the control plane overheads, and improve the performance of serverless workflows by leveraging function colocation. At low loads, FaaSFlow and Nightcore reduce the tail latency by 36%–56% compared to the OpenWhisk Composer. However, they still rely on centralized controllers to orchestrate the entire workflow execution process, resulting in the interleaving of the control plane and function containers, while the overhead of the control plane remains on the critical path. In contrast, microservices directly communicate with each other over RPCs based on application logic hard-coded in the service imple-

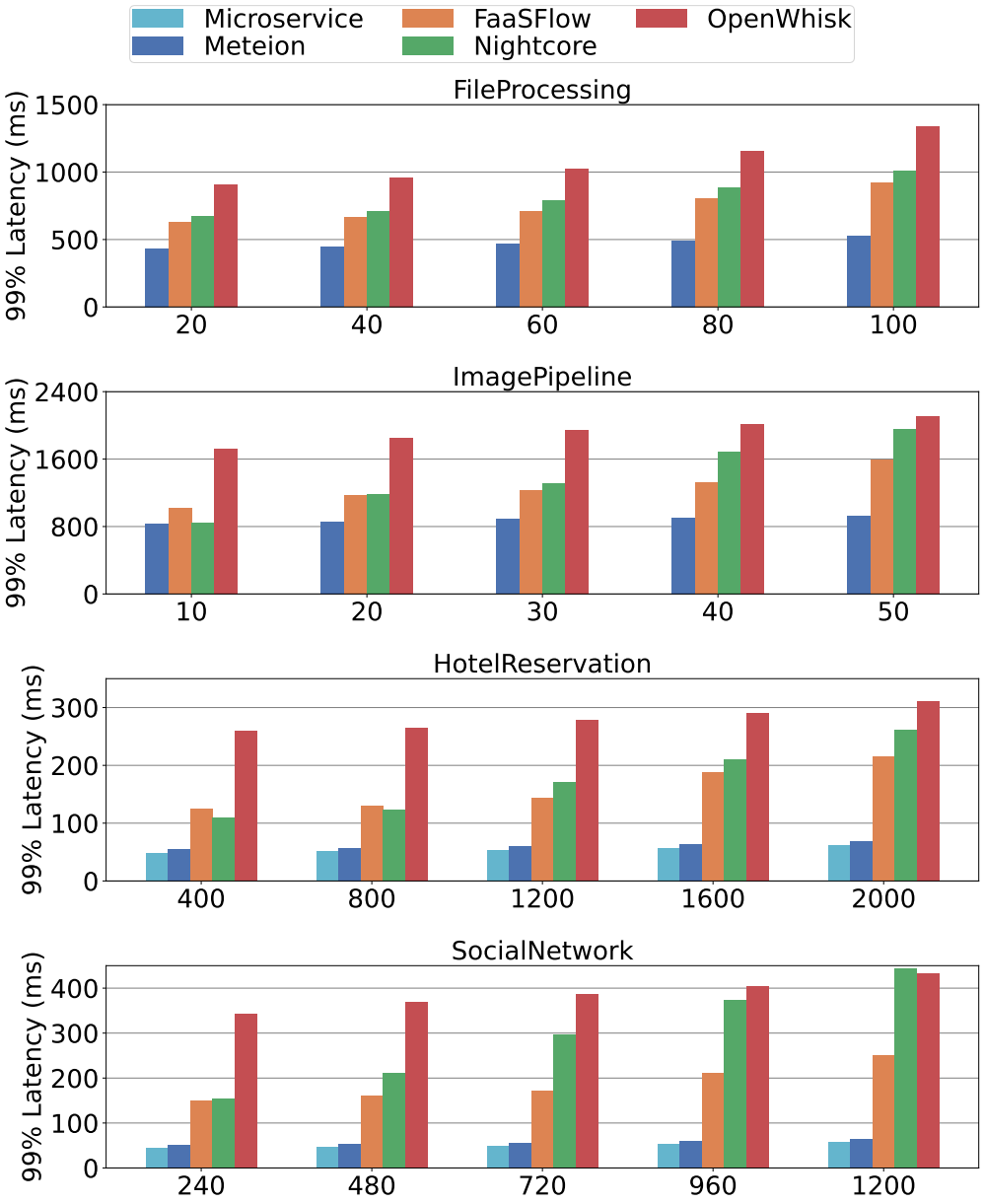


Figure 4.16: End-to-end performance of two real-time data processing workflows and two interactive web applications.

mentation itself. For interactive web applications, FaaSFlow and Nightcore experience a latency performance gap of up to 7× compared to the microservice implementation of the same application. Moreover, containers in workflows are more likely to be distributed across the cluster under high load, which can lead to increased control plane overheads and severe performance degradation of

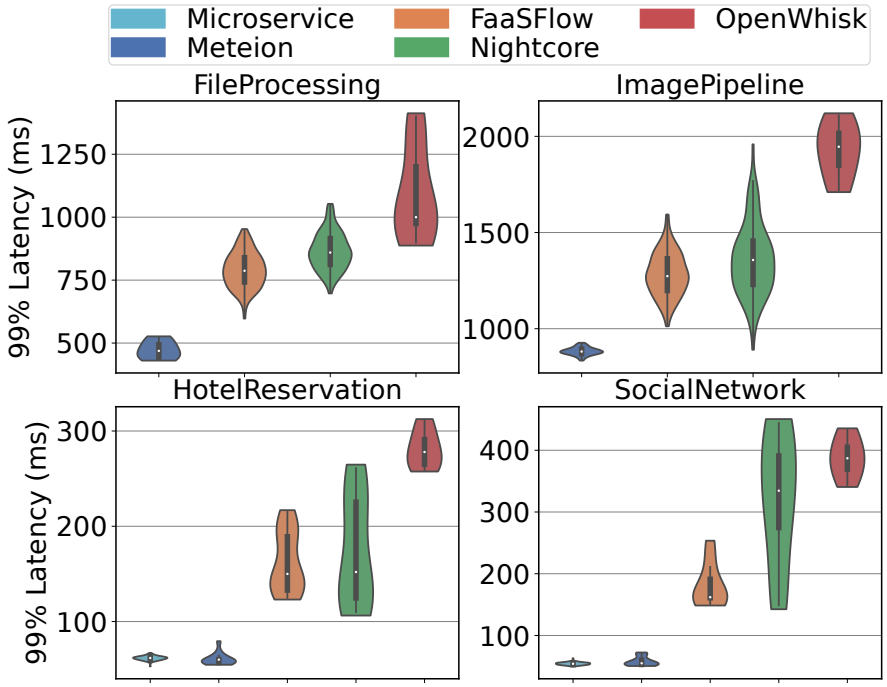


Figure 4.17: Latency distributions of serverless workflows with real-world workload traces.

existing colocation-based workflow engines.

Meteion consistently outperforms all baseline workflow engines across all the examined benchmarks, significantly reducing end-to-end latency. For the real-time data processing workflows, where functions have longer execution times, Meteion reduces tail latency by 37%–44% compared to the best-performing prior work. For the interactive web applications, where the control plane overhead dominates the end-to-end latency, Meteion further reduces tail latency by 64%–75% compared to the best-performing prior work, closing the latency performance gap between serverless workflows and microservices down to 5.1%–10.7%, and showing that serverless workflows do not need to be fundamentally slower than microservices, or unsuitable for low-latency applications.

We have also examined workflow engine performance with real-world workload traces from the Azure Function Dataset based on the methodology in [133]. The resulting tail latency distributions are shown in Figure 4.17. Meteion significantly improves the end-to-end latency, making serverless workflow suitable for interactive applications. Meteion also reduces the provisioned memory time by 9%–23% compared to the other workflow engines across all benchmarks. Meteion provides these performance benefits because it decouples the control plane from the workflow execution process, enables decentralized workflow orchestration, direct inter-function communication, fast in-situ workflow execution regardless of function placement, and provisions workflow containers in a timely manner.

4.6 Conclusion

We presented Meteion, a fast and efficient serverless workflow engine for latency-critical interactive applications. Met-eion decouples the control plane from the function execution, enabling fast in-situ workflow processing. Across several diverse real-world serverless applications, Meteion significantly reduces end-to-end latency, making serverless a high-performance solution for latency-critical cloud applications.

CHAPTER 5

CHARACTERIZING A MEMORY ALLOCATOR AT WAREHOUSE SCALE

5.1 Introduction

In addition to high-level system infrastructures, low-level software libraries also serve as foundational building blocks for datacenter applications. These software components, including serialization, remote procedure calls (RPCs), compression, hashing, compression, data movement, and memory allocation, are known as components of the *datacenter tax* [89,141]. Optimizing components of datacenter tax can significantly improve the efficiency and performance of the datacenter fleet, since entire classes of datacenter applications benefit from the improvements made.

In this chapter, we focus on improving the memory allocation and deallocation process, which constitute a substantial component of warehouse-scale computation [89]. We focus on memory allocator optimizations that maximize the productivity of datacenters by doing more useful work with the same or fewer hardware resources. Memory allocators directly affect the data locality of allocated objects and provide significant opportunities to optimize application performance. In comparison, optimizing the amount of time spent in the allocator itself is less important. Prior work profiles the CPU usage of memory allocators in datacenters [76,89,141] or measures the allocator performance using sets of benchmarks [36,50,65,78,99,104,106,121,150]. However, these studies solely focus on the time spent in the allocator or use benchmarks with limited memory allocation patterns, and thus provide only a narrow understanding of the performance characteristics and memory allocation behavior of datacenter

workloads. To fill this gap, we present the first comprehensive characterization study of TCMalloc [36], a memory allocator used by datacenter applications in Google's global datacenter fleet. We collect fleet-wide statistics from production workloads, and perform a detailed quantitative analysis of general memory allocator properties and memory allocation behaviors of datacenter workloads: the latency of allocation for different levels of allocator caches, the CPU cycles and memory fragmentation breakdown, and the distribution of allocated object sizes and their lifetimes. We also take an in-depth look at each component in the TCMalloc cache hierarchy, from the front-end per-CPU cache to the back-end pageheap [84], to identify performance bottlenecks resulting from the diverse allocation characteristics of warehouse-scale applications running on a fleet of heterogeneous servers.

Our characterization reveals profound diversity in memory allocation patterns, hardware platforms, as well as allocated object sizes and lifetimes for datacenter workloads. We observe that workloads are often co-located, and constrained to run on a subset of CPUs by the control plane. The dynamic input load causes the number of worker threads of a datacenter application to fluctuate constantly, resulting in significant variation in the cache miss ratio across the allocator's front-end caches. We show that the heterogeneity in our server fleet (e.g., differences in cache topologies of hardware platforms) can lead to varied data transfer overheads and increased cache pressure. We also observe that the distribution of object lifetimes varies across different sizes, which makes it challenging to make allocation packing decisions at different granularities (e.g., spans, hugepages) to reduce memory fragmentation and improve hugepage coverage [84, 108, 109, 157].

Based on our characterization, we derive unique insights and use them to design a memory allocator for datacenter applications. In particular, such an allocator needs to (1) adapt to the dynamic resource usage of datacenter applications, (2) be aware of heterogeneity in hardware platforms, and (3) utilize diverse lifetime information to make memory packing decisions. While our characterization study centers on TCMalloc, most modern memory allocators (e.g., jemalloc [65], mimalloc [99]) share a similar hierarchical system architecture and cache memory allocations in multiple tiers, making these insights universal to memory allocators used in datacenters. Based on these insights, we redesign and tune each component in the TCMalloc cache hierarchy for datacenter environments, including enabling usage-based dynamic sizing of per-CPU caches, leveraging the hardware topology to mitigate the inter-core communication overhead in the transfer cache, and improving the allocation packing algorithms based on statistical data in the central free list and the pageheap. We evaluate these design choices with fleet-wide A/B experiments and longitudinal rollout in our production datacenters. Evaluation results show that by redesigning the memory allocator for warehouse-scale environments, we achieve a significant improvement in fleet productivity.

This chapter makes the following main contributions:

- The first comprehensive characterization study of TCMalloc, a memory allocator used in warehouse-scale environments. Based on profiling data collected from production datacenter workloads, we characterize the general memory allocator properties, and delve into each tier of the TCMalloc cache hierarchy.
- Based on our characterization, we uncover insights into designing a mem-

ory allocator for datacenter applications, including adapting to dynamic application resource usage, being aware of server heterogeneity, and leveraging object lifetime information to improve allocation placement decisions.

- We redesign and tune each component in the TCMalloc cache hierarchy for warehouse-scale environments and evaluate their performance impact through fleet-wide experiments, resulting in a 1.4% throughput improvement and a 3.4% memory reduction across the fleet. For the applications with the highest `malloc` usage, we observe up to 8.1% and 6.3% improvement in throughput and memory usage respectively. At our scale, a single percent CPU or memory improvement translates to significant resource savings.

5.2 Background and Methodology

TCMalloc [36] is a memory allocator used in warehouse-scale environments. It is a fast, multi-threaded `malloc` implementation that has shown robust performance in large-scale production services [84, 90, 108, 109].

5.2.1 TCMalloc System Architecture

Figure 5.1 shows the hierarchical system architecture of TCMalloc. In TCMalloc, allocations of small objects (i.e., ≤ 256 KB) are rounded up to one of 80–90 size classes. Objects of each size class are cached by multiple *per-CPU caches* (①), a *transfer cache* (②), and a *central free list* (③). Large objects that exceed the

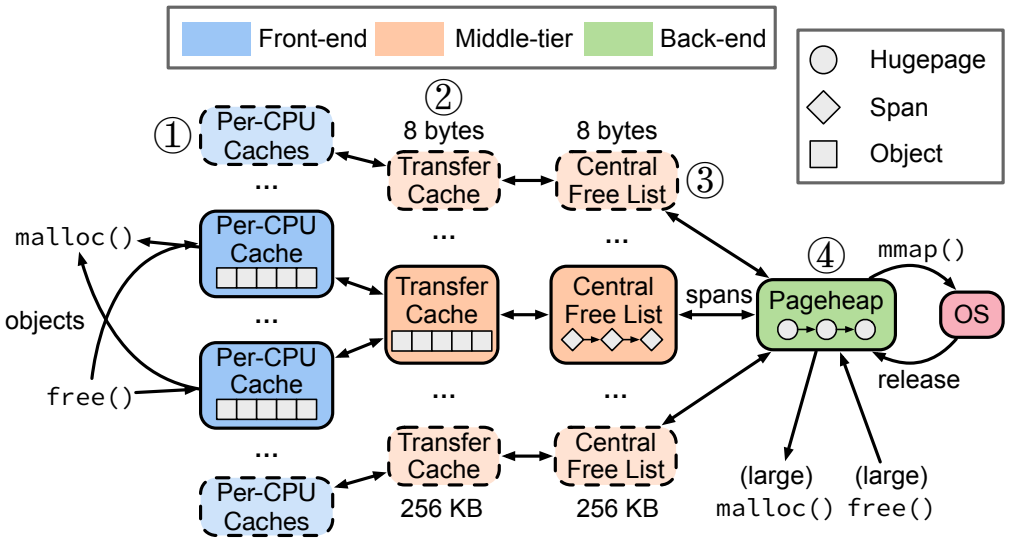


Figure 5.1: System architecture of TCMalloc. It has a tiered cache structure that aids fast allocations and deallocations.

threshold are directly allocated from the *pageheap* (④), without being cached by the front-end or middle-tier caches.

The front-end contains per-CPU caches (①) that provide fast memory allocation and deallocation for the application. The per-CPU caches store *objects* in a large contiguous block of memory that is divided between CPUs, and each CPU uses a portion of the block to store metadata and pointers to the available objects. Each per-CPU cache can only be accessed by a single thread at a time. Therefore, no locks are required and most operations are fast. When the front-end is empty, it requests objects from the middle tier to refill the cache.

The middle tier consists of small, fast, mutex-protected transfer caches, and large, mutex-protected central free lists. The transfer cache (②) stores objects in flat arrays. It allows memory to rapidly flow between different CPUs (e.g., CPU 0 may allocate memory that is deallocated by CPU 1). If the transfer cache is unable to satisfy the memory request, or has insufficient space to hold the returned objects, it reaches out to the central free list. The central free list (③) manages

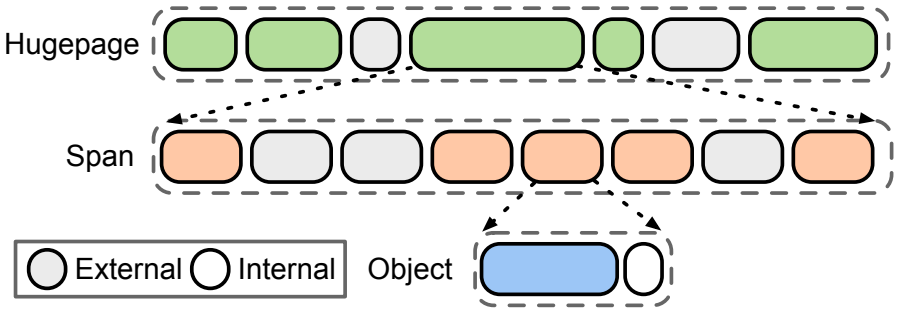


Figure 5.2: Memory organization layout and fragmentation in TCMalloc. Hugepages are divided into spans of various sizes, and spans are sub-divided into objects of fixed size classes.

spans in linked lists, and fulfills allocation requests by extracting objects from the spans. A span is a collection of contiguous fixed-size regions, aligned to an 8 KB TCMalloc *page* [1]. As shown in Figure 5.2, a span contains multiple objects of the same size class. If there are insufficient available objects in the spans, more spans are requested from the back-end.

The back-end pageheap (④) manages memory in units of hugepages [84]. The pageheap requests hugepage-aligned memory blocks from the system, which provides an opportunity for the kernel to use hugepages to cover consecutive pages in the page table [37]. A hugepage is divided into TCMalloc pages, and the pageheap extracts spans from hugepages to refill the central free list. The pageheap also periodically releases memory to the OS, either by releasing hugepages that are completely free, or by breaking partially-filled hugepages into smaller pages and subreleasing them [84, 109].

Memory organization. Storing and managing memory at different granularities can lead to *external* and *internal fragmentation* in TCMalloc, as shown in Figure 5.2. External fragmentation refers to the memory that is cached by the

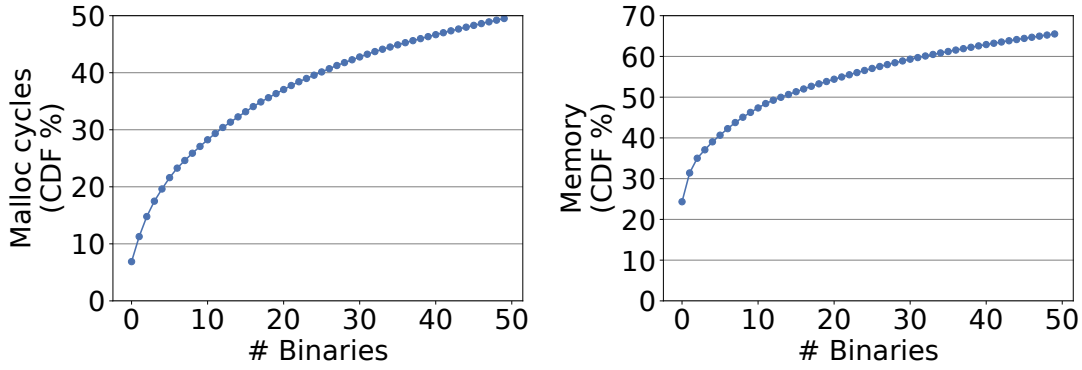
[1]TCMalloc page size should not be confused with the system page size – the default 8 KB TCMalloc page is composed of two native x86 memory pages.

allocator, but is yet to be allocated by the application. For instance, a hugepage may contain unallocated memory regions in the spans, while a span may contain unallocated objects. In contrast, internal fragmentation in TCMalloc results from rounding allocation requests to discrete size classes. As such, it is slack between the object size requested by the application and the size class allocated by the allocator. It is critical to manage both internal and the external fragmentation to minimize memory overheads in TCMalloc.

5.2.2 Characterization Methodology

We put in place telemetry for collecting fleet-wide statistics from production workloads. We use these statistics to perform a detailed characterization of TCMalloc.

Application productivity. Prior characterization studies focus on "datacenter tax" [76, 89, 140, 141] for common libraries in datacenter applications and propose several acceleration opportunities to lower their CPU overhead. In this work, we argue that optimizing for the CPU overhead of these libraries is less important. Instead, it is more crucial to focus on improving the efficiency of these common libraries to improve fleet *productivity*, i.e., fulfilling more requests with the same or fewer hardware resources. Prior work [89,141] analyzes processor pipeline stalls in datacenter applications, attributing 20-64% stalls to back-end, primarily due to cache misses. Memory allocators directly impact the data locality of allocated objects, and thus, present significant opportunity to optimize application performance bottlenecks at scale. To this end, we focus on improving application productivity and showing how improvements to



(a) CDF of malloc cycles.

(b) CDF of allocated memory.

Figure 5.3: `malloc` cycle and allocated memory distribution in our fleet. The top 50 binaries in datacenters only cover ≈ 50% malloc cycles and ≈ 65% allocated memory.

cache and dTLB locality impact datacenter productivity, without directly targeting improvements to the `malloc` CPU overhead. Our fleet productivity metrics consist of per-application-defined custom throughput metrics (e.g., RPCs processed per second) that define performance for the respective applications.

Continuous profiling. We collect performance metrics and memory allocator telemetry from the production fleet using Google-Wide Profiling [125] (GWP), an unobtrusive profiling framework with negligible overhead. GWP randomly selects a small fraction (i.e., 1%–10%) of machines in the fleet to profile each day, and triggers profile collection remotely on each machine for a brief period of time. Continuous profiling allows us to study the memory allocation behavior of applications across the fleet at different levels of granularity and time intervals.

Fleet experiment. The diversity of datacenter applications implies that there is no single killer application to optimize for. Figure 5.3 shows the fleet-wide cumulative distribution of `malloc` cycles and allocated memory, where the top 50

binaries account for over 50% `malloc` cycles and 65% of allocated memory. To measure the impact of optimizations on fleet productivity, we use an experimentation framework to A/B test implementations across our fleet at scale. For each design, the framework randomly selects 1% of the machines in the fleet as an experiment group and a separate 1% as a control group. We apply the change to all the binaries running in the experiment group and compare their performance with the control group. This lets us evaluate design choices on diverse production applications with realistic loads.

Generalizability. Our characterization study focuses on TCMalloc, and the design choices are closely related to TCMalloc's internal implementation. However, most modern memory allocators (e.g., jemalloc [65], mimalloc [99]) share similar hierarchical architecture and cache memory allocations in multiple tiers, which makes the insights derived from our characterization universal to memory allocators in warehouse-scale environments. For example, jemalloc has a tiered cache structure consisting of thread caches and arenas, in which it organizes memory in regions, runs and extents [65]. These memory allocators can also benefit from a comprehensive characterization of TCMalloc, and adopt similar optimizations to improve their performance at scale.

5.2.3 Production Workloads and Benchmarks

In addition to the fleet-wide metrics, we also use five production workloads in our fleet with the highest `malloc` usage for characterization and evaluation.

- **Spanner** [56] is a node in a distributed SQL database. It includes an in-memory cache of storage data, which adapts to the memory provisioned

for the process.

- **Monarch** [41] is part of a scalable monitoring system that collects and stores time-series metrics for production services. It is responsible for holding stream data in memory, and participating in query evaluation.
- **Bigtable** [53] is a tablet server that hosts and serves the user data of a large-scale key-value NoSQL database. It also implements replication and coordinates compactions with external compactors.
- **F1 query** [131] is a high-performance distributed query engine. It uses RPCs to communicate with clients and data sources.
- **Disk** is a low-level distributed storage system that provides RPC access to read and write files directly to a machine's local hard disk or flash memory.

We also run benchmarks on a dedicated server to demonstrate the performance impact of several optimizations.

- **Redis** [30] is an open-source in-memory key-value store (v7.0.8). We use the standard redis-benchmark, configured with 500 concurrent connections and 100K operations of 1000B as the workload generator.
- **Data processing pipeline** is a data processing workload running word count on a 1 GB file with 100M words. We run the entire computation as a single process, which creates pressure on memory allocation.
- **Image processing server** is a production server that filters and transforms images. We use a synthetic workload generator to create concurrent client requests.

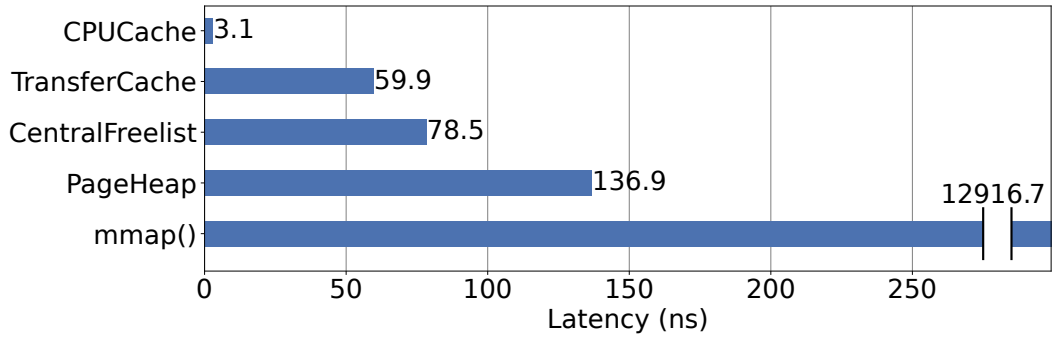


Figure 5.4: Disparity in allocation latency of hitting different tiers in the TCMalloc cache hierarchy.

- **Tensorflow** is the open-source Serving [115] framework that runs the InceptionV3 [144] image recognition model. It uses libraries (e.g. Eigen linear algebra library [11]) with complex memory allocation behavior.

5.3 General Characterization

We first conduct a general characterization to gain insights into how the memory allocator behaves in our production fleet and how its characteristics affect system performance.

Allocation latency. We use microbenchmarks to measure the mean allocation latency for hitting different tiers of caches. As shown in Figure 5.4, allocations fulfilled by the per-CPU cache have the lowest latency, since it stores objects in a contiguous block of memory, and uses a highly optimized fast path supported by restartable sequences [31, 32] to handle allocation requests. The fast-path that hits the per-CPU caches consists of ~40 hand-coded x86 instructions, with an allocation latency of 3.1 ns. Hitting the transfer cache and the central free list indicates that the front-end cache is empty, and needs to be refilled with a batch of objects. Both the transfer cache and the central free list

are protected by mutex locks, which denotes an additional cost to access them. The central free list needs to extract objects from spans organized in linked lists, resulting in increased latency.

If both the front-end and the middle-tier caches are empty, the allocation request hits the pageheap, which holds memory in units of hugepages [84]. The pageheap tracks allocations in hugepages and results in the longest latency of over 137 ns in the cache hierarchy. We also include the latency of the `mmap` system call used in TCMalloc measured with strace [35]: when the allocation request misses all the front-end and middle-end caches, and the back-end pageheap is also empty, TCMalloc requests a zero-initialized 2MB hugepage from the system using `mmap` to refill the pageheap. The latency of refilling the pageheap is orders of magnitude higher than the latency of hitting the caches, highlighting the need for caching in a userspace allocator.

Malloc CPU cycles. Memory allocation and deallocation make up a substantial component of warehouse-scale computation. Figure 5.5a shows the relative amount of CPU cycles spent in allocation and deallocation functions over a two-week period, where the `malloc` overhead accounts for 4.3% fleet CPU cycles. For the top 5 applications with the highest `malloc` cycles in the fleet, the `malloc` overhead varies between 3.6%–10.1%. While understanding the `malloc` CPU overhead itself helps prioritize optimization opportunities, as we explained earlier in Section 5.2.2, we primarily aim to improve overall fleet productivity at scale.

We also include data collected from SPEC CPU2006 [80] benchmarks for comparison. Most of the SPEC benchmarks do not actively allocate or deallocate objects in stable state and have near-zero `malloc` cycles, which makes

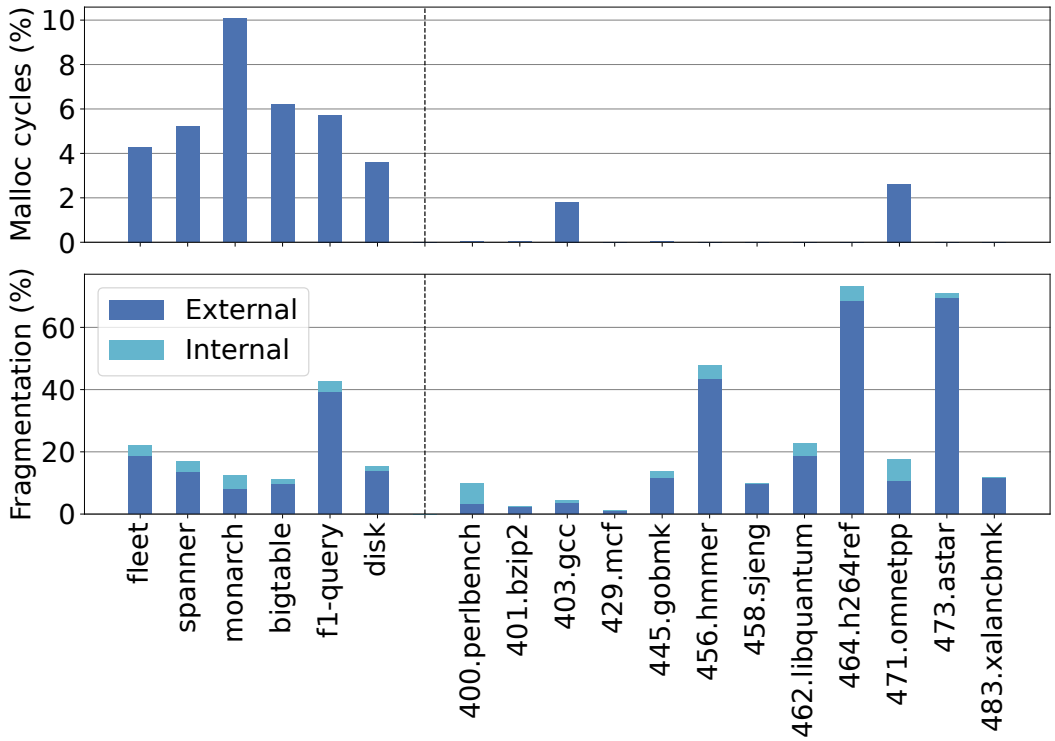


Figure 5.5: (a) Relative amount of time (% of cycles) spent in memory allocation, and (b) memory fragmentation ratio for the fleet and top 5 production workloads in two weeks. We also include SPEC CPU2006 benchmarks for comparison.

them unsuitable for studying memory allocation behavior.

CPU cycles breakdown. We classify the profiled call stack traces into several categories to further understand the breakdown of the CPU cycles used by the memory allocator. As shown in Figure 5.6a, TCMalloc spends most of its time (i.e., 53% of fleet-wide `malloc` cycles) in the per-CPU cache, since most of the requests hit the front-end cache. Allocation requests fulfilled by the per-CPU caches have the lowest latency (as described earlier in Figure 5.4), so we expect TCMalloc to spend most of its CPU cycles serving requests from front-end caches. Another 3% of the `malloc` cycles are spent in the transfer cache. The central free list accounts for 12% of the fleet `malloc` cycles, since it employs a linked-list structure to manage spans that incur higher cost to allocate objects

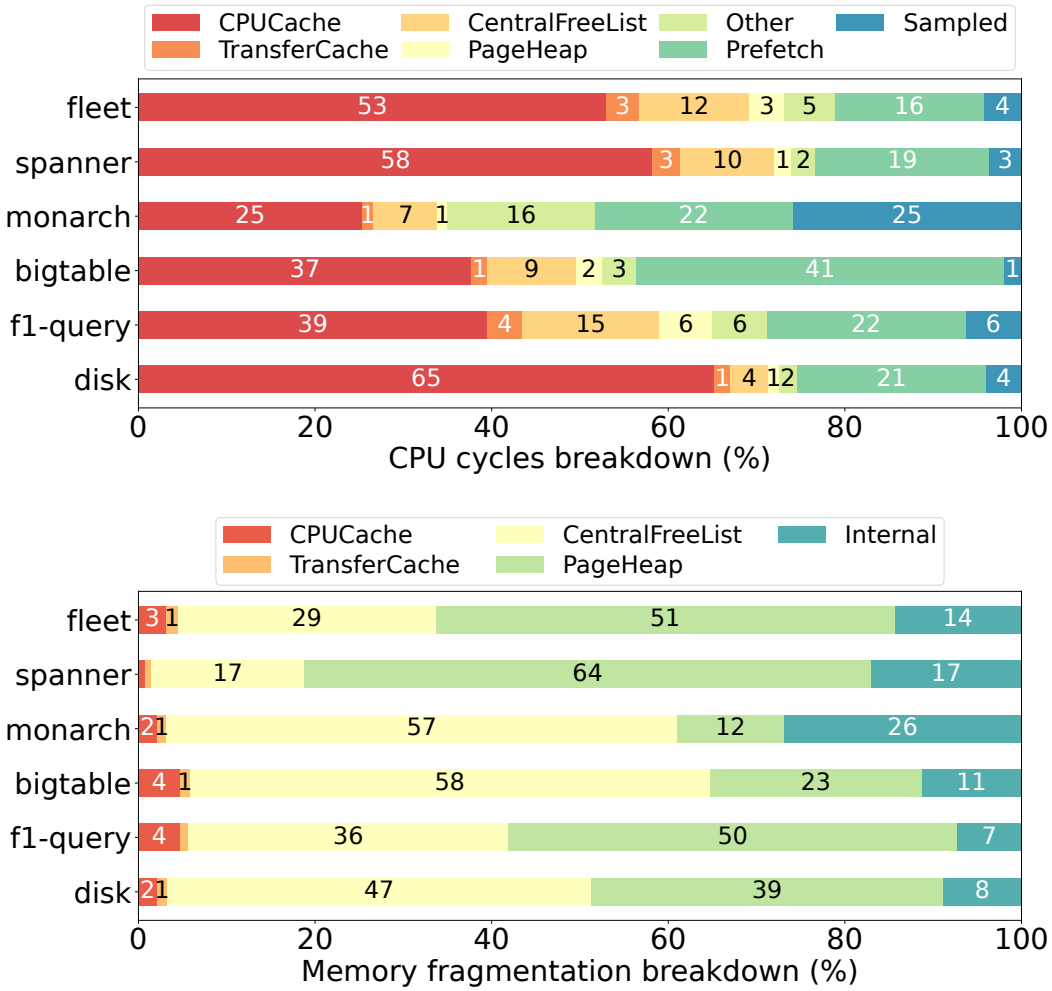


Figure 5.6: (a) Breakdown of CPU cycles consumed by TCMalloc. (b) Memory fragmentation breakdown of TCMalloc.

from and deallocate objects to. Finally, TCMalloc spends 3% of its CPU cycles in the pageheap.

In the production setting, TCMalloc samples an allocation request for every 2 MB of memory allocations. *Sampled* accounts for time spent in sampled allocations, where the allocator additionally records the current call stack trace. Sampling accounts for 4% of `malloc` cycles, but in a production environment, it is invaluable for analyzing memory usage and debugging memory leaks. Some fleet applications (e.g., `monitor`) employ extensive sampling, so `Sampled` ac-

counts for higher proportion of CPU cycles. *Other* refers to CPU cycles that were not classified into a specific category (e.g., due to allocations that require complex logic).

For each allocation request, TCMalloc prefetches the next object of the same size class that would be returned. It is too late to prefetch the current object when it is returned [98]: the user code can start using the object within a few cycles, well before prefetching from the main memory can complete. Prefetching gives time for the next object to be loaded into the cache before the next allocation request. *Prefetch* appears to be costly, taking 16% of `malloc` cycles in the fleet, but is key in reducing data cache misses.

Memory fragmentation. The *fragmentation ratio* is the ratio of the fragmented memory to the live in-use memory by the application. Figure 5.5b shows the average memory fragmentation ratio for the fleet and the top 5 applications. The fleet-wide fragmentation ratio is 22.2% of the total application heap size, which consists of 18.8% external fragmentation (i.e., unused memory cached by the allocator) and 3.4% internal fragmentation (i.e., the slack between the allocated size class and the requested object size). For the top 5 applications with the highest allocator usage, the fragmentation ratio ranges from 11.2% to 42.5% of the respective heap size.

Memory fragmentation breakdown. Figure 5.6b decomposes external fragmentation into fragmentation within each component of the TCMalloc cache hierarchy. The major sources of fragmentation are the central free list and the pageheap, which account for 29% and 51% of the total fragmentation respectively. A span in the central free list can be returned to the pageheap only when all objects are returned to it. A single long-lived object on a span may

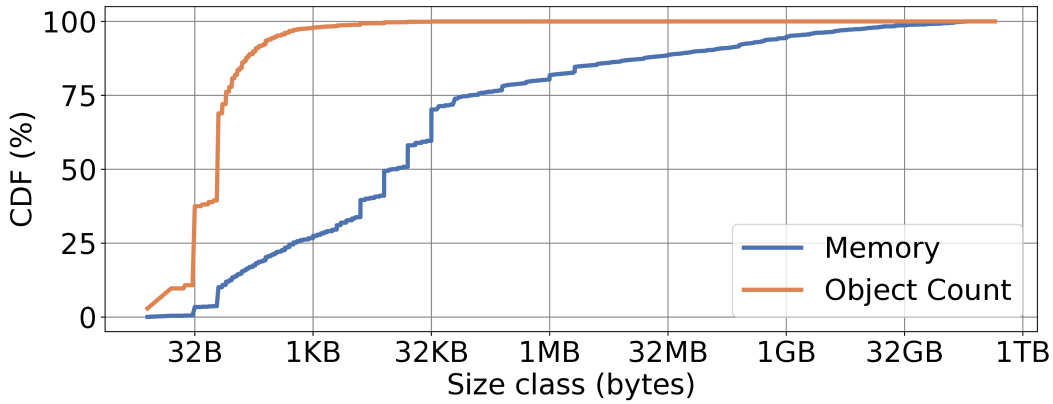


Figure 5.7: CDF of allocated objects in datacenter applications.

disallow the central free list to return that span, leading to substantial memory fragmentation. The pageheap manages free spans in hugepages, and accounts for the majority of memory fragmentation in TCMalloc. It can wait for an entire hugepage to become free before releasing the memory back to the OS, or subrelease a hugepage [109] by breaking it into non-hugepage-aligned memory regions. The former preserves hugepage coverage but leaves memory idle, while the latter leads to performance degradation due to decreased hugepage coverage. TCMalloc prioritizes keeping hugepages intact [84, 109] by releasing memory gradually from the pageheap.

Internal fragmentation accounts for 15% of the fleet fragmentation, which results from the slack between the requested object sizes and the next available size class. TCMalloc may use finer-grained size classes to reduce the gap between requested memory and allocated size classes, but this also prevents reuse of memory blocks in the hierarchy. With finer-grained size classes, TCMalloc needs to manage additional per-size-class free lists in its front-end and middle-tier caches, increasing external fragmentation and reducing object reuse. Through its size class selection, TCMalloc strikes a balance between the internal fragmentation due to the number of size classes and the external fragmentation

from unused memory blocks in its cache hierarchy.

Distribution of allocated objects. To study the distribution of allocated objects in production, we sample the memory allocations in the fleet over a two-week period. Figure 5.7 shows the cumulative distribution of the number of allocated objects and memory in the fleet, as a function of object size. Objects smaller than 1 KB make up 98% of the allocated objects, but occupy only 28% of the memory. However, when we focus on the total allocated memory size per size class, objects larger than 8 KB, account for 50% of the fleet memory. The largest size class in TCMalloc is 256 KB. Objects that exceed this threshold bypass TCMalloc's cache hierarchy and are directly allocated from the pageheap. They account for 22% of the allocated memory. To avoid excessive fragmentation due to these objects, the pageheap maps these allocations in a separate set of a continuous run of hugepages, as we discuss in Section 5.4.1. The distribution of allocated objects shows that small objects occupy only a fraction of the memory but dominate the total number of allocated objects. Therefore, the TCMalloc caches prefer optimizing available capacity towards smaller size classes to reduce overall allocation latency.

Distribution of object lifetime. Figure 5.8 shows the distribution of object lifetime, based on object size and weighted by the number of sampled allocations. We collect object lifetime profiles from servers with uptime of at least a week. We observe that objects have diverse lifetimes. For objects smaller than 16 MB, they can be long-lived (i.e., $\geq$ 7 days) or short-lived (i.e., $\leq$ 1 millisecond), or somewhere in between. Lifetimes vary greatly even for objects within the same size range. In general, smaller objects ($\leq$ 1 KB) are heavily allocated by our applications (as shown earlier in Figure 5.7), and also have shorter lifetimes,

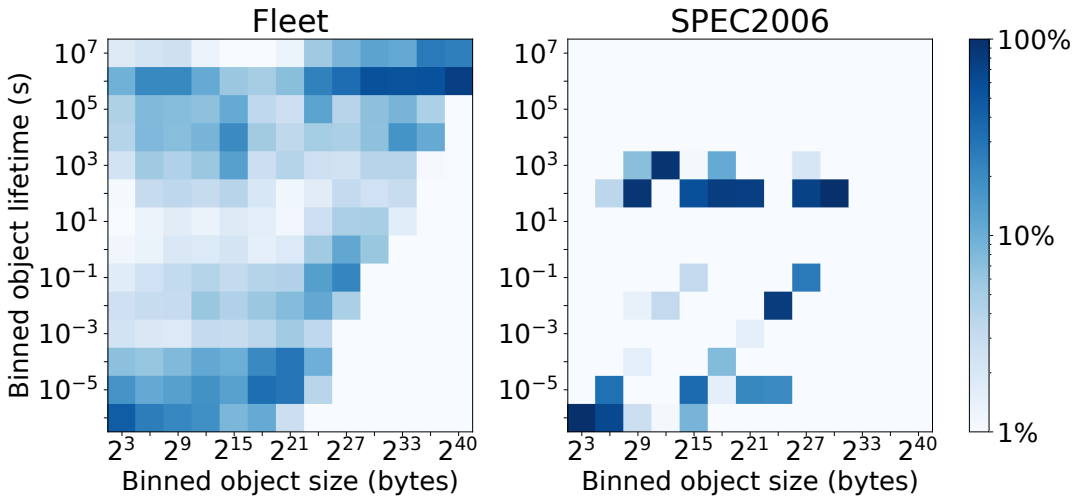


Figure 5.8: Distribution of fleet-wide object lifetime, based on object size and weighted by the number of sampled allocations. We also include the object lifetime distribution of SPEC CPU2006 benchmarks.

with 46% of objects living shorter than 1 millisecond. We also observe that large objects are likely to have longer lifetimes, where 65% of objects that are larger than 1 GB live longer than 1 day. This diversity in object lifetimes provides an interesting opportunity for the allocator to place objects with similar lifetimes together in the cache tiers (e.g., to reduce external fragmentation).

In Figure 5.8, we also include the object lifetime distribution sampled from SPEC CPU2006 [80] benchmarks. We run each benchmark to completion and combine the lifetime profiles together. The object lifetime distribution in SPEC benchmarks is much less diverse than what we observe in the fleet. These benchmarks do not actively allocate or deallocate objects in their stable state. Most objects are either alive as long as the program lives or only live for a short period of time (i.e., ≤ 1 ms). This again makes SPEC benchmarks unsuitable for evaluating the performance of memory allocators.

5.4 Characterizing and Redesigning Caches

Next, we take an in-depth look at each tier in the TCMalloc cache hierarchy to uncover performance insights and potential optimization opportunities. For each tier, we perform a performance characterization, derive performance insights, propose new designs, and evaluate their performance impact.

5.4.1 Per-CPU Cache

The per-CPU cache in TCMalloc is the front-end cache that provides fast allocation and deallocation of memory to the application. A per-CPU cache [2] is shared by the software threads scheduled to run on a given CPU core, which allows for more efficient use of the cache. The per-CPU cache contains a single large block of memory that is divided between CPUs. Each CPU is assigned a section of this memory to hold metadata and pointers to the available objects of a particular size class. The block size provides a bound on the capacity that TCMalloc may cache in the front-end caches.

Virtual CPUs. The per-CPU caches are populated for all the CPUs on which the application runs. datacenter applications are often co-located on the same server, and are constrained to run on a subset of CPUs by the control plane scheduler [10, 54, 107]. For applications that use only a part of the machine, the available CPU range is excessive. In our fleet, we have observed a 4× increase in the number of hyperthreads per server system over the last five platform

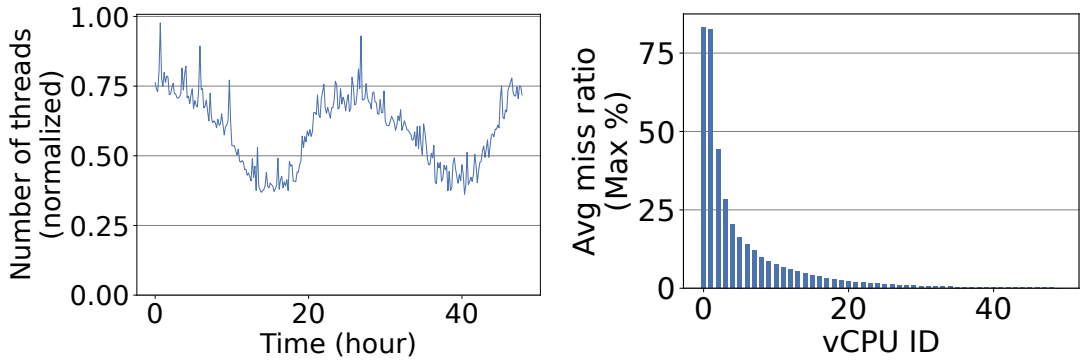
[2] TCMalloc now uses the per-CPU cache as its front-end cache by default, which is a major improvement over earlier versions that used per-thread caches. Being inaccessible to other application threads, per-thread caches strand memory when the threads become idle. The scalability becomes worse in applications with thousands of threads. Unfortunately, this also makes TCMalloc, *a thread-caching malloc*, a misnomer.

generations. As the number of hyperthreads increases, the per-CPU caches and the associated metadata may grow substantially between platform generations, even though the populated caches are not effectively used by the applications.

To improve scalability across platform generations, TCMalloc makes use of virtual CPU (vCPU) IDs [55], which are managed by the kernel in process-private number space. The vCPU IDs are assigned to prevent TCMalloc from initializing and maintaining per-CPU data structures for all CPU IDs available on the platform. By using a dense set of vCPU IDs to index the per-CPU cache, TCMalloc significantly reduces the number of unique CPU caches that need to be populated, and avoids populating caches on all *accessible* cores. For example, if an application runs on two CPU cores, virtual CPUs always expose IDs `0` and `1`, irrespective of which physical cores the application threads are scheduled on.

Disparity in cache usage. By default, each per-CPU cache is statically sized to store up to 3 MB of objects. While the vCPUs reduce the number of used caches, we observe that they also bias cache usage towards the lower-indexed per-CPU caches. Figure 5.9a shows the number of worker threads of a middle-tier service in our search service stack. We can see that the number of worker threads constantly fluctuates, due to load spikes and diurnal usage. As such, datacenter applications typically handle dynamic loads by varying the number of CPU cores they use. A sudden burst of load may populate caches for the higher-indexed vCPUs, but the usage of these caches may subside as the load decreases.

We notice the disparity in cache usage based on the cache miss ratio. We collect the number of misses, i.e., the number of allocation and deallocation misses due to insufficient cache capacity, for all per-CPU caches with differ-



(a) Threads count.

(b) Average miss ratio.

Figure 5.9: (a) The dynamic nature of datacenter workloads. The number of active threads constantly fluctuates. (b) Significant variation in miss ratio of per-CPU cache for different vCPU IDs. Higher-indexed caches are inefficiently used.

ent vCPU IDs over a two-week period. Deallocation misses occur when the application frees an object, and the corresponding front-end cache is full and does not have sufficient capacity for the returned object. In such cases, the request spills over to the transfer cache. Figure 5.9b shows the average ratio of the number of misses encountered by each per-CPU cache to the total number of misses over all per-CPU caches. We observe that vCPU 0 suffers the highest number of misses, and the miss ratio is substantially lower for higher-indexed vCPU IDs. This clearly demonstrates that the higher-indexed per-CPU caches are infrequently used. As each per-CPU cache is only allowed to cache up to 3 MB of objects, the higher-indexed per-CPU caches use this capacity much more inefficiently than the lower-indexed per-CPU caches. This disparity suggests the need for a heterogeneous cache design.

Heterogeneous per-CPU cache. In contrast to statically-sized per-CPU caches that are inelastic to changing application behavior, we propose heterogeneous per-CPU caches that can be dynamically sized to balance the misses across all the populated caches. With dynamic resizing, we expect the lower-

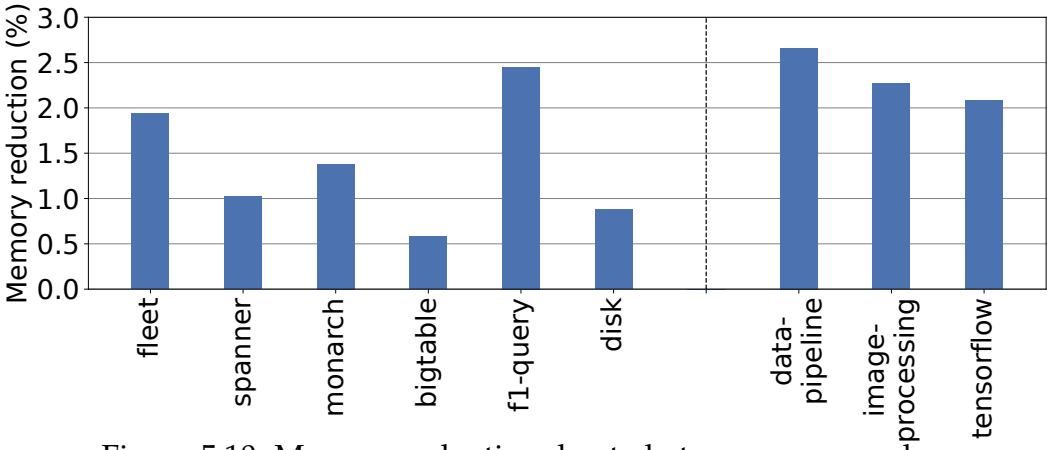


Figure 5.10: Memory reduction due to heterogeneous caches.

indexed per-CPU caches to have a larger capacity compared to the higher-indexed per-CPU caches in. Measuring the absolute number of requests served by each per-CPU cache can slow down the fast path. Instead, we record the total number of misses encountered by each per-CPU cache every 5 seconds, and use it as a proxy for cache utilization.

To balance the cache utilization, we employ a background thread that periodically resizes and re-allocates the capacity from lower-utilized to higher-utilized caches. During each resize interval, we identify the top five per-CPU caches with highest misses during the previous 5-second interval as the candidates we may want to grow. We then iterate through the remaining per-CPU caches in a round-robin fashion to identify the candidate per-CPU caches to steal capacity from. For each per-CPU cache we aim to shrink, we prioritize shrinking capacity for larger size classes, since the majority of allocations in our workloads are smaller objects (see Fig. 5.7).

Evaluation. As we described in Section 5.3, the external fragmentation overhead in TCMalloc accounts for 22.2% of the total application heap size in our fleet. Through our heterogeneous per-CPU cache design, we aim to improve

this fragmentation overhead. Because the dynamic scheme improves the utilization of front-end caches, we simultaneously reduce the default size of each per-CPU cache from 3 MB to 1.5 MB. Note that, due to the reduced capacity of the front-end caches, we also observe a reduction in fragmentation in the transfer cache, central free list and pageheap, as TCMalloc ends up caching fewer objects in aggregate. Our fleet experiments reveal that lowering the capacity results in no performance impact for our applications. As shown in Figure 5.10, we observe a 1.94% reduction in fleet memory usage, and a 0.58% – 2.45% reduction in memory usage of the top 5 applications. For the benchmarks in Section 5.2.3, the memory usage of the data processing pipeline, image processing server and Tensorflow serving reduces by 2.66%, 2.27%, and 2.08%, respectively. We omit Redis because it is single-threaded, hence it uses a single per-CPU cache.

5.4.2 Transfer Cache

The transfer cache holds an array of pointers to free objects. When the per-CPU cache is depleted or full, it reaches out to the transfer cache to request or return a batch of objects. The transfer cache provides a centralized repository of objects that is shared by all the per-CPU caches. That is, an object de-allocated by one per-CPU cache may be later allocated by another per-CPU cache. As such, the transfer cache allows memory objects to flow rapidly between the per-CPU caches.

Datacenter heterogeneity. To achieve the desired performance, a memory allocator needs to adapt to the heterogeneity in hardware platforms. In recent years, Moore's law has slowed down [63, 64] and the cost scaling of newer sili-

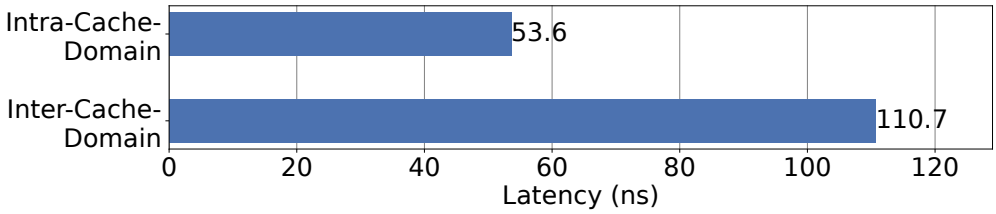


Figure 5.11: Cache to cache data transfer overhead on a platform with heterogeneous cache topology.

con process nodes continues to diminish. The decline in these technology trends has given rise to datacenter designs with greater heterogeneity [60, 79, 141]. To meet the ever-growing computing demands, CPU vendors have adopted chiplet-based architectures [58, 114, 137] to improve scaling and reduce manufacturing costs. A significant portion of our fleet is composed of platforms with chiplet architectures, which provide multiple last-level cache domains within a socket, leading to Non-Uniform Cache Accesses (NUCA) [91].

Non-uniform data transfer overhead. To investigate the performance implications of chiplet architectures, we use Intel MLC [19] to measure the core-to-core access latency on a production platform. Figure 5.11 shows the access latency for data shared between the cores within the same cache domain and for different cache domains within the same socket. We observe that the inter-cache-domain latency is 2.07× of the intra-cache-domain access latency. datacenter applications may span across multiple cache domains, owing to the fact that they are too large to fit within a single cache domain and/or be scheduled as such by the scheduler [10]. The disparity in access latency suggests that the memory allocator should allocate objects that are cache domain local. To this end, we propose NUCA-aware transfer caches that shard a singleton transfer cache into multiple chiplet-local caches.

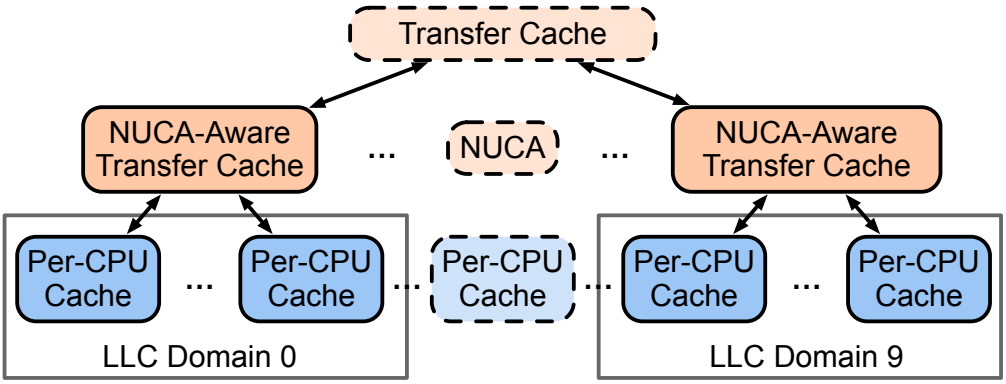


Figure 5.12: Structure of NUCA-aware transfer caches. We maintain a NUCA-aware transfer cache per cache domain.

NUCA-aware transfer caches. The legacy transfer cache is a centralized cache. In chiplet architectures, the legacy transfer cache may transfer memory objects between multiple cache domains. That is, objects freed by cores in one cache domain may be allocated by cores in another cache domain. Accessing such objects would require fetching data from non-local LLCs. To minimize inter-cache-domain sharing, we design NUCA-aware transfer caches that track an array of free objects local to each LLC domain. As shown in Figure 5.12, each NUCA-aware transfer cache only serves allocation and deallocation requests originating from its corresponding cache domain. We periodically release unused free objects in these transfer caches to prevent stranding over time. We also make sure to activate only as many NUCA-aware transfer caches as the application is scheduled on. Note that, we retain a centralized legacy transfer cache that backs NUCA-aware transfer caches, as it still offers cheaper memory allocation than the central free list.

Evaluation. Table 5.1 shows the performance improvement due to NUCA-aware transfer caches. Overall, the NUCA-aware transfer caches improve the cache locality, reducing the LLC load miss rate by 4.37%. In our fleet, we observe over 17.05% of the CPU cycles to be wasted due to back-end stalls [151].

Application	Throughput change(%)	Memory change (%)	CPI change (%)	LLC Load Miss (MPKI)	
				Before	After
fleet	0.32	0.10	-0.57	2.52	2.41
spanner	0.28	0.08	-0.42	3.80	3.21
monarch	0.62	0.32	-2.89	2.64	2.37
bigtable	0.47	0.10	-1.28	2.09	1.96
f1-query	1.05	0.01	-3.32	2.28	2.15
disk	1.72	0.62	-0.52	4.60	3.99
redis	/	/	/	/	/
data-pipeline	2.19	0.08	-2.69	1.82	1.39
image-processing	1.37	0.14	-8.02	0.81	0.52
tensorflow	3.80	0.16	-7.46	1.88	1.41

Table 5.1: Results of fleet-wide experiments and local benchmarks for enabling NUCA-aware transfer caches.

Due to an improvement in the LLC miss rate, we achieve 0.32% improvement in application throughput in the fleet. For the top 5 applications in the production fleet, we observe a throughput improvement of 0.28%–1.72% and a reduction in the cycles per instruction (CPI) of 0.32%–3.32%. Note that, due to an additional caching layer, we also observe an increase in fragmentation by 0.10% of the fleet memory. As we discuss earlier, even with this small increase in fragmentation, we see an outsized improvement in application productivity that results in overall server resource savings. Experiments with benchmarks described in Section 5.2.3 also show that we can increase throughput by 1.37–3.80% with a 0.08%–0.16% increase in memory usage. We again skip Redis in this study because it is single-threaded and does not benefit from optimizations targeting multi-threaded applications.

5.4.3 Central Free List

The central free list manages memory in spans, which are collections of TCMalloc pages. It fulfills requests from the transfer cache for one or more objects by

extracting free objects from spans. When the objects are returned to the central free list, each object is freed to the span it belongs to.

Diverse object lifetime. A span may only be released when all of the objects belonging to it are freed. However, object lifetime is extremely diverse. Even for objects of the same size class, any particular allocation might be freed instantly or may live forever (shown in Figure 5.8). The long-lived allocations prevent the spans from being freed, leading to increased memory fragmentation. We can potentially reduce memory fragmentation by using lifetime annotations (e.g., compiler-guided [117, 136] or application-specified) to allocate short-lived and long-lived objects on different spans. Indeed, prior work [108] uses machine learning to predict object lifetime, which can introduce significant runtime overheads.

Live allocations and span return rate. The central free list maintains different sets of spans to allocate objects for each size class. That is, the 8B and 16B objects are allocated from separate spans – an 8KB span may allocate up to 1024 8B objects or 512 16B objects. We use fleet-wide telemetry collected over a two-week period to study the correlation between the probability of returning a span to the pageheap, and the number of live allocations on a span. Figure 5.13 shows the release rates for spans with different numbers of live allocations for the 16B size class, where a span can allocate up to 512 objects. As the number of live allocations increases, the probability of a span release goes down.

For each size class, the central free list organizes spans in a singly linked list. It fulfills incoming allocation requests from a span at the front of that list. As such, the objects may be allocated from spans with the fewest live allocations that are most likely to be released, just because they happen to lie in the front

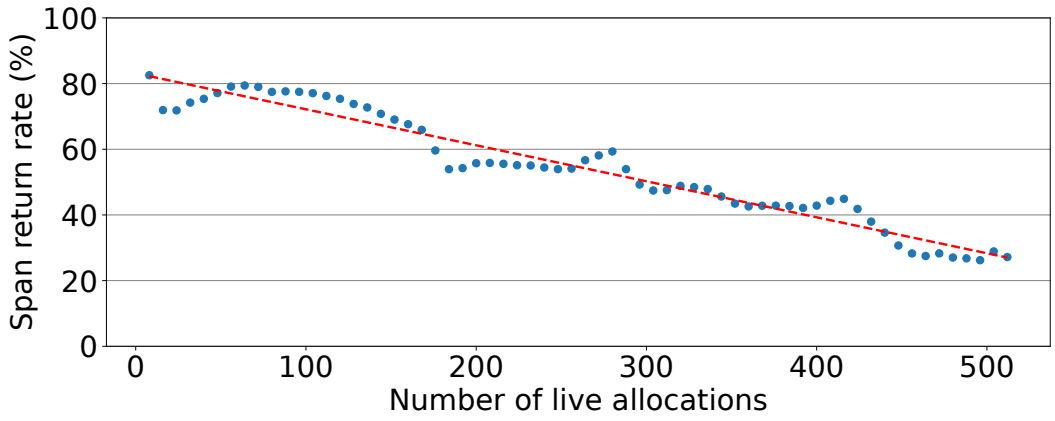


Figure 5.13: Correlation between the number of live allocations and span return rate for size class of 16 bytes.

of the linked list. We utilize the observation from Figure 5.13 to propose a prioritization scheme that allocates objects from spans that are least likely to be released.

Span prioritization. We aim to minimize memory fragmentation in the central free list by fulfilling incoming allocations from spans that have the least likelihood of being freed, while deprioritizing spans that are expected to be freed in the near future. We restructure the central free list to manage spans in L linked lists (instead of a singleton list) to track spans with varying occupancy separately. Spans with fewer live allocations on them are mapped to higher-indexed lists. Specifically, we map a span with A live allocations into a list indexed $max(0, L - \log_2(A))$. This allows us to differentiate spans with fewer allocations at a finer granularity – spans with 132 or 255 live allocations are unlikely to be released and can be mapped in the same list. Our experiments show that $L = 8$ lists are sufficient to differentiate spans.

The central free list allocates objects from spans in the list with the lowest possible index, since it contains spans with a higher number of allocated ob-

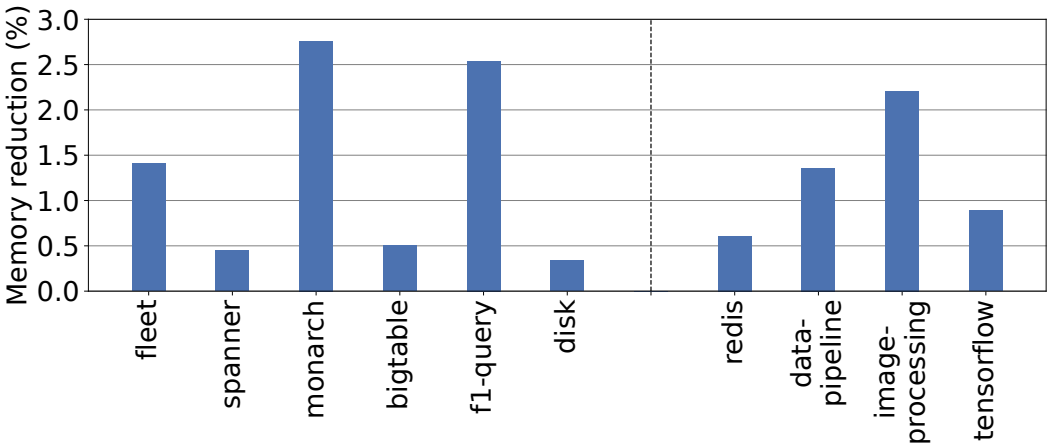


Figure 5.14: Memory reduction with span prioritization.

jects. We also move spans between the lists as required on each allocation and deallocation, as the number of live allocations on them change.

Evaluation. With span prioritization, the central free list can densely pack allocations on fewer spans. Figure 5.14 shows the resulting reduction in memory fragmentation. In a fleet-wide experiment, we achieve a 1.41% reduction in fleet memory usage. At our scale, this reduction leads to significant cost savings in server resources. The memory fragmentation in `monitor` reduces by 2.76%, and by 0.34%–2.54% for other fleet applications. We also confirm that the application's productivity metrics remain unchanged. For the benchmarks, we observe a memory usage reduction of 0.61%–1.36%.

5.4.4 Pageheap

TCMalloc's hugepage-aware pageheap [84] manages memory in hugepage-sized chunks to take advantage of Transparent Huge Pages (THP) [37], which provides an opportunity for the kernel to cover consecutive pages using hugepages in the page table. An entire aligned hugepage (typically 2MB on

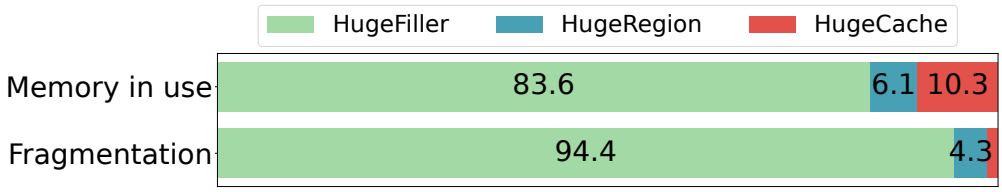


Figure 5.15: In-use memory and fragmentation (%) in the pageheap. HugeFiller is the major contributor to fragmentation.

x86) occupies just one TLB entry, which reduces stalls by increasing the TLB coverage and reducing TLB misses [96]. The pageheap plays a critical role in efficiently managing the memory layout to maximize TLB efficiency.

The pageheap [84] consists of three major components: (1) the *hugepage filler* handles allocation requests smaller than a hugepage. Here, spans are packed into hugepages; (2) the *hugepage region* is used for allocations that slightly exceed the size of a hugepage (e.g., 2.1MB). It packs such allocations on to a contiguous run of hugepages; (3) the *hugepage cache* also handles large allocations of at least a hugepage. Such allocations can generate slack (e.g., 1.5 MB slack from a 4.5 MB allocation), which is then donated to the hugepage filler.

pageheap fragmentation. While the pageheap cannot control the amount of memory that the application uses, it plays a critical role in placing those allocations in hugepage-aligned memory regions to improve the TLB efficiency. The pageheap is a major contributor to fragmentation, accounting for 51% of the total external fragmentation (Section 5.3). As shown in Figure 5.15, hugepage filler manages 83.6% of the total in-use memory and accounts for 94.4% of the pageheap fragmentation. Given this, we focus on the hugepage filler.

Hugepage filler. The hugepage filler allocates spans from hugepages-aligned memory regions. It frees up a hugepage when all the spans previously allocated from it are returned by the central free list. Similar to the span pri-

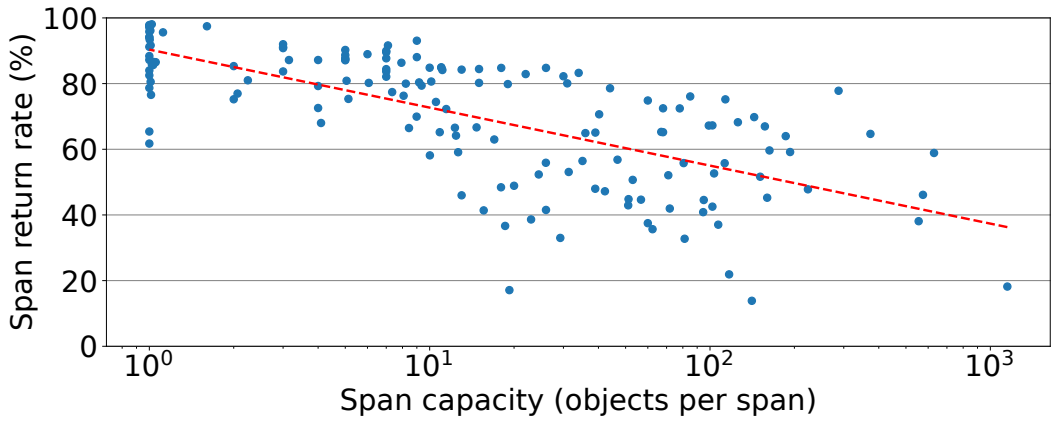


Figure 5.16: Correlation between the span capacity and span return rate for different size classes.

oritization mechanism that we discussed in Section 5.4.3, the hugepage filler prioritizes span allocations from hugepages that already have a higher number of allocations, and thus are least likely to be released. It assumes that spans themselves are independently and equally likely to become free. Instead, we analyze heuristics that can be used to identify spans that are more likely to be freed. We can then assign span allocations to different sets of hugepages based on their lifetime – we can place short-lived spans densely on fewer hugepages, thus improving TLB efficiency and fragmentation.

Span lifetime. We notice that spans of different size classes have diverse lifetimes. As we described in Section 5.2.1, TCMalloc uses a span to exclusively allocate objects of a particular size class; *span capacity* denotes the total objects for a size class that may be allocated from that span. For instance, an 8 KB span has a capacity of 1024 8B objects.

We perform a correlation study of span lifetime versus span capacity. Figure 5.16 shows the span capacity and its rate of returning from the central freelist to the hugepage filler for different size classes. We see a strong negative corre-

lation (with a Spearman's correlation coefficient of -0.75) between the capacity of a span and its return rate. In Figure 5.16, the leftmost data points show the spans allocating large size classes that can only hold one object. When the only object is returned, the span is released, resulting in a high span return rate. In contrast, spans with a significantly larger capacity are long lived and have a much lower return rate.

We use span capacity as a proxy for its lifetime to distinguish between short-lived and long-lived spans. Since a span is only released when all objects from it are freed, the object lifetime (Figure 5.8) is not necessarily a good measure for the span lifetime. It also requires lifetime annotations from compiler or applications that incur significant runtime overhead [108] . In contrast, span capacity is statically determined and can be used without any runtime overhead.

Lifetime-aware hugepage filler. We propose to make the hugepage filler aware of span lifetime to maximize the probability that a hugepage becomes totally free. That is, we aim to allocate short-lived and long-lived spans from separate sets of hugepages. To that end, we use dedicated hugepages for allocating spans with capacity $> C$ and $\leq C$. Our experiments reveal $C = 16$ as an acceptable threshold for separating span allocations. We replicate linked-list structures to track these dedicated hugepages separately. For incoming requests in each lifetime category, we prioritize allocating from hugepages that have the most allocations.

By differentiating between long-lived and short-lived spans, the lifetime-aware hugepage filler is able to densely place short-lived allocations on dedicated hugepages and release memory to the OS in complete hugepages. This improves hugepage coverage, reduces TLB misses, and boosts application per-

Application	Throughput change(%)	Memory change (%)	CPI change (%)	dTLB Load Walk (%)	
				Before	After
fleet	1.02	-0.82	-6.75	9.16	6.22
spanner	0.38	-0.45	-0.99	7.92	7.60
monarch	3.30	-0.05	-10.10	20.34	15.55
bigtable	2.83	-0.13	-4.44	17.25	15.00
f1-query	1.40	-1.40	-4.56	9.62	9.07
disk	6.29	-0.38	-17.61	8.42	6.55
redis	1.05	-7.02	-9.04	10.34	10.25
data-pipeline	1.43	-1.50	-2.76	5.36	4.97
image-processing	2.15	-1.29	-7.59	1.46	0.96
tensorflow	3.91	-2.69	-2.72	6.79	5.91

Table 5.2: Fleet workloads and benchmarks using the lifetime-aware hugepage filler. dTLB load walk (%) is the fraction of cycles spent in page walk, without accessing the L2 TLB.

formance. We also observe that smaller objects have higher access density. By separating spans with smaller size classes, we also efficiently utilize the limited TLB resources.

Evaluation. Table 5.2 shows results of enabling the lifetime-aware hugepage filler. The baseline implements the state-of-the-art hugepage-aware pageheap as proposed by Hunter et al. [84]. For the fleet experiment, we achieve 1.02% improvement in throughput and 0.82% reduction in memory usage. For the top 5 applications, we improve the throughput by 0.38%–6.29% and reduce the CPI by 0.99%–17.61%. Figure 5.17a shows the hugepage coverage for applications. We can see that the lifetime-aware hugepage filler improves hugepage coverage, increasing the average percentage of heap memory backed by hugepages from 54.4% to 56.2%. Due to improved TLB efficiency, we observe a 2.94% reduction in dTLB load walk cycles and an 8.1% reduction in dTLB misses (Figure 5.17b) in our fleet. This again supports our argument that optimizing for application productivity provides more efficiency gains than optimizing for the `malloc` CPU overhead alone, since the memory allocator has a substantial leverage on improving data locality and maximizing TLB efficiency.

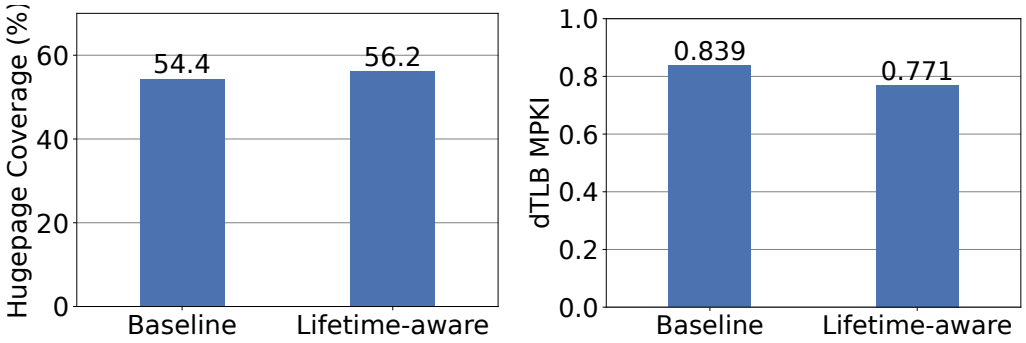


Figure 5.17: (a) Improved hugepage coverage rate and (b) reduced dTLB miss rate with lifetime-aware hugepage filler.

Benchmark experiments on a dedicated server also show that we can achieve a 1.05–3.91% increase in throughput with a 1.29%–7.02% reduction in memory usage.

5.4.5 Putting It All Together

Our characterization study shows that a datacenter memory allocator must accommodate the dynamic resource needs of datacenter applications, be aware of heterogeneity in hardware platforms, and leverage the diverse lifetime characteristics to improve data locality. Based on these insights, we redesign each cache tier in TCMalloc to propose a heterogeneous per-CPU cache, a NUCA-aware transfer cache, a central freelist with span prioritization, and a lifetime-aware hugepage filler.

Redesigning the memory allocator for warehouse-scale environments helps us achieve our ultimate goal: maximize the productivity of datacenter applications by using fewer server resources to complete the same or more units of work, even if it results in a lower reduction in the overall "datacenter tax". The designs in this work have been gradually rolled out to our fleet over a two-

year period. Given the ever-changing nature of datacenter workloads, it is non-trivial to perform a strict end-to-end evaluation of these designs. Regardless, we can estimate the aggregate performance impact of the four designs based on their relative improvement. We achieve a 1.4% increase in fleet throughput and a 3.5% reduction in fleet memory usage. For the top 5 applications, we achieve 0.7%–8.1% throughput and 1.0%–6.3% memory improvement. The source code for all the designs is open sourced and publicly available.

5.5 Discussion

In this work, we discussed certain design choices in TCMalloc as case studies following our characterization insights. Next, we discuss potential opportunities as future work.

Room at the top [100]. With Moore's law slowing down, performance gains need to come from improvements at the top of the computing stack [101]. In this work, we focus on the memory allocator to show how leveraging fleet-wide profiling and improving common libraries [89,140] can uncover horizontal efficiency opportunities. With diminishing returns due to technology scaling, optimizing common libraries can yield dramatic performance improvements.

Datacenter tax and productivity metrics. While understanding the datacenter tax can help us identify the largest building blocks of datacenter applications, reducing the tax itself may yield limited efficiency gains. The efficiency improvements to the `malloc` CPU overhead may yield up to a 4.3% improvement in fleet CPU, but a larger opportunity lies in optimizing for application productivity – 20-64% of CPU cycles incur memory stalls [89, 141] and optimiz-

ing for data locality can have a larger performance upside. While we present four case studies, several opportunities remain that may improve cache efficiency and/or hugepage coverage through improved allocation placements.

NUMA architecture and beyond. TCMalloc has a built-in support [18] for Non-Uniform Memory Access (NUMA) architectures. In NUMA mode, it duplicates the set of size classes and page allocator for each NUMA node, ensuring that allocations always return local memory. In this work, we demonstrate the need for the memory allocator to adapt to the heterogeneity of hardware platforms. We propose NUCA-aware transfer caches that preserve cache locality in platforms with non-uniform cache domains.

Cooperation with kernel features. Although TCMalloc is a userspace memory allocator, it needs to cooperate with the kernel to achieve the desired performance. It uses restartable sequences [31] to optimize the fast path of the per-CPU cache, avoiding the use of locks or expensive atomic instructions when accessing per-CPU data. Virtual CPU [55] support exposes a dense set of CPU IDs to index the per-CPU cache, which helps TCMalloc reduce the memory footprint of the front-end cache. TCMalloc also makes use of transparent hugepages [37] in the back-end pageheap [84] to improve hugepage coverage and reduce dTLB misses. These optimizations illustrate the need to leverage kernel features to improve the performance of a memory allocator.

Object lifetime and access density. In this work, we propose a lifetime-aware allocator that uses span lifetime to improve allocation placements in hugepages. We can also use user-defined or profile-guided [117, 136] lifetime and access density annotations that can further improve TLB efficiency.

5.6 Conclusion

We present the first comprehensive characterization study of TCMalloc, a memory allocator used in datacenters. We show that the memory allocator needs to adapt to the dynamic resource usage of datacenter applications, be aware of heterogeneity in hardware platforms, and utilize the objects' diverse lifetime to make memory packing decisions. Based on these insights, we redesign each tier in the TCMalloc cache hierarchy for datacenter environments, We evaluate these design choices using fleet-wide A/B experiments in our production fleet, resulting in a 1.4% improvement in throughput and a 3.4% reduction in RAM usage.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this dissertation, we optimize the foundational system build blocks for datacenter applications, from high-level computing frameworks, such as serverless computing to the underlying system software libraries, such as the memory allocator. In particular, we have made the following contributions.

- **Resource management for serverless applications:** To improve the resource efficiency of emerging serverless applications in datacenters, we built Aquatope, a QoS-and-uncertainty-aware resource manager for multi-stage serverless workflows. Aquatope shows that function cold starts and resource allocation are correlated and must be tackled jointly. Moreover, Aquatope addresses the importance of handling noise and uncertainty in datacenters. Aquatope demonstrates that a joint solution to cold starts and resource management, taking into account uncertainty, can effectively improve the resource efficiency of serverless applications.
- **Efficient serverless workflow execution:** Despite efficient resource management approaches, serverless workflows still suffer from increased latency due to control plane overheads. We presented Meteion, a fast and efficient serverless workflow execution engine for latency-critical applications. We demonstrate that the control plane and communication overheads are the major sources of degraded performance of serverless workflows. Meteion shows that by decoupling the control plane from the workflow execution, and enabling decentralized workflow orchestration and direct inter-function communication, we can make serverless workflows a feasible high-performance solution for latency-critical applications.

- **Memory allocator optimization:** One of the low-level system libraries with great potential to improve performance across the datacenter is the memory allocator. We performed the first comprehensive characterization study of TCMalloc, a memory allocator used in the production datacenter fleet. Based on our characterization, we derive unique insights and use them to design a memory allocator for datacenter applications. In particular, such an allocator needs to (1) adapt to the dynamic resource usage of datacenter applications, (2) be aware of heterogeneity in hardware platforms, and (3) utilize diverse lifetime information to make memory-packing decisions. Evaluation results show that redesigning the memory allocator for warehouse-scale environments significantly improves fleet productivity.

We believe that this dissertation opens up several directions for future work.

- **Efficient serverless LLM inference:** Serverless frameworks have been adopted by Large Language Model (LLM) applications. By serving LLM inference in a serverless manner, LLM service providers can efficiently multiplex LLMs on shared GPUs, which improves utilization, and helps application developers avoid the expense of long-term GPU provisioning. However, serverless inference also faces significant overheads, as loaded LLM checkpoints range from gigabytes to terabytes. Serverless frameworks can be further optimized to reduce the storage overhead of LLM checkpoints and achieve optimal scheduling for model loading and migration.
- **Self-tuning system libraries:** Foundational system libraries usually have multiple adjustable knobs (e.g., the cache sizes of a memory allocator)

that can affect the performance of an application. Different applications have different performance behaviors, therefore the optimal parameters for these libraries vary. Manually adjusting these performance knobs for all applications takes a substantial effort and is impractical. On the other hand, creating a system service to automatically tune these parameters for various applications has great potential to improve system efficiency.

BIBLIOGRAPHY

[1] Apache couchdb. https://couchdb.apache.org.

[2] Apache kafka. https://kafka.apache.org.

[3] Apache openwhisk. https://openwhisk.apache.org.

[4] Apache openwhisk composer. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer.

[5] Aws lambda. https://aws.amazon.com/lambda.

[6] Aws step functions. https://aws.amazon.com/step-functions.

[7] Azure durable functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[8] Azure functions. https://azure.microsoft.com/en-us/services/functions.

[9] Best practices for working with aws lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html.

[10] Completely fair scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html.

[11] Eigen linear algebra library. https://eigen.tuxfamily.org.

[12] Gevent. https://www.gevent.org.

[13] The gnu c library. https://www.gnu.org/software/libc.

[14] Google cloud functions. https://cloud.google.com/functions.

[15] Google workflows. https://cloud.google.com/workflows.

[16] How prewarmed containers are provisioned with a reactive configuration. https://github.com/apache/openwhisk/blob/master/docs/actions.md.

[17] Ibm cloud function. https://cloud.ibm.com/functions.

[18] Implement numa awareness in tcmalloc. https://github.com/google/tcmalloc/commit/ef7a3f8d794c42705bf4327ca79fa17186904801.

[19] Intel memory latency checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[20] Kubernetes. https://kubernetes.io.

[21] Lambda function scaling. https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html.

[22] Locust. https://locust.io.

[23] Managing concurrency for a lambda function. https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html.

[24] mi-malloc. https://microsoft.github.io/mimalloc/.

[25] Minio. https://min.io.

[26] Mongodb. https://www.mongodb.com.

[27] Nginx. https://www.nginx.com.

[28] Openfaas: Serverless functions, made simple. https://www.openfaas.com/.

[29] Pytorch. https://pytorch.org.

[30] Redis. https://redis.io/.

[31] Restartable sequences. https://github.com/torvalds/linux/commit/d82991a8688ad128b46db1b42d5d84396487a508.

[32] Restartable sequences. https://dynamorio.org/page_rseq.html.

[33] Serverless reference architecture: Image recognition and processing backend. https://github.com/aws-samples/lambda-refarch-imagerecognition.

[34] Serverless reference architecture: Real-time file processing. https://github.com/aws-samples/lambda-refarch-fileprocessing.

[35] Strace: Linux syscall tracer. https://strace.io.

[36] Tcmalloc. https://github.com/google/tcmalloc.

[37] Transparent hugepage support. https://www.kernel.org/doc/html/next/admin-guide/mm/transhuge.html.

[38] State of serverless. https://www.datadoghq.com/state-of-serverless, 2023.

[39] Advantages and disadvantages of cloud computing, 2024.

[40] Benefits of cloud migration, 2024.

[41] Colin Adams, Luis Alonso, Ben Atkin, John P. Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George T. Talbot, Adam Jacob Tart, and Nick Taylor, editors. *Monarch: Google's Planet-Scale In-Memory Time Series Database*, 2020.

[42] Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. *ACM SIGPLAN Notices*, 46(11):55–64, 2011.

[43] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 923–935, USA, 2018. USENIX Association.

[44] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 469–482, USA, 2017. USENIX Association.

[45] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[46] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Feb 2009.

[47] Autoscale. `https://cwiki.apache.org/cloudstack/autoscaling.html`.

[48] Aws autoscaling. `http://aws.amazon.com/autoscaling/`.

[49] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. Botorch: A framework for efficient monte-carlo bayesian optimization. In *Advances in Neural Information Processing Systems 33*, 2020.

[50] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[51] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[52] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 15(8), 2022.

[53] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), jun 2008.

[54] Shuang Chen, Christina Delimitrou, and José F. Martínez. 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, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Jonathan Corbet. Extending restartable sequences with virtual cpu ids. `https://lwn.net/Articles/885818`, 2022.

[56] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[57] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Middleware '20*, pages 356–370. ACM, 2020.

[58] Jeff Defilippi. Why chiplets and why now? `https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/why-chiplets-why-now`.

[59] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013.

[60] Christina Delimitrou and Christos Kozyrakis. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Transactions on Computer Systems (TOCS)*, 31(4):1–34, 2013.

[61] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014, March 2014.

[62] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo 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, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[63] Lieven Eeckhout. Is moore's law slowing down? what's next? *IEEE Micro*, 37(04):4–5, 2017.

[64] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[65] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[66] Jason Evans. Scalable memory allocation using jemalloc. `https://engineering.fb.com/2011/01/03/core-infra/scalable-memory-allocation-using-jemalloc/`, 2011.

[67] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[68] Alexander Fuerst and Prateek 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 2021, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[69] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1050–1059. JMLR.org, 2016.

[70] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 1027–1035, 2016.

[71] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.

[72] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina 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, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Yu Gan, Yanqi Zhang, Kelvin Hu, Yuan He, Meghna Pancholi, Dailun Cheng, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[74] Jacob R. Gardner, Matt J. Kusner, Zhixiang Xu, Kilian Q. Weinberger, and John P. Cunningham. Bayesian optimization with inequality constraints. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–937–II–945. JMLR.org, 2014.

[75] Jacob R. Gardner, Geoff Pleiss, David Bindel, Kilian Q. Weinberger, and Andrew Gordon Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7587–7597, Red Hook, NY, USA, 2018. Curran Associates Inc.

[76] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16, 2023.

[77] Jian Guo, He He, Tong He, Leonard Lausen, Mu Li, Haibin Lin, Xingjian Shi, Chenguang Wang, Junyuan Xie, Sheng Zha, Aston Zhang, Hang Zhang, Zhi Zhang, Zhongyue Zhang, Shuai Zheng, and Yi Zhu. Gluoncv and gluonnlp: Deep learning in computer vision and natural language processing. *Journal of Machine Learning Research*, 21(23):1–7, 2020.

[78] Yacine Hadjadj, Chakib Mustapha Anouar Zouaoui, Nasreddine Taleb,

Sarah Mazari, Mohamed El Bahri, and Miloud Chikr El Mezouar. Vcmalloc: A virtually contiguous memory allocator. *IEEE Transactions on Computers*, 2023.

[79] Md E Haque, Yuxiong He, Sameh Elnikety, Thu D Nguyen, Ricardo Bianchini, and Kathryn S McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 625–638, 2017.

[80] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[81] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. pages 261–272, 01 2011.

[82] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[83] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[84] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273. USENIX Association, July 2021.

[85] Rob Hyndman and Yeasmin Khandakar. Automatic time series forecasting: The forecast package for r. *Journal of Statistical Software*, 26, 07 2008.

[86] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[87] Zhipeng Jia and Emmett 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 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[88] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, Feb 2019.

[89] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[90] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating memory allocation. *SIGPLAN Not.*, 52(4):33–45, apr 2017.

[91] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGARCH Comput. Archit. News*, 30(5):211–222, oct 2002.

[92] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 427–444, USA, 2018. USENIX Association.

[93] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021.

[94] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE micro*, 30(4):8–19, 2010.

[95] Bradley C Kuszmaul. Supermalloc: A super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 International Symposium on Memory Management*, pages 41–55, 2015.

[96] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Im-*

plementation (OSDI 16), pages 705–721, Savannah, GA, November 2016. USENIX Association.

[97] Nikolay Laptev, Jason Yosinski, Li Erran Li, and Slawek Smyl. Time-series extreme event forecasting with neural networks at uber. In *International conference on machine learning*, volume 34, pages 1–5, 2017.

[98] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1), mar 2012.

[99] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*, pages 244–265. Springer, 2019.

[100] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[101] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[102] Benjamin Letham, Brian Karrer, Guilherme Ottoni, and Eytan Bakshy. Constrained bayesian optimization with noisy experiments. *Bayesian Analysis*, 14(2):495–519, 2019.

[103] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. Rambo: Resource allocation for microservices using bayesian optimization. *IEEE Computer Architecture Letters*, 20(1):46–49, 2021.

[104] Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. Nextgen-malloc: Giving memory allocator its own room in the house. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 135–142, 2023.

[105] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. *FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service*, page 782–796. Association for Computing Machinery, New York, NY, USA, 2022.

[106] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J Parkinson, Alex Shamis, Christoph M Wintersteiger, and David Chisnall. Snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, pages 122–135, 2019.

[107] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. *SIGARCH Comput. Archit. News*, 43(3S):450–462, jun 2015.

[108] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 541–556, New York, NY, USA, 2020. Association for Computing Machinery.

[109] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S McKinley, and Paul Turner. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 28–38, 2021.

[110] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[111] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[112] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), jun 2022.

[113] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, page 21, USA, 2019. USENIX Association.

[114] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 57–70. IEEE Press, 2021.

[115] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving, 2017.

[116] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, April 2016.

[117] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 2–14. IEEE Press, 2019.

[118] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360, 2019.

[119] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 679–692, 2018.

[120] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206, 2020.

[121] Bobby Powers, David Tench, Emery D Berger, and Andrew McGregor. Mesh: Compacting memory management for c/c++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–346, 2019.

[122] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association.

[123] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing architectural resources for all page sizes in x86 processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1106–1120, 2021.

[124] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[125] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[126] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[127] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Satori: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains¡sup¿*¡/sup¿. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 292–305, 2021.

[128] Rohan Basu Roy, Tirthak Patel, and Devesh 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 2022, page 753–767, New York, NY, USA, 2022. Association for Computing Machinery.

[129] Aakanksha Saha and Sonika Jindal. Emars: Efficient management and allocation of resources in serverless. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 827–830, 2018.

[130] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. SOSP '23, page 231–246, New York, NY, USA, 2023. Association for Computing Machinery.

[131] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan

Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. F1 query: declarative querying at scale. *Proc. VLDB Endow.*, 11(12):1835–1848, aug 2018.

[132] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1063–1075, New York, NY, USA, 2019. Association for Computing Machinery.

[133] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[134] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[135] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 69–87, USA, 2018. USENIX Association.

[136] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 617–631, New York, NY, USA, 2023. Association for Computing Machinery.

[137] Teja Singh, Sundar Rangarajan, Deepesh John, Russell Schreiber, Spence Oliver, Rajit Seahra, and Alex Schaefer. 2.1 zen 2: The amd 7nm energy-efficient high-performance x86-64 microprocessor core. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 42–44. IEEE, 2020.

[138] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable

low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[139] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, page 2951–2959, Red Hook, NY, USA, 2012. Curran Associates Inc.

[140] Akshitha Sriraman and Abhishek 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, page 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[141] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. Softsku: Optimizing server architectures for microservice diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 513–526, New York, NY, USA, 2019. Association for Computing Machinery.

[142] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[143] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10, 2020.

[144] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[145] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris 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 2021, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[146] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 133–145, USA, 2018. USENIX Association.

[147] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[148] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 698–710, 2019.

[149] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of ISCA*. 2013.

[150] Hanmei Yang, Xin Zhao, Jin Zhou, Wei Wang, Sandip Kundu, Bo Wu, Hui Guan, and Tongping Liu. Numalloc: A faster numa memory allocator. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*, pages 97–110, 2023.

[151] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[152] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '20. Association for Computing Machinery, 2020.

[153] Guoqiang Peter Zhang. Neural networks for classification: a survey. *IEEE*

Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 30(4):451–462, 2000.

[154] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[155] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery.

[156] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Edward Suh, and Christina 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)*, April 2021.

[157] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, et al. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[158] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 1–14, New York, NY, USA, 2022. Association for Computing Machinery.

[159] Lingxue Zhu and Nikolay Laptev. Deep and confident prediction for time series at uber. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 103–110, Los Alamitos, CA, USA, Nov 2017. IEEE Computer Society.