SERVERLESS CONTROL PLANES FOR ORCHESTRATION OF CLOUD RESOURCES

Alexander Joseph Fuerst

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Luddy School of Informatics, Computing, and Engineering

Indiana University

June 2024

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

Prateek Sharma, PhD (Chair)

Martin Swany, PhD

Dingwen Tao, PhD

Eleftherios Garyfallidis, PhD

June 10th, 2024

Veni Sancte Spiritus

ACKNOWLEDGEMENTS

TODO: Parents

Family

St Paul's Catholic Center priests & friends

All Saints Orthodox Church people

ISE Friends

Advisor Prateek

ABSTRACT

Cloud computing is comprised of a variety of services and abstractions to reduce the complexity of building and running applications. Function as a Service is a unique abstraction amongst the many: user supplied code entirely managed by the cloud provider. Serverless control planes face unique challenges caused by the highly heterogeneous workloads predominant. Users run machine learning inference, web services, multimedia analysis, and even scientific computing, to the tune of billions of invocations per day. These, in turn, vary widely in resource usage, execution time, and frequency of invocation, which the control plane must handle efficiently. It must also handle problems like cold start overheads neccessitated by executing invocations within isolated containers, and resource underutilization during idle periods.

This thesis addresses performance and resource management challenges by developing novel system designs and tailoring algorithms to handle these workloads. In it, we detail several algorithms that tackle poor resource allocation and load scheduling at both individual server and cluster levels. The key finding of orchestration at both levels is the reliance on function characteristics in decision-making. In addition, we describe a redesigned control plane, Ilúvatar, that reduces latency spikes by 100x in existing open-source systems. This is primarily accomplished by disaggregating scheduling decisions to avoid contention between distributed services of the control plane. Finally, we leverage the high degree of resource control made possible by this new system to integrate GPU acceleration into the serverless ecosystem. This uses a variety of novel mechanisms to minimize overhead despite limited device resources, boosting performance by several orders of magnitude over baseline solutions. Altogether, the contributions of this thesis serverless improve latency by 75% and cluster

resource utilization by up to 20x.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ABI: Application Binary Inferface

AES: Advanced Encryption Standard

AI: Artificial Intelligence

API: Application Programming Interface

ARIMA: AutoRegressive Integrated Moving Average

AWS: Amazon Web Services

CDF: Cumulative Distribution Function

CDN: Content Delivery Network

CH-BL: Consistent Hashing with Bounded Loads

CH-RLU: Consistent Hashing with Random Load Update

CH: Consistent Hashing

CLI: Command Line Interface

CNI: Container Netowrk Interface

CNN: Convolutional Neural Network

CPU: Central Processing Unit

CRIU: Checkpoint/Restore In Userspace

CSV: Comma-Separated Value

DAG: Directed Acyclic Graph

DNN: Dense Neural Network

DPU: Data Processing Unit

ECDF: Empirical distribution function

EEDF: Earliest Effective Deadline First

FCFS: First-Come First-Serve

FFT: Fixed-Fourier Transform

FIFO: First-in-First-out

FPGA: Field-Programmable Gate Array

FaaS: Function as a Service

GC: Garbage Colection

GCP: Google Cloud Platform

GD: Greedy-Dual

GPU: Graphical Processing Unit

HPC: High-Performance Computing

HTTP: Hyper-Text Transport Protocol

HW: Hardware

IAT: Inter-arrival Time

IP: Internet Protocol

IPC: Instructions per Cycle / Inter-Processor Communication

JSF: Shortest Job First

JSON: JavaScript Object Notation

JSQ: Join-Shortest-Queue

JVM: Java Virtual Machine

LFU: Least Frequently Used

LL: Least Loaded

LRU: Least Recently Used

LWL: Least-Work-Left

MIG: Multi-Instance GPU

ML: Machine Learning

MPI: Message-Passing Interface

MPS: NVIDIA Multi-Process Service

MQ: Multi-Queuing

MQFQ: Multi-Queue Fair Queuing

NVML: NVIDIA Management Library

OCI: Open Container Initiative

OS: Operating System

OW: OpenWhisk

PTE: Page-Table Entry

RAM: Random Access Memory

RAPL: Running Average Power Limit

RNN: Recurrent Neural Network

RPC: Remote Procedural Call

SIMT: Single Instruction Multiple Threads

SJF: Shortest Job First

SLA: Service-Level Agreement

SW: Software

TCP: Transmission Control Protocol

TPU: Tensor Processing Unit

TCB: Trusted Computing Base

TEE: Trusted Execution Environment

TTL: Time-to-Live

UVM: Unified Virtual Memory

VM: Virtual Machine

VT: Virtual Time

WASM: WebAssembly

1. Introduction

This thesis describes the current state of a new computing paradigm, *serverless computing*, and proposes improvements to various aspects of the control planes that orchestrate serverless computing systems. Serverless computing, otherwise known as Function as a Service (FaaS) has the possibility to revolutionize cloud computing. It emerged from out of the ethos of cloud computing where hardware is abstracted and *rented* to customers. Serverless computing is the most recent and exemplary of this abstraction, where the cloud provider totally manages users' applications using a FaaS *control plane*, orchestrating deployment, execution, scaling, resources, and more. Major cloud providers such as Amazon Web Services [2], Google Cloud [3], Microsoft Azure [4], and Alibaba Cloud [5], have taken up and popularized serverless since first appeared in 2016.

Traditionally, an application would be hosted by the developer using hardware that they personally owned and managed. This causes problems when trying to rapidly scale up the number of users for the application – additional hardware must be purchased (slowly), or the system must be over-provisioned to handle peak user load. Cloud computing platforms can provision new "hardware", in the form of a *virtual machine* (VM) [6], in a matter of minutes. The developer then deploys their application onto the VM and can connect it to the existing cluster of machines. When user load decreases, say at night, these VMs can be deleted and do not have to be paid for.

At the same time, software development practices were pushing to split up large monolithic applications into functionality-specific *microservices*. Deploying microservices onto individual

VMs lead to wasted resources and high orchestration overhead. Developers started to ship their applications inside pre-packaged *containers* [7], that contained all necessary dependencies for it to run. Control planes like Kubernetes [8] and OpenStack [9], along with many others, were also created that could deploy these containers and manage the services within them using some simple configuration. Cloud providers could then run these orchestration tools on their hardware, and deploy user-provided containers directly.

This approach is highly popular today, but has several drawbacks and overheads. Containers reduce, but do not wholly eliminate duplicate state, often consisting of several hundred MBs of dependencies. Developers still need to be aware of when, where, and how their application would run, now more convoluted because *they* were responsible for putting this together themselves. Mitigating such problems and reducing developer overhead required specialist "DevOps" positions to handle the complexity of deploying software. Idle services can not be eliminated at the granularity of container orchestration, so possible minutes of idle time between serving requests must be paid for.

Serverless computing completes this transition by taking responsibility for all of these concerns. Individual serverless *functions* are typically small, which are created by developers by directly uploading source code to the provider. It is then responsible for preparing the function's dependencies, and executing it when a request with arguments comes in. The user is only billed for resources used *during* this execution time, a unique shift in billing costs. Several functions can be chained into a complex *application* via a directed acyclic graph (DAG) description. The low-maintenance and pay-per-use model has proven highly attractive to customers, and FaaS growth since its inception in 2014 [2] has been dramatic.

Cloud providers are pushing users to this new platform because they get several benefits from this style of computing. A function's resources are considered *ephemeral*, and can be removed at the provider's discretion when not being used. The resource demands of a function are much smaller than that of a VM or container, allowing many hundreds

to be hosted on a single *server* (also called a *worker*). Many invocations can be run on a worker at a time, and the provider can load-balance these to maximize utilization of their clusters that are plagued by wasted resources [10–14]. The transition to more fine-grained resource management allows for more advanced optimizations and opportunities for improving resource utilization.

The value in serverless comes from its ability offer low-latency *performance*, which requires a *simple yet scalable* control plane tailored with FaaS-specific distributed computing optimizations. The primary components of such a control plane are outlined in Figure 1.1, highlighting the contributions of this thesis. A worker executing serverless functions can host dozens of concurrent executions and thousands of idle functions to fully utilize resources (Chs. 3 & 6). The entire control plane coordinates the execution of billions of invocations per day [15] and must scale to clusters of thousands of machines, guaranteeing good performance under uncertain conditions (Chs. 4 & 5). Active work on serverless spans from low-level virtualization and OS abstractions to high-level scheduling and load-balancing (Ch. 4) algorithms. This thesis discusses issues existing in current FaaS control planes and proposes methods, new designs, and future-thinking systems to tackle them.

This thesis hinges on the fact that serverless systems draw on solutions and designs from many fields of computing, but they must be adapted and modified to truly suit this paradigm. Given the abstraction FaaS provides, one can ask several questions: 1) What techniques help individual invocations achieve minimum latency?, 2) How must these change to keep global latency low as we scale to millions of invocations?, and 3) How do orchestration decisions by the platform impact latency? It analyzes the functions and workloads seen by serverless systems to accurately understand the conditions we must support. The systems designs presented here are tested empirically on real hardware, and bolstered by simulations matching the runtime characteristics of live systems. The chapters herein show that all the layers and pieces comprising serverless platforms are in fact opportunities for dramatically

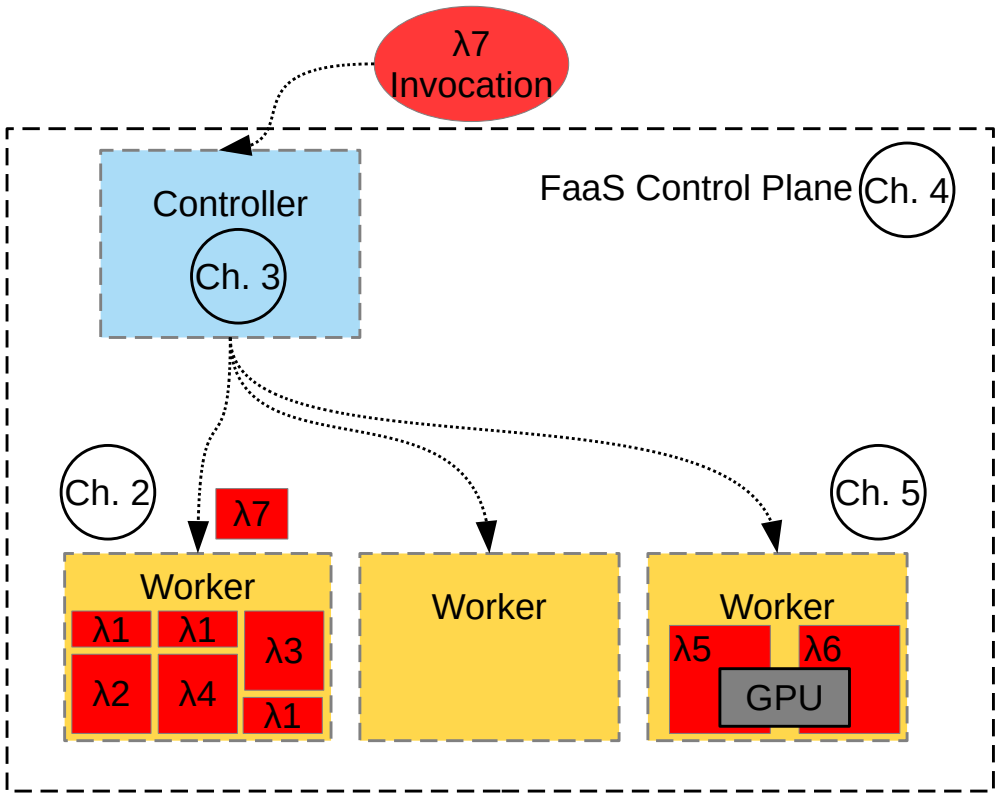


Fig. 1.1: The major components of the control plane, and the areas of each this thesis impacts. A controller accepts invocations for functions and distributes them amongst a cluster of workers. These in turn run invocations in isolated sandboxes

increasing performance and resource utilization.

FaaS control planes themselves have been designed, re-design, and ported to a variety of ecosystems to solve unique problems and constraints. FaaS has been placed along the edge-to-cloud continuum [16–19], taking advantage of its scale-to-zero capability. Moving serverless platforms beyond cloud computing to heterogeneous hardware has been explored [20], which is key for integration of IoT [18, 19, 21, 22] and FPGA [23, 24] hardware. Meta created an internal FaaS platform [15] which could be optimized using their knowledge of what

workloads run on it. Another unique entry, FuncX [25], creates a layer over supercomputing resources to run user experiments in a FaaS-like manner. The most well-known platforms are the public offerings of the major cloud providers, AWS Lambda [2], Google Functions [3], Azure Functions [4], and Alibaba Function Compute [5]. Several open-source equivalents have been made [26–29] which are popular in both research and as products for end-users. Included in this thesis is a design for a jitter-free, low-latency control plane and is highly configurable and able to run on heterogeneous and edge hardware.

More focused research into serverless has branched into nearly every field of systems research. To serve a function invocation, the system must create an isolated *sandbox* to execute the user code in. Numerous isolation mechanisms have been put forward, containers [7, 30, 31], language runtimes [32, 33], and lightweight virtualization [34]. All of these take time to start, in what's referred to as a *cold starts* and can significantly increase invocation latency. Additional research has targeted this problem specifically to accelerate existing isolation mechanisms creation time [35–37]. Future invocations see lower latency by benefiting from *locality*, where the isolation sandbox is re-used in a *warm start* invocation.

Knowingly keeping idle sandboxes resident in memory improves latency, but may lead to resource underutilization. Sandbox sizes are chosen by users, who often over-provision to negate performance problems [13, 38–41]. Worker nodes also cannot host a container per function, who often require several for ideal performance, therefore remove them periodically to conserve space [42]. The highly heterogeneous workload of FaaS has proven challenging when trying to predict when containers will be needed, which would allow their removal from memory to conserve resources [43, 44]. Adjusting the number of containers a function has reduces footprint, but may impact latency as they cannot be shared [45, 46]. Maintaining locality to provide acceptable performance while maximizing use of resources is an open area of research and addressed here.

The serverless abstraction allows for workers with various capabilities, not limiting them

to CPU-only computation. The major cloud providers currently do not expose alternative compute, but latency-critical applications like machine learning (ML) inference being moved onto FaaS platforms to take advantage its high scalability can also benefit from heterogeneous hardware. Accelerators come in many flavors: SmartNICs [47], GPUs [48, 49], FPGAs [23], and more [20, 50] – each with unique characteristics and types of applications they can support. A host of applications have moved to FaaS [51–56] that can leverage faster hardware. FaaS invocations are also unable to easily coordinate computation with each other [57–63], requiring slow intermediaries for interaction. These accelerators also require locality to achieve usable performance, having significant and often longer cold start times. A unique *data locality* for these devices exists too, as they must have data on hand that may have been cached on a system with more available memory. Running accelerated and massively parallel computation in FaaS that puts not restrictions on applications is important for adoption and continued growth.

We cannot consider the challenges of serverless solely from a single-worker perspective, as by design it exists at a cluster level. Functions have a variety of characteristics: execution time, inter-arrival-time of invocations, memory usage, and more. These can all vary in orders of magnitude, from sub-second to several minute runtimes and sub-second to daily invocation arrivals [43]. Even worse, functions are notoriously bursty, rapidly changing how frequently invocations occur. One could schedule function sandboxes [26, 64–66] and have the controller micro-manage the cluster state. Load balancing invocations [67–69] is a more scalable way to address the complex load and large cluster uncertainty, trusting workers to make ideal decisions. The cluster control plane targets locality in both cases, knowing that running invocations in existing sandboxes is much more performant.

1.1 Thesis Outline

This thesis is ordered as follows. The following chapter, Chapter 2, provides background on serverless computing, what technologies it builds on top of, and how it is being used in both research and by users. Serverless evolved from, and is still built on top of, abstractions in cloud computing like virtual machines (VMs) and containers. These mechanisms are used to isolate functions from one another and allow the control plane to control and protect resources. This control plane itself consists of several components, a *controller*, *workers*, and ancillary services. Users interact with the controller to create functions, invoke them, and receive results. It, in turn, load balances invocations amongst the cluster of workers that execute them using the aforementioned isolation tools. The many and varied designs presented by researches are shown and compared here. Techniques to improve isolation mechanism performance and load balancing are frequent. Wholly new systems have also been built, typically targeting a specific workload class such as ML, which has become popular in serverless. This exploration ends with a detailing of the many use cases serverless has been put towards, including ML, scientific computing, and even distributed computing.

Chapter 3 has the first contribution of this thesis and describes a container cache management design called *FaasCache*. Control planes keep idle function containers resident in memory in the expectation that they will be used again in the future. These warm executions are significantly faster than their counterparts that must wait for a container to be spun up. Unfortunately, memory is neither a free nor infinite resource, so control planes remove containers to both conserve and make room for others. FaasCache optimizes worker memory usage by treating these containers as a *cache*, and carefully considers what to evict when under resource constraint. Each worker monitors function characteristics such as memory usage, frequency, and container startup time. These are fed into a policy deciding which will be more valuable to keep given the cost of having to re-create it and how soon that might be. It also monitors the cache-hit ratio of invocations to dynamically

resize the memory allocated to the cache for targeting performance.

The next chapter, chapter 4, moves up a level in the control plane to present a novel load-balancing algorithm. As FaasCache shows, functions benefit from running in warm containers, which is referred to as function *locality*. The heterogeneous nature of functions would imbalance workers if we sent each function's invocations to a single worker. Some would be overloaded and suffer significant performance degradation, others sitting idle. Our *CH-RLU* algorithm targets locality for functions while at the same time avoiding worker overload. We use *consistent hashing* [70] to give perfect locality and distribute functions amongst workers. To detect overloads, we keep usage reports from each worker and add anticipated load from dispatched invocations to such reports. Extremely popular functions, those with the highest frequency of invocations, also model Gaussian noise of the load impact before sending an invocation, to model overloading scenarios. In all cases, if we predict a worker has too much work, we direct invocations away from it in a fixed pattern, maintaining locality while minimizing overloaded workers. The effectiveness of this at both minimizing platform overhead and keeping load even across worker clusters is described at the end of the chapter.

Next, Chapter 5 details a new serverless control plane design called Ilúvatar. This new control plane is written in Rust and seeks to solve the deficiencies in current open-source control planes used for research [26]. Its worker is designed to be highly modular, allowing swap-able implementations to support heterogeneous platforms and ease comparisons for research. It supports a novel queue mechanism to support new designs that may not run all workloads immediately, or handle cases of severe resource demand. The controller is built on a stateless design and uses the load balancing algorithm from Chapter 4, relying on the worker's local knowledge for scheduling. The third runtime component is a time-series database [71] used to aggregate function and worker metrics, reducing communication overhead between platform components. Finally, and a first for such systems, it has a

built-in load generation suite that integrates seamlessly with both worker and load balancer. With this, one tool can test any level of the control plane under various load conditions, and record highly detailed information for post-experimental analysis. We use this tool to compare with other control plans and show the performance benefits of Ilúvatar.

The final piece of this thesis is Chapter 6, which uses Ilúvatar to multiplex GPU resources to serve black-box functions. To prevent maintain control of GPU resources, we insert a *shim* in between the function and GPU driver to intercept calls. This lets us both over-subscribe memory and control when the memory is on-device or moved to the host. We leverage the increase in memory control to allow many containers to share one GPU, creating a *warm-pool* similar to that of regular CPU serverless. As GPUs cannot support the same concurrency as many-cored CPUs, we use the queue mechanism of Ilúvatar to dynamically control ordering and dispatching of invocations. The queue monitors GPU utilization and sends invocation when compute is available, moving memory around to ensure we don't overload the device. Invocations also benefit from locality here, successive runs and having their memory available gives better execution performance. We balance these needs with fairness, using the queue to prevent any function from starving others of device time. The effectiveness of all these are demonstrated with thorough experimental analysis at the end of this chapter.

2. Background: Serverless computing and Function as a Service

Serverless computing builds on architectures and designs used throughout cloud computing. These have been adapted and specialized in a myriad of ways by researchers and companies to both improve performance and enable new features. This chapter starts by describing in detail how serverless cloud control planes operate, and the systems used to build them. Additionally, the plethora of FaaS research in areas related to this thesis are examined and compared against.

2.1 What is Serverless Computing?

Serverless computing presents a departure from previous iterations of cloud computing architectures. Users can run arbitrary applications without concern for how or where it is ultimately run. FaaS control planes operate as a complex distributed system, operating from the lowest levels of OS abstractions to the highest levels of cloud system designs.

All such control planes work similarly and follow the generic architecture shown in Figure 2.1. Users upload their *source code* (1) such as those in Figures 2.2 and 2.3 to the cloud provider to create a **function**, and such functions can be chained together inside the provider's system to form a larger **application**. When the code is **invoked** (e.g. an HTTP request is made (2)), the provider routes it to a worker (3), creates a sandbox for it and executes the function (4), passing in any custom parameters. This sandbox must provide isolation, both covering both resource and security, ensuring that hogs or malicious actors do

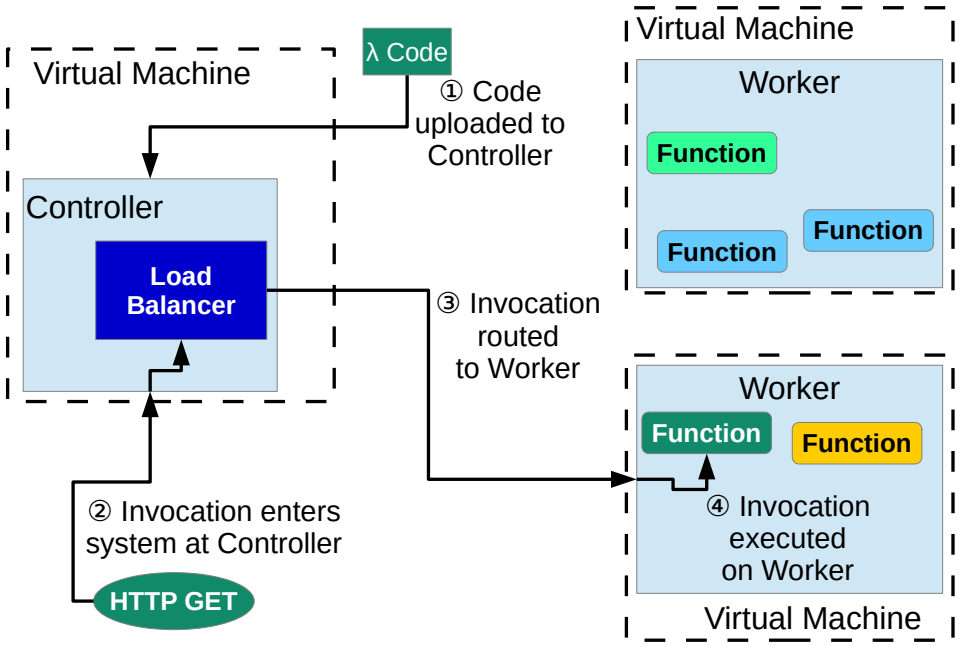


Fig. 2.1: A common architecture for serverless control planes. A controller distributes invocations to workers who run them inside containers.

```python
# Initialization code
import numpy as np
import tensorflow as tf

m = download_model("http://model_serve/img_classify.pb")
session = create_tensorflow_graph(m)

def lambda_handler(event, context):
    # This is called on every function invocation
    picture = event["data"]
    prediction_output = run_inference_on_image(picture)
    return prediction_output
```

Fig. 2.2: A classic serverless function: simple Python code performing ML inference on image data. In this example library and model initialization are done before execution starts.

not interfere with co-located executions. Often, this sandbox utilizes existing technologies such as Docker [7] or VMs. The major cloud computing providers have all created offerings, Amazon Lambda [2], Google Cloud Functions [3], Azure Functions [4], alternative providers such as IBM [26] and Alibaba [5] have joined in, and even non-commercial open-sourced control planes exist [27, 72].

FaaS functions are extremely diverse, encouraged by platforms who want to make it easy to adapt code of any type. An extremely common example is Figure 2.3, a microservice written in JavaScript that will operate as part of a larger application. The other common language is Python, often used for ML inference functions like Figure 2.2. Previously running applications like these would need VMs to host each piece, or managing a complex orchestration tool such as Kubernetes [8]. The transition to FaaS also removes the need to manually scale how many instances of each service are running as user demand grows and shrinks.

In a change from the billing model for rented VMs, users are billed only for the time their code is executing, often in small millisecond-sized time slices. Most providers set the cost to a formulation of the amount of memory used per time period [73], roughly $\$1.66 \times 10^{-5}$ per GB/second. Should the provider choose to keep that sandbox resident in memory to use for

```
const doc = require("dynamodb-doc");
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {
    const done = (err, res) => callback(null, {
        statusCode: err ? "400" : "200",
        body: err ? err.message : JSON.stringify(res),
        headers: {
          "Content-Type": "application/json",
        },
    });
    switch (event.httpMethod) {
        case "DELETE":
            dynamo.deleteItem(JSON.parse(event.body), done);
            break;
        case "GET":
            dynamo.scan({ TableName: event.queryStringParameters.TableName }, done);
            break;
        case "POST":
            dynamo.putItem(JSON.parse(event.body), done);
            break;
        case "PUT":
            dynamo.updateItem(JSON.parse(event.body), done);
            break;
        default:
            done(new Error("Unsupported method '${event.httpMethod}'"));
    }};
```

Fig. 2.3: Functions have varied tasks and implementation languages. This JavaScript function is designed to operate a microservice as part of a larger application.

Tab. 2.1: FaaS workloads are highly diverse in their resource requirements and running times. The initialization time can be significant and is the cause of the cold start overheads, and depends on the size of code and data dependencies.

Application	Memory size	Run time	Initialization time
ML Inference (CNN)	512 MB	6.5 s	4.5 s
Video Encoding	500 MB	56 s	3 s
Matrix Multiply	256 MB	2.5 s	2.2 s
Disk-bench (`dd`)	256 MB	2.2 s	1.8 s
Web-serving	64 MB	2.4 s	2 s
Floating Point	128 MB	2 s	1.7 s

a future invocation, the user will not be charged nor be aware that it is happening, save for lower latency on future invocations.

2.1.1 Function Isolation

Each function is run inside an isolated sandbox environment such as a Docker container [7], or a lightweight VM such as Firecracker [34]. By encapsulating function state and any side effects, the virtual execution environment provides isolation among multiple functions, and also allows for concurrent invocations of the same function. Due to the overhead of starting a new virtual execution environment (i.e., container or VM), and initializing the function by importing libraries and other data dependencies, function execution thus incurs a significant "cold start" penalty. Table 2.1 shows the breakdown of initialization time (last column) vs. the total running time of different FaaS applications, and we can see that the initialization overhead can be as much as 80% of the total running time. Thus, FaaS can result in significant performance (i.e., total function execution latency) overheads compared to conventional models of execution where applications can maintain application state between handling user requests and do not face the high initialization and cold start overheads.

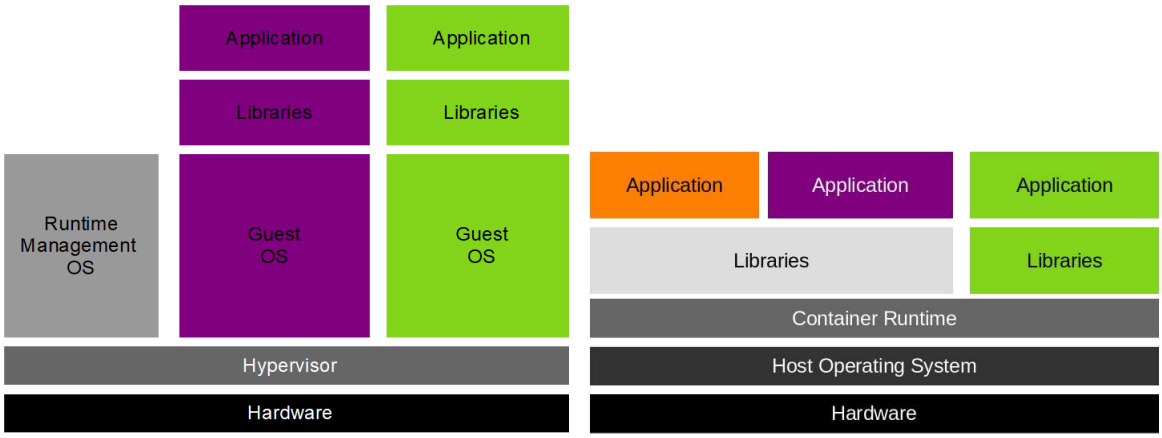
2.2 Virtualization for FaaS

2.2.1 Virtual Machines

To both prevent takeover of the physical hardware and ensure isolation between different users, cloud providers typically offer virtualized infrastructure [6]. Mimicking the stack in Figure 2.4a, user applications run inside a virtual machine and cannot directly access the hardware. A hypervisor manages a set of virtual machines that have a private OS inside them. Using hardware virtualization techniques, the provider's hypervisor interposes itself between guest and hardware, maintaining total control. Memory is protected via virtual memory in the CPU, protected instructions are trapped to the hypervisor, and network and disk I/O interaction can be run through the hypervisor. A major drawback is that users must install (duplicate) copies of OS's, libraries, and applications into each VM, and are responsible for maintenance of the guest OS.

2.2.2 VM Resource Management

A number of works have sought to reduce the overhead from virtualization by techniques such as securely exposing hardware to guests [74] or reducing layers of indirection [75]. A number of optimizations to memory usage that reduce the footprint of VMs were outlined by [76], which are still used in production hypervisors today. Choosing where to place VMs, knowing that they may run for days or months, is critical to reduce fragmentation on hosts across provider datacenters. Bin-packing studies have sought to minimize fragmentation [77] by overcommitting resources even at the risk of violating capacity. Even today, after much research in the area, providers see 20%-40% of unallocated resources [11] that they seek ways to make use of.



(a) Virtual Machines (b) Containers

Fig. 2.4: Different layers of abstraction between hardware and kernel based virtualization.

2.2.3 Containers

Enabling both improved process-level protections and a viable alternative to hardware virtualization, a new method of isolation was devised. Kernel virtualization, built originally around Linux *cgroups*, provides similar isolation and security guarantees to VMs. A combination of kernel utilities enables limiting CPU, memory, and I/O usages, restricting views of the file system, blocking interaction with other processes, and more. Popularized by various projects [7, 78], such *containers* reimagined how cloud computing and applications could work. Multiple applications could share the same operating system, and even libraries, safe from all interference from one another. Developers could ship applications, complete with any dependencies, that ran securely, anytime, and anywhere. Providers have begun offering services around these, tools like Kubernetes [8] orchestrate placement of containers onto hosts, removing the need for users to rent VMs at all.

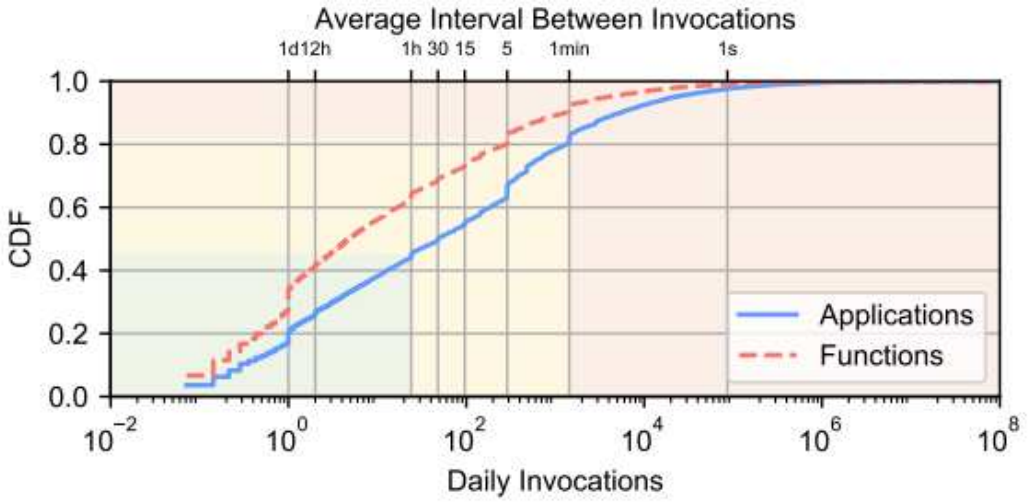


Fig. 2.5: A CDF of daily invocations for functions. Invocation frequencies taken from an Azure dataset range from sub-second to less than one per day. Figure from [1]

2.3 Serverless Control Plane Research

2.3.1 Workload Characterization

Published traces from major providers give a glimpse into the scale of serverless computing. Shahrad et al. [1] publicized the first dataset of the functions served by Azure Functions, with a detailed characterization breakdown. As seen in Figure 2.5, invocation rates are *extremely* heavy tailed. The most frequently invoked functions are executed several times a second, while the rarest perhaps once per day. Because of this, a small portion, 19% of functions, account for 99.6% of invocations in Azure. A paper describing an internal FaaS platform at Meta [15], showcased handling trillions of invocations daily and a similar skewed workload.

Several examples of functions are shown in Table 2.2, these are taken from FunctionBench [79]. Functions have high variation in warm and cold start times, with cold start times being significantly higher. Memory usage can range from a few dozen megabytes, to

Tab. 2.2: FaaS workloads are highly diverse in their resource requirements and execution times. The initialization time can be significant and is the cause of the cold start overheads, and depends on the size of code and data dependencies.

Application	Memory size	Warm time (sec.)	Cold start time (sec.)
Web-serving	64 MB	0.179	1.153
ML Inference (CNN)	512 MB	2.211	7.833
Disk-bench (dd)	256 MB	1.068	2.944
Floating Point	128 MB	0.083	1.432
Image Manipulation	300 MB	4.806	5.268
Matrix Multiply	256 MB	0.117	1.067
AES Encryption	128 MB	0.587	2.064
Video Encoding	500 MB	10.28	11.51
JSON Parsing	256 MB	0.414	1.962

half a gigabyte, matching real-world data from [1]. Control planes must be able to handle these high variations in frequency and resource usage, and treat all functions fairly.

2.3.2 Cold Start Mitigation

Many techniques have been proposed to reduce the initialization overhead from cold starts. They fall into two general categories, the first try to prevent cold starts by smart container management, and the latter reduce the time of a cold start.

2.3.2.1 Keep-Alive Locality for FaaS resource management has been explored in the form of function *keep-alive* policies. Once a container for a function is created and the function finishes execution, the container can be kept resident in-memory instead of immediately terminating it. Subsequent invocations of the function can then *reuse* the already running container. However, keep-alive is not a panacea for all FaaS latency problems. Specifically, idle containers being kept alive in anticipation of future function invocations can reduce the efficiency of the servers by intentionally not using resources.

Designing general keep-alive policies is challenging due to the extreme heterogeneity in the different function popularities, resource requirements, and cold start overheads. Commonly,

platforms use a default, fixed, time-to-live (TTL) to wait before deleting idle containers [26]. Predicting when future invocations will occur has been tried [1]. The platform can use this prediction to keep idle containers or remove them expecting it won't be needed for some time and re-create ahead of time when anticipated. FaaS users have even taken it upon themselves to keep their containers alive in various ways [37, 80] to avoid cold starts.

2.3.2.2 Reusing Initialization Performing multiple container startups of a function will involve executing the same startup procedures repeatedly: language runtimes, the OS of a micro-VM, etc. Utilizing past startups by checkpointing the container at a known position and booting from there has proved promising in reducing cold start times [36, 81]. Memoization approaches that leverage existing containers to duplicate their state have proven highly effective [36, 82].

2.3.2.3 Alternate Isolation Mechanisms While a simple implementation would run functions inside Docker containers [7], so long as isolation guarantees are upheld, using alternate mechanisms with faster start times is effective. Directly optimizing the isolation mechanism [34] or common libraries and language runtimes used by FaaS workloads [83] both give immediate performance boosts. Running functions inside the worker's language runtime can dramatically reduce initialization times [32, 36, 81, 84]. Security may be relaxed between functions in a larger application [85, 86] allowing for the reuse of sandboxes.

2.3.2.4 Worker Resource Management Worker resources are not infinite, especially memory for keeping function sandboxes available. Packing functions together can improve memory utilization [87]. Developers often over-assign memory, causing under-utilization, the control plane is in the perfect place to minimize this waste [38, 40]. Concurrent invocations of the same function has been reused to share resources, as isolation may be relaxed [88].

The initialization overheads of serverless functions and their repeated invocations have

spawned a great deal of research into optimizing their resource management. Recent surveys [89–93] provide an overview of the challenges and solutions in this very active research area.

2.3.3 Load Balancing

Serverless control planes are a distributed system, assigning invocations to worker nodes. The load-balancing of invocations can significantly affect latency, from either overloading workers or causing excessive cold starts. Several works have used locality as the primary way of ensuring invocation warm-starts, trying to direct a function to the same worker(s) [68, 94]. FaaS load-balancing may be seen as akin to VM bin-packing, and some have used reinforcement learning to choose sandbox placement [64]. Predicting a window of function's invocation arrivals and preparing a container for it [43] works for relatively uncommon functions. Scheduling functions based on their larger application DAG can minimize startup and communication time [66, 95–100]. Functions may compete for certain resources like network, so scheduling them on different machines can speed up certain workloads [101].

Package-aware load balancing [94] identifies and uses function code dependencies (software packages) as an important source of data locality. While this is an important factor, we focus on the in-memory locality of kept-alive functions since memory capacity is much smaller than permanent storage and caching functions in memory has a very large performance impact. CPU contention and interference is a major source of performance bottlenecks for co-located functions, and adjusting CPU-shares using cgroups can provide significant benefits [102–104]. The repetitive nature of functions and their workflows can also be used to improve resource utilization and latency [105–108] to select workers and sandboxes before they are needed.

The tradeoff between locality and performance has also been explored in the context of

delay scheduling [109] for data-parallel applications such as MapReduce. Load-balancing is seen as a "dispatch" problem in queuing theory, and the FaaS cluster system most closely approximates G/G/PS, since the arrivals and service times are not Markovian. Techniques such as "join the shortest queue", and "least work left" [110] have been shown to be effective. The online-greedy policy evaluated in the previous section closely approximates least-work-left. However, it is difficult to implement in practice since the running times of functions is hard to predict due to their volatile arrival distribution mixtures and high variances in running time due to various system interference effects.

2.3.4 Heterogeneous Hardware

Numerous control plane designs have been proposed targeting various accelerators such as SmartNICs, GPUs, DPUs, and FPGAs [20, 47, 48, 111]. One load balancing design sought to lower latency by mixing in low- and high-end servers for executing functions [112], using cheap and plentiful memory to host many sandboxes with a few fast machines to speed up popular functions.

2.3.5 Serverless Data-plane

A lot of work has gone into improving the performance hit caused by serverless functions needing to repeatedly download data and state from remote storage. Commonly used data can be cached on worker nodes for faster function access [38, 39]. Fault-tolerant storage systems targeting serverless systems can accelerate shared worker between invocations [60,62]. Co-locating different functions that access the same data to identical machines can improve data locality [66].

2.4 Application Mitigations of Control Plane Deficiencies

Researchers have often found the offerings by cloud providers to be lacking, but naturally do not want to host their own serverless control plane. To compensate for this, many have developed workarounds for issues such as poor performance and missing features. Even cloud providers face bottlenecks when rapidly scaling a function to several thousand workers, hence [113] allocates very large sandboxes and manually run multiple functions inside them. To overcome the lack of communication, [58] creates a secondary VM to punch TCP/IP holes that enables an MPI-like interface for custom Python functions.

2.5 Serverless Applications

A variety of applications have been built on serverless computing, in all manner of industries, use cases, and scales.

FaaS can be used to distribute embarrassingly parallel tasks such as MapReduce [114] or a `make` task [115]. A common use case is parallel encoding of videos using hundreds of workers [53, 116] or performing analytics on live video [50, 54]. On-demand scaling and usage has made FaaS attractive as a place to run streaming applications [117–119] and real time [120, 121] tasks.

The event-driven nature of serverless execution has garnered a lot of excitement from the IoT community [21, 22, 122, 123]. Edge computing pairs nicely with the ephemeral and on-demand execution model of serverless and has seen significant work [16, 18, 19]. Industrial applications, especially for monitoring and control have proven popular [124–126]. Even running interactive multiplayer video games on top of serverless computing has been explored [127].

Complex distributed applications are often a poor fit to be split into functions and linked into a serverless application. One attempt by [59] recreates Apache Zookeeper as a test-case

for distributed serverless systems. It suffers scalability issues due to high communication overheads between invocations over external storage.

2.5.1 ML in Serverless

Machine learning in all its forms has made its way into serverless research. Commonly to make scheduling decisions for containers [64] or resource allocation to them [38, 40]. Others have built control planes or systems dedicated to inference, targeting the performance bottlenecks in FaaS [51, 52]. And finally there are works that do training, taking advantage of the serverless scaling [63, 128, 129].

2.5.2 Scientific Serverless Computing

A group has made a serverless control plane that can connect with university supercomputing resources to run scientific workloads [25]. FaaS scalability has been used to accelerate biomedical research [55, 130]. Others have performed common linear algebra computations [56, 131] and optimization algorithms [33] on FaaS.

3. Keeping Serverless Computing Alive with Greedy-Dual Caching

Keeping every created function container indefinitely is not feasible for serverless control planes. They can be invoked sparsely – perhaps a handful of times per day. When a container isn't handling an invocation, it sits idle, occupying memory the control plane may want to use for other, more active, purposes. How long to keep a function sandbox in memory is called a **keep-alive** decision, and has a dramatic effect on latency.

The primary goal of keep-alive is to amortize the initialization and cold start latencies by keeping functions alive for different durations based on their characteristics. Keep-alive policies must be generalizable and yield high server utilization, because servers must handle hundreds of short-lived functions concurrently. Functions can have vastly different characteristics, and keep-alive policies must work efficiently in highly dynamic and diverse settings. We use the following characteristics of functions for keep-alive policies.

The **initialization time** of functions can vary based on the code and data dependencies of the function. For example, a function for machine learning inference may be initialized by importing large ML libraries (such as TensorFlow, etc.), and fetching the ML model, which can be hundreds of megabytes in size and take several seconds to download. Functions also differ in terms of their **total running time**, which includes the initialization time and the actual execution time. Again, functions for deep-learning inference can take several seconds, whereas functions for HTTP servers and microservices are extremely short-lived (few milliseconds). The **resource footprint** comprises the CPU, memory, and I/O use, and also differs widely based on the application's requirements. Finally, functions have

different **frequencies** and invocation rates. Some functions may be invoked several times a second, whereas other functions may only be invoked rarely (if they are used to serve a very low-traffic website, for instance).

3.1 Related Works

Mitigating cold starts is one of the central performance problems in FaaS, and has received commensurate attention in both academia and industry. The initialization or startup time of functions can be reduced by reducing container startup overheads [85, 132, 133], or deploying functions inside ultra-light containers, VMs, or unikernels [34, 134]. While these mechanisms can reduce the cold start overhead associated with the virtual environment creation, other sources of overheads remain, such as losing all application initialized variables, cached files, etc. As we have shown, keep-alive essentially serves the role of caching, and fast startup only reduces the "miss" penalty, and does not eliminate it.

Catalyzer [36] implements new mechanisms for checkpointing and restoring application and sandbox state, which significantly reduce the initialization cost of functions deployed in their gvisor-based sandbox environment. Our approach is complementary to these techniques since we focus on retaining the entire execution environment instead of optimizations for restoring/recreating it. Keep-alive policies can be combined with these optimized mechanisms to improve system-wide performance even further.

Principled keep-alive policies for functions have recently gained attention: the recent dataset and policy from the Azure function trace [1] shows the importance and effectiveness of keep-alive policies. In contrast to our work, their policy does not take the function size into consideration and uses a time-series prediction approach (effectively capturing recency and frequency), and combines it with a predictive "prefetching" approach. As we have shown, function memory footprints are a crucial characteristic, and the use of caching allows

the use of advanced analytical and modeling approaches for serverless computing in general. Earlier work has focused on simple "warm container pools" [135], in which Kubernetes cluster runs a certain number of warm containers for functions. Our caching-based policies take this one step further and decide *which* container to keep-alive, and for how long. Polling to keep cloud functions warm has also been a popular method [37, 80].

Our work considers functions individually—function scheduling with DAG based approaches [136] is effective for function-chains, and are orthogonal and complementary to our work. Hiding function latency using data caching (such as redis) for database applications is investigated in [137]. The ENSURE [138] system handles keep-alive and resource provisioning for CPU resources using queuing theory techniques. Our focus is on memory-constrained keep-alive and provisioning, and CPU-focused approaches are complementary to our work.

3.1.1 Caching Background

Our answer to solving the twin conundrum of keep-alive and provisioning that is robust to workload heterogeneity and dynamism, is to use concepts from a related, well-known field with the same challenges. Caching has a long history of robust eviction algorithms that use temporal locality such as LRU (Least Recently Used). The effectiveness of a caching algorithm depends on the workload's inter arrival time distribution, the relative popularities of different objects, and thus many variants of LRU such as LRU-k [139], segmented LRU [140], ARC [141], and frequency based eviction such as LFU [142], are widely used in caching systems. Because functions show a lot of diversity in their memory footprints, and since keep-alive is primarily constrained by server memory, we seek to use *size-aware* caching methods. While conventional caching algorithms and analytical models largely deal with constant-sized objects, many size-aware caching policies have been developed for webpages and data [143]. In particular, we use the Greedy-Dual [144] online caching framework that deals with objects with different eviction costs that are determined based on size and other

factors. The Greedy-Dual family of eviction algorithms for non-identical objects can be extended in many ways. We use a common variant, Greedy-Dual-Size-Frequency [145–147], which considers the size and frequency of objects.

Caching has a rich collection of analytical and modeling techniques to determine the efficacy of caches for different workloads. Hit (or miss) ratio curves are widely used for cache sizing to achieve a target performance, and for understanding and modeling cache performance. Hit-ratio curves can be constructed both in an offline and online manner, using techniques involving reuse distances [148], eviction times [149], Che's approximation [150], footprint descriptors [151], and estimation techniques such as SHARDS [152], counterstacks [153], etc.

3.2 Keep-alive Tradeoffs

In this section, we first present an empirical analysis of cold start overheads of common serverless applications, followed by the tradeoffs in keep-alive policies.

System model. We assume that each function invocation runs in its own container. A FaaS control plane may use a cluster of physical servers and forward the function invocation requests to different servers based on some load-balancing policy. Our aim is to investigate general techniques that are independent of cluster-level load-balancing, and we therefore focus on *server-level* policies. Even on a single server, a function can have multiple independent and concurrent invocations, and hence containers. Each function has its own container disk-image and initialization code, and thus containers cannot be used by different functions. A function's containers are nearly identical in their initialization overheads and resource utilization since they are typically running the same function code. When a function finishes execution, its container may be terminated, or be kept alive and "warm" for any future invocations of the same function. At any instant of time, each container is either running

a function, or is being kept alive/warm. Thus, server resources are consumed by running containers, and containers being kept alive in anticipation for future invocations.

Keeping functions alive/warm presents a fundamental tradeoff: it can reduce application latency and CPU and I/O overhead, but it increases memory pressure. Nevertheless, recycling the execution environment and keeping function containers alive is a useful performance optimization that is supported by large public cloud platforms [154–156]. In some scenarios, server resources may also be shared with long-running containers and VMs. In such cases, function keep-alive also influences the performance of other co-located applications and services, and the overall cloud efficiency. Therefore, understanding and optimizing this tradeoff is important, and we develop caching-based dynamic resource provisioning policies in Section 3.4. Our goal is to allow FaaS operators to understand the benefits of different levels of aggressive keep-alive policies.

Cold start overheads in OpenWhisk. In order to understand the performance and latency implications of function cold starts, we investigate the chain of events necessary to run function code in a popular FaaS control plane, OpenWhisk [26]. A timeline of a function invocation request for a TensorFlow machine learning inference task is shown in Figure 3.1. The figure shows the major sources of cold start overhead: from request arrival to the actual function execution. OpenWhisk first checks whether the function can be served from the pool of warmed containers it maintains, and if no container is found, a Docker container is launched, and the runtime for the function is initialized: which comprises of OpenWhisk and Python runtime initialization, as well as any specific *explicit* function initialization provided by the application. The total compulsory overhead, from the request arrival to the actual function execution, is significant: up to 2.5 seconds are spent loading all runtime dependencies, before the user-provided initialization and actual event handling code can begin execution.

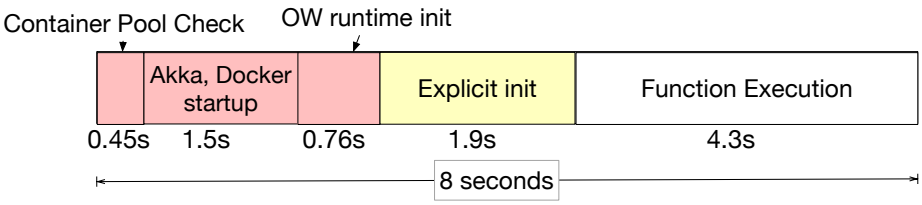


Fig. 3.1: Timeline of function execution and sources of cold start delay in OpenWhisk for an ML inference application.

```
1  #Initialization code
2  import numpy as np
3  import tensorflow as tf
4
5  m = download_model('http://model_serve/img_classify.pb')
6  session = create_tensorflow_graph(m)
7
8  def lambda_handler(event, context):
9      #This is called on every function invocation
10     picture = event['data']
11     prediction_output = run_inference_on_image(picture)
12     return prediction_output
```

Fig. 3.2: Initializing functions by importing and downloading code and data dependencies can reduce function latency by hiding the cold start overhead.

Function Initialization. Function initialization refers to function-specific code for downloading and resolving code and data dependencies, which can be run before actual function execution (explicit-init component in Figure 3.1). For example, this can be used for downloading data dependencies ahead of time such as large neural network models for inference, or for runtime initialization such as downloading and importing package dependencies (e.g., Python packages). An example of function initialization is shown in Figure 3.2, which shows a pseudo-code snippet of a function that performs machine learning inference on its input. For ML inference, the function downloads an ML model and initializes the TensorFlow ML framework (lines 5 and 6). If the function's container is kept alive, then invocations of the function do not need to run the expensive initialization code (lines 2–6).

Workload Diversity and Dynamism. Designing keep-alive policies is not trivial due to

the highly diverse and expanding range of applications that are using FaaS control planes. Conventionally, FaaS has been used for hosting web services, which is attractive because of the pay-per-use properties. Event handling functions for web responses typically have a small memory footprint but require low execution latency. Increasingly, FaaS is also being used for "heavy" workloads with high memory footprint and large initialization overheads such as highly parallel numerical computing (such as matrix operations [114], scientific computing [157], and machine learning [85]. The diversity of FaaS applications also results in a wide range of function memory footprints, running times, and initialization times, as seen in Table 3.2. Keep-alive policies must therefore balance the resource footprint of the containers with the benefits of keeping containers alive—and do so in manner that is applicable across a wide range of applications.

Furthermore, FaaS workloads show a high degree of dynamism and temporal effects. The Azure function [1] trace shows sharp diurnal effects: the function arrival rate is about 2× higher during the peak periods compared to the average. Function workloads are also heavy-tailed: a few "heavy hitting" functions are invoked much more frequently than others or consume a larger amount of computing resources, often by 2 or 3 orders of magnitude.

3.2.1 Policy Goals and Considerations

Because server resources are finite, it is important to prioritize functions which should be kept alive, based on their **initialization time**, **total running time**, **resource footprint**, and **frequencies**. A function which is not popular and is unlikely to be called again in the near future, sees little benefits from keep-alive, and wastes server memory. Similarly, the resource consumption of the functions is also important: since keeping large-footprint functions alive is more expensive than smaller functions, smaller functions should be preferred and kept alive for longer. Finally, functions can also be prioritized based on their initialization overhead, since it is effectively wasted computation.

The problem of designing keep-alive policies is complicated by the fact that functions may have vastly different keep-alive priorities across the different characteristics. Consider a function with a large memory footprint (like those used in ML inference), high initialization overhead, and a low popularity. Such a function should have a low keep-alive priority due to its size, high priority due to large initialization overhead, and a low priority due to its low popularity. Thus, keep-alive policies must carefully balance all the different function characteristics and prioritize them in a coherent manner.

Current FaaS systems have shirked this challenge and use primitive keep-alive policies that are not designed with the diversity and dynamism in mind. FaaS frameworks such as OpenWhisk, keep all functions alive for a *constant* period of time (10 minutes). This is agnostic to different function characteristics such as resource footprint and initialization overheads, and only loosely captures popularity. More principled approaches are needed, which we provide next.

3.3 Caching-based Keep-Alive Policies

Formulating a keep-alive policy that balances priorities based on its competing characteristics (memory footprint, frequency, initialization time, and execution time) of functions seems daunting.

Keeping a function alive reduces its effective execution (or response) latency, in the same way as caching an object reduces its access latency. When all server resources are fully utilized, the problem of which functions *not* to keep alive is equivalent to which objects to *evict* from the cache. The high-level goal in caching is to improve the distribution of object access times, which is analogous to our goal of reducing the effective function latencies.

This caching analogy provides us a framework and tools for understanding the tradeoffs in keep-alive policies, and improving server utilization. Caching has been studied a in

wide range of contexts and many existing caching techniques can be applied and used for function keep-alive. Our insight is that we can use classic observations and results in object caching to formulate equivalent keep-alive policies that can provide us with well-proven and sophisticated starting point for understanding and improving function keep-alive.

In the rest of this section, we will show how cache eviction algorithms can be adapted to keep-alive policies. Caching systems typically seek to improve hit ratios (the fraction of accesses that are cache hits). However, focusing on hit-rates alone does not necessarily translate to improved *system* level performance if the objects have different sizes and miss costs. For instance, caching all small objects may yield a high hit ratio, but the infrequent misses of larger objects results in higher miss costs and poor system throughput. Therefore, we will also focus on minimizing the overall cold start overhead, which is equivalent to the "byte hit ratio" used in caching systems.

3.3.1 Greedy-Dual Keep-Alive Policy

While many caching techniques can be applied to the function keep-alive policies, we now present one such caching-inspired policy that is simple and yet captures all function characteristics and their tradeoffs. Our **GDSF** policy is based on Greedy-Dual-Size-Frequency object caching [145], which was designed for caches with objects of different sizes, such as web-proxies and caches. Classical caching policies such as LRU or LFU do not consider object sizes, and thus cannot be completely mapped to the keep-alive problem where the resource footprint of functions is an important characteristic. As we shall show, the Greedy-Dual approach provides a general framework to design and implement keep-alive policies that are cognizant of the frequency and recency of invocations of different functions, their initialization overheads, and sizes (resource footprints).

Fundamentally, our keep-alive policy is a function *termination* policy, just like caching focuses on eviction policies. Our policy is resource conserving: we keep the functions warm

whenever possible, as long as there are available server resources. This is a departure from current constant time-to-live policies implemented in FaaS frameworks and public clouds, that are *not* resource conserving, and may terminate functions even if resources are available to keep them alive for longer.

Our policy decides which container to terminate if a new container is to be launched and there are insufficient resources available. The total number of containers (warm + running) is constrained by the total server physical resources (CPU and memory). We compute a "priority" for each container based on the cold start overhead and resource footprint, and terminate the container with the lowest priority.

Priority Calculation. The GDSF keep-alive policy is based on Greedy-Dual caching [144], where objects may have different eviction costs. For each container, we assign a *keep-alive priority*, which is computed based on the frequency of function invocation, its running time, and its size:

$$\text{Priority} = \text{Clock} + \frac{\text{Freq} \times \text{Cost}}{\text{Size}} \tag{3.1}$$

On every function invocation, if a warm container for the function is available, it is used, and its frequency and priority are updated. Reusing a warm container is thus a "cache hit", since we do not incur the initialization overhead. When a new container is launched due to insufficient resources, some other containers are terminated based on their priority order—lower priority containers are terminated first. We now explain the intuition behind each parameter in the priority calculation:

Clock is used to capture the recency of execution. We maintain a "logical clock" per server that is updated on every eviction. Each time a container is used, the server clock is assigned to the container and the priority is updated. Thus, containers that are not recently used will have smaller clock values (and hence priorities), and will be terminated before more recently used containers.

Containers are terminated only if there are insufficient resources to launch a new container and if existing warm containers cannot be used. Specifically, if a container j is terminated (because it has the lowest priority), then Clock = Priority$_j$. All subsequent uses of other, non-terminated containers then use this clock value for their priority calculation. In some cases, *multiple* containers may need to be terminated to make room for new containers. If E is the set of these terminated containers, then Clock = $\max_{j \in E}$ Priority(j)

We note that the priority computation is on a per-container basis, and containers of the same function share some of the attributes (such as size, frequency, and cost). However, the clock attribute is updated for each container individually. This allows us to evict the oldest and least recently used container for a given function, in order to break ties.

Frequency is the number of times a given function is invoked. A given function can be executed by multiple containers, and frequency denotes the *total* number of function invocations across all of its containers. The frequency is set to zero when all the containers of a function are terminated. The priority is proportional to the frequency, and thus more frequently executed functions are kept alive for longer.

Cost represents the termination-cost, which is equal to the total initialization time. This captures the benefit of keeping a container alive and the cost of a cold start. The priority is thus proportional to the initialization overhead of the function.

Size is the resource footprint of the container. The priority is inversely proportional to the size, and thus larger containers are terminated before smaller ones. In most scenarios, the number of containers that can run is limited by the physical memory availability, since CPUs can be multiplexed easily, and memory swapping can result in severe performance degradation. Thus, for ease of exposition and practicality, we consider only the container *memory* use as the size, instead of a multi-dimensional vector.

We can also use multi-dimensional resource vectors to represent the size, in which

case we convert them to scalar representations by using the existing formulations from multi-dimensional *bin-packing*. For instance, if the container size is $\mathbf{d}$, then the size can be represented by the magnitude of the vector $||\mathbf{d}||$. Other size representations can also be used. A common technique is to normalize the container size by the physical server's total resources ($\mathbf{a}$), and then compute the size as $\sum_j \frac{d_j}{a_j}$ where d_j, a_j are the container size and total resources of a given type (either CPU, memory, I/O) respectively. Cosine similarity between $\mathbf{d}$ and $\mathbf{a}$ can also be used, as is widely used in multi-dimensional bin-packing.

FaaS-specific considerations. The application of cache eviction algorithms to FaaS keep-alive is fairly straight-forward. The various inputs Greedy-Dual (memory size, cold start time, frequency) are available once a function has finished execution, and thus the keep-alive policy is completely online. Our policy calculates eviction priorities at the function level, but evicts at the container level. Recall that a particular function may have multiple containers associated with concurrent function invocations. We assume that all containers of a function are identical, i.e., they have the same initialization cost, footprint, etc. Thus, any one of the identical containers can be evicted.

3.3.2 Other Caching-Based Policies

The Greedy-Dual approach also permits many specialized and simpler policies. For instance, allowing for different parameters in Equation 3.1 results in different caching algorithms. If only the access clock is used as a priority, and other parameters are ignored, then we get **LRU**, with its ease of analysis and generality which has been well established with over half a century of empirical and analytical work. Using only frequency yields **LFU.** Similarly, a size aware keep-alive policy can be obtained by using 1/size as the priority, which would be useful in scenarios where memory size is at a premium.

Other size-aware online algorithms with tight online theoretical guarantees can also be applied. We also implement the **LANDLORD** [158] algorithm, which can be understood

as a variant of the Greedy-Dual approach. Landlord also considers the frequency, size, and initialization cost of functions. When the server is full and some container is to be evicted, a "rent" is charged from each function based on its size and initialization cost (specifically, it is equal to $\min(\frac{\text{initialization cost}}{\text{size}})$. This subtly differs from Greedy-Dual-Size-Frequency: the decrease in priority is computed based on the state of all the cached containers, and not independently applied. Upon a function invocation, its containers get a "credit", and their priority is set to their initialization cost. The containers with the lowest credits are evicted. Landlord has appealing and well-proven properties of its online performance: its competitive ratio (the performance compared to an optimal *offline* algorithm that knows future requests) has been well analyzed [158].

3.4 Server Provisioning Policies

Resource provisioning, i.e., determining the size and capacity of the servers for handling FaaS workloads, is a fundamental problem in serverless computing. In this section, we develop techniques that allocate the appropriate amount of resources to servers based on the characteristics of the function workloads. Resource provisioning policies must consider the rate of function invocations, the resource footprints of the functions, and the inter-arrival time between function invocations. To handle the interplay and tradeoffs between these factors, we use similar principles for provisioning that we used for developing our keep-alive policies. In case FaaS workloads are co-located with other applications such as long-running containers and VMs, our provisioning policies can also be used to determine the resource allocation of the combined running and warm function pool.

The fundamental challenge underlying resource provisioning for FaaS workloads is the performance vs. resource allocation tradeoff. Running a workload on large servers/VMs provides more resources for the keep-alive cache, which reduces the cold starts and improves the application performance. However, we must also be careful to not *overprovision*, since

it leads to wasted and underutilized resources. Additionally, since function workload can be dynamic, resource provisioning must be *elastic*, and be able to dynamically scale up or down based on the load. We therefore present a *static* provisioning policy that determines the server memory size for a given function workload, and then develop an elastic-scaling approach for handling workload temporal dynamics.

3.4.1 Static Provisioning

In Section 3.1.1, we have seen how keeping function containers warm in a keep-alive cache can help mitigate the cold start overheads. The effectiveness of any keep-alive policy depends on the size of this keep-alive cache, and thus the server resources available, i.e., the server size. Our *static* provisioning policy thus selects a server size for handling a given workload. We want to optimize the resource provisioning to avoid over and under provisioning, both of which are detrimental to cost and performance respectively.

Having established that keep-alive policies are equivalent to cache eviction in the previous section, we now extend the use of the caching analogy further, to develop a caching-based provisioning approach. We claim that the performance vs. resource availability tradeoff of serverless functions can be understood and modeled using cache hit (or miss) ratio curves. Hit-ratio curves are widely used in cache provisioning and modeling, since they give insights into cache performance at different sizes. Once a hit-ratio curve is obtained, it is used to provision the cache size based on system requirements. A common approach is to size the cache based on a target hit-ratio (say, 90%). Alternatively, the slope of a hit-ratio curve can be understood to be the marginal utility of the cache, and a cache size that maximizes this marginal utility is picked. This entails choosing a cache size which corresponds to the *inflection point* of the hit-ratio curve.

Hit-ratio Curve Construction. We use a function hit-ratio curve for determining the percentage of warm-starts at different server memory sizes. The hit-ratio curve is constructed

by using the notion of *re-use distances.* A function's reuse-distance is defined as the total (memory) size of the unique functions invoked between successive invocations of the same function. For example, in the request *reuse* sequence of `ABCBCA`, the reuse distance of function `A` is equal to `size(B) + size(C)`. The *distribution* of these reuse distances can yield important insights into the required cache size. If the cache size is greater than the reuse distances, then there will be no cache misses. This can be generalized to find the hit-ratio at cache size c:

$$\text{Hit-ratio}(c) = \sum_{x=0}^{c} P(\text{Reuse-distance} = x), \tag{3.2}$$

where the reuse distance probability is obtained by scanning the entire input function workload for all reuse sequences. Conveniently, the hit-ratio is the CDF (cumulative distribution function) of the reuse distances, which can be empirically determined based on all the computed reuse distances. We show one such hit-ratio curve constructed with reuse distances, for a representative sample of the Azure function workload in Figure 3.3. We can see that the hit-ratio curve of functions *also* follows the classic long-tailed behavior: the hit-ratio steeply increases with cache size up to an inflection point, after which we see diminishing returns.

This technique and observation informs our provisioning policy. We construct a hit-ratio curve based on reuse distances, and size the server's memory based on the inflection point. Alternatively, we can set a target hit ratio (say, 90%), and use that to determine the minimum memory size of the server. Finding the reuse-distances for an entire trace can be an expensive, one-time operation, and takes $O(N * M)$ time where N is the number of invocations and M is the number of unique functions. However, sampling techniques such as SHARDS [152] can be applied to drastically reduce the overhead, making this a practical and principled technique for resource provisioning.

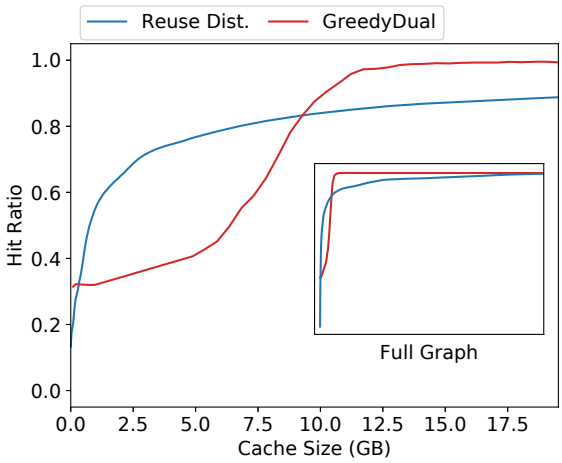


Fig. 3.3: Hit ratio curve using reuse distances show slight deviations from the observed hit ratios due to dropped requests at lower sizes, and concurrent executions at higher sizes.

Limitations of the Caching Analogy. The error in hit-ratios with the reuse-distance approach in Figure 3.3 highlights an important facet where caching does not fully map to FaaS. The main difference is due to the limitations on the concurrent execution of functions: caching deals with unique objects, whereas there can be multiple containers for a function. At lower cache sizes, a high miss rate results in higher server load, and hence a higher number of dropped requests, that the classical reuse-distance approaches do not capture. If all warmed containers of a function are in use, then a new invocation results in a cold start—which would be counted as a cache "hit". Thus, at lower sizes, the real hit-ratio is lower than the ideal. At larger sizes, *multiple* containers corresponding to concurrent invocations of a function will be present, which results in a deviation from the hit-rate curve. Reconciling these differences is an interesting area of future work. However, we note that hit-ratio curves are only used for coarse-grained allocation, and small deviations result in slight under or over provisioning. Moreover, our dynamic allocation policy described next can reduce these errors using proportional control.

3.4.2 Elastic Dynamic Scaling

We also use the hit-ratio curve approach for a *dynamic* auto-scaling policy that adjusts the server size based on workload requirements. We assume that the FaaS server backend is running functions as containers either inside a virtual machine (VM), or is sharing the physical server with other cloud applications. In either case, it is important to be able to reclaim unused keep-alive cache resources and reduce its footprint, in order to increase the efficiency of the cloud platform.

Our vertical elastic scaling policy is simple and is intended to demonstrate the efficacy of a general caching-based approach. We implement a proportional controller [159] which periodically adjusts the VM memory size based on the rate of cold starts. Thus, during periods of low rate of function invocations (i.e., arrival rate), the cache size can be reduced. This may *increase* the miss-ratio—but we care about the cold starts (i.e., misses) per second, which is product of miss-ratio and invocations per second. Our controller monitors the arrival and cold start rate, and uses the hit-ratio curve to decrease or increase VM size dynamically. We use VM resource deflation [160] to shrink or expand the VM by using a combination of hypervisor level page swapping, or guest-OS memory hot-plug and unplug.

Assume that we have a target miss speed (number of cold starts/misses per second). For instance, this target value can be a product of the desired hit-ratio, h, and the average function arrival rate for the entire workload trace, $\bar{\lambda}$. Periodically, we monitor the exponentially smoothed arrival rate λ, and the observed miss speed. Our proportional controller adjusts the cache size in order to reduce the difference between the actual vs. target miss speed. This error is used to compute the new *miss rate*, m, and the associated cache size c' as follows:

$$\mathrm{HR}(c') = 1 - m = 1 - h\frac{\bar{\lambda}}{\lambda} \tag{3.3}$$

The new cache size c' is then determined by inverting the hit-rate function HR. Our vertical scaling controller is designed for coarse-grained VM size adjustments, and only tracks the workload at time granularities of several minutes. Our intent with this policy is to not be overly aggressive with the capacity changes, but only to capture the coarse diurnal effects. Therefore, we use a large error deadband: the cache size is only updated if the error is more than 30%. Finally, the memory scaling can also be combined with cpu auto-scaling based on the function arrival rate, using classical predictive and reactive auto-scaling techniques found in web-clusters [161].

Online adjustments. Our policies rely on the *aggregate* function characteristics, which is used for constructing the hit-ratio curve. Once done, the traffic intensity (invocations per second) can change. We primarily assume that the probability distribution of function characteristics such as their frequency and size, does not significantly change. However, our dynamic scaling policy can adjust to changes in the traffic intensity (invocations per second). In other words, we assume that the future traffic is going to be similar to the past, which is the basis of the timeseries-forecasting based policies (such as in [1]), and is the fundamental principle underlying caching in general. Our provisioning policies are not completely online, since they have a preparation phase for constructing the hit-rate curves. A "drift" in function characteristics is fixed by periodically updating the hit-ratio curve, which we currently do once per week. Online hit-ratio curves can also be constructed, and adapting techniques such as [162] is part of our future work.

3.5 Implementation

We have implemented the keep-alive and the provisioning policies as part of our FaasCache framework built on top of OpenWhisk (Figure 3.4).

Keep-Alive. FaasCache replaces the default OpenWhisk TTL-based keep-alive policy with

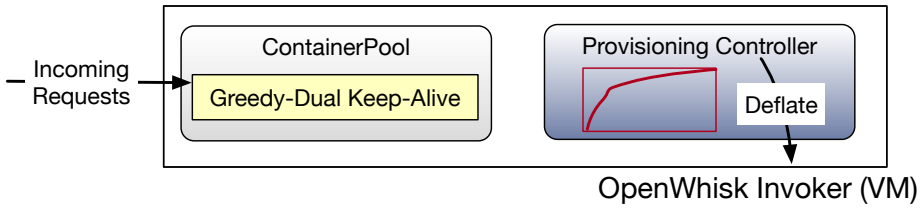


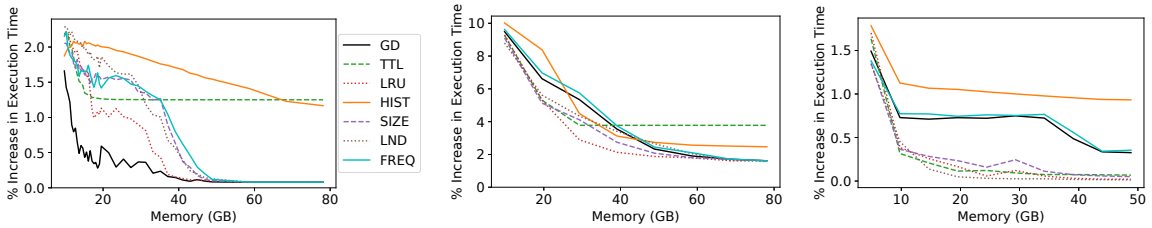
Fig. 3.4: FaasCache system components. We build on OpenWhisk and augment it with new keep-alive policies and a provisioning controller.

the Greedy-Dual-Size-Frequency approach. For each initialized container, we assign and maintain the keep-alive prioritized ContainerPool, which is only a 100-line Scala modification. Each invocation of a function (OpenWhisk action) in ContainerPool records the launch time and when results are returned.

If the container was prewarmed before the invocation arrived, we record it as the function's warm runtime. For new functions, the initialization overhead is captured and assumed to be the worst-case runtime until a warmed invocation is recorded. In the subsequent invocations, the initialization overhead is computed by subtracting the cold from the warm time. The function's frequency and clock value are updated with each request. If the last container of a function is evicted, its cold and warm runtimes are stored and used to compute priority for its future invocations. To preserve the invocation fast-path, the ContainerPool is not kept sorted by priority. Instead, it is sorted by priorities only during evictions, when the lowest priority container(s) are terminated. We batch eviction operations to optimize the slow-path: we evict multiple containers to reach a certain free resource threshold (1000 MB is the current default).

In the future, we intend to implement a similar design that is found in the Linux kernel page eviction. A separate thread (analogous to kswapd) can be used to periodically sort the containerpool list and asynchronously evict containers, so that eviction is not on the critical path.

Provisioning. For the static provisioning, we compute the reuse distance distribution for a



(a) Representative functions. (b) Rare functions. (c) Random sampling.

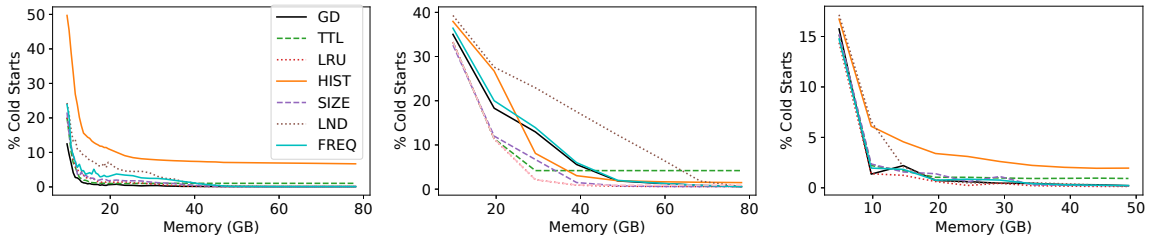
Fig. 3.5: Increase in execution time due to cold starts for different workloads derived from the Azure function trace.

given workload trace, and assume stationarity — that it will be applicable on similar future workloads. We compute the reuse distances conventionally, by examining all reuse-sequences. The dynamic provisioning controller runs periodically (every 10 minutes), to deflate or inflate the VM size, if the cold start rate deviates from the target significantly (by more than 30%). When the VM has to be shrunk, we use cascade deflation [160]. We shrink the ContainerPool first, and reclaim the free memory using guest OS-level memory hot-unplug and hypervisor-level page swapping.

Keep-alive Simulator. We have implemented a trace-driven discrete event simulator for implementing and validating different keep-alive policies. Our simulator is written in Python in about 2,000 lines of code, and implements the various variants described in Section 3.3.2. It allows us to determine the cache hit ratios and the cold start overheads for different workloads and memory sizes. Additionally, it also implements the static and dynamic provisioning policies for adjusting server size.

3.6 Experimental Evaluation

We now present the experimental evaluation of our caching-based keep-alive technique by using function workload traces and serverless benchmarks. Our goal is to investigate the effectiveness of these techniques under different workload and system conditions.



(a) Representative functions. (b) Rare functions. (c) Random sampling.

Fig. 3.6: Fraction of cold starts is lower with caching-based keep-alive.

Setup, Workloads, and Metrics. For evaluating different keep-alive performance with different workload types, we use different trace samples from the Azure Function trace [1], which contains execution times, memory sizes, and invocation-timestamps for more than 50,000 unique functions. Since our goal is to examine performance at a *server* level, we use smaller samples of this trace for realistic server sizes, and replay them in our discrete-event keep-alive simulator. This also allows us to examine the behavior with different *types* of workloads, which is important because our keep-alive policies are designed to be general and workload-agnostic. We use the following three trace samples (more details in the Table 3.1):

RARE: A random sample of 1000 of the rarest, most infrequently invoked functions. These functions will usually result in cold starts under a classic 10 minute TTL.

REPRESENTATIVE: A sample of 400 functions, sampled from each quartile of the dataset based on frequency—yielding a more representative sample with higher function diversity.

RANDOM: A random sample of 200 functions.

Functions from the FunctionBench [163] suite are used for generating a realistic workload. A single server with 250 GB RAM and 48-core Intel Xeon Platinum 2.10 GHz CPUs is used for running all functions. The server is running modified OpenWhisk (i.e., FaasCache), and Ubuntu 16.04.5.

Tab. 3.1: Size and inter-arrival time (IAT) details for the Azure Function workloads used in our evaluation.

Trace	Num Invocations	Reqs per sec	Avg. IAT
Representative	1,348,162	190 /s	5.4 ms
Rare	202,121	30 /s	36 ms
Random	4,291,250	600 /s	1.8 ms

Adapting the Azure Functions Trace. The format of the original Azure Function trace [1] requires some additional pre-processing and extrapolation for generating a workload. The full dataset consists of 14 days of function invocations, and billions of individual invocations. We use the first day's data, and do not consider functions that are never reused (i.e., with less than two invocations).

The original trace provides memory consumption at the *application* level—with the application made up of multiple functions. Therefore, we evenly split the memory allocation between all functions in an application. The dataset provides invocations in minute-wide buckets. When injecting/replaying the workload, if there is only one invocation in a minute-bucket, it is injected at the beginning of the minute. For multiple invocations, they are equally spaced throughout the minute.

The cold start overhead of each function is estimated as `maximum - average` runtime, and the execution times provided in the dataset are used for this computation. The dataset does not account for certain important sources of cold start overheads such as execution environment creation (e.g., Docker). This unfortunately underestimates the cold start overheads. However, because it applies uniformly to all functions, it preserves the relative performance of the different keep-alive policies, and does not affect the cache hit ratios.

We are interested in two metrics: the cold start ratio; and the average increase in the execution time due to cold starts. The increase in execution time is computed by averaging across all function invocations.

3.6.1 Trace-Driven Keep-Alive Evaluation

In this subsection, we use the Azure function traces to evaluate different keep-alive policies in our discrete-event simulator. We compare all caching-based variants against the default keep-alive policy in OpenWhisk (10 minute TTL). When the server is full, this TTL policy evicts containers in an LRU order. We also evaluate different Greedy-Dual variants: GD is our GDSF policy described in Section 3.3.1. The others are the caching-based variants described in Section 3.3.2: LND is Landlord, and FREQ is LFU.

We also compare against the histogram-based keep-alive policy in [1], which is the state of the art technique. We have reproduced this policy (HIST) from the details in the paper, and have implemented it in a "best-effort" manner without any knowledge of the optimizations in the actual implementation. This is effectively a "TTL+Prefetching" policy: it uses a histogram of *inter-arrival times* to predict future function invocations and eagerly evict warm functions. It uses timeseries forecasting to capture temporal locality, but does not consider the other function characteristics such as function size and initialization cost. The IAT, computed by taking a function's execution time plus the subsequent idle time, between each actual invocation is recorded in minute granularity buckets, tracking up to four hours between executions. The policy uses ARIMA modeling for those invocations that fall outside this four hour window, we chose not to implement this specific feature due to its complexity, and the fact that it accounted for a minor fraction (˜0.56%) of all invocations. From these buckets, a function's coefficient of variation (CoV) is calculated using Welford's online algorithm [164]. When the function's IAT is predictable (CoV ≤ 2), the function's historical/customized preload and TTL time are used. Otherwise, the function has a generic TTL of two hours. When an invocation is anticipated, it is brought into memory and kept there until its TTL expires. A function is evicted when the policy predicts it will not have an invocation in the near future.

The increase in execution time for different traces and for different cache sizes is shown

in Figure 3.5. The increase in execution time is the cold start overheads averaged across all invocations of every function, and captures the user-visible response-time.

For the representative trace (Figure 3.5a), Greedy-Dual reduces the cold start overhead by more than 3× compared to TTL for a wide range of cache sizes (15–80 GB). Interestingly, it is able to achieve a low overhead of only 0.5% at a much smaller cache size of 15GB, compared to other variants, which need 50 GB to achieve similar results—a reduction of cache size by more than 3×. For rare functions (Figure 3.5b), caching-based approaches such as LRU reduce the cold start overhead by 2× compared to TTL for cache sizes of 40–50 GB. This shows that for rare functions, recency is a more pertinent characteristic, and the complex four-way tradeoff used in Greedy-Dual is not necessarily ideal in all workload scenarios. For this workload, the HIST policy outperforms TTL, as reported in [1]. However, it results in 50% higher cold start overhead compared to caching-based approaches. Furthermore, because HIST uses only inter-arrival times, it is unable to perform well with heterogeneous representative workloads (Figure 3.5a).

Finally, the randomly sampled trace has a large number of infrequent functions because of the low probability of selecting the heavy-hitting functions. In Figure 3.5c, the recency component again dominates, and we see LRU outperforming other variants. The equivalence of LRU and TTL-based caching for rare objects has been noted [165, 166], which explains their similar behavior seen in Figure 3.5c.

Result: *For representative, diverse workloads, our GD policy can improve the performance and shrink cache sizes by up to* 3×*. For more homogeneous workloads, LRU can outperform current TTL-based approaches by* 2×*.*

We can observe from Figure 3.5 that the increase in execution time is generally small ($< 10\%$). This is because of two main factors: the evaluation metric chosen, and the properties of the workload trace. The execution time is averaged across *all* function invocations. However, serverless workloads consist of a large number of very frequently

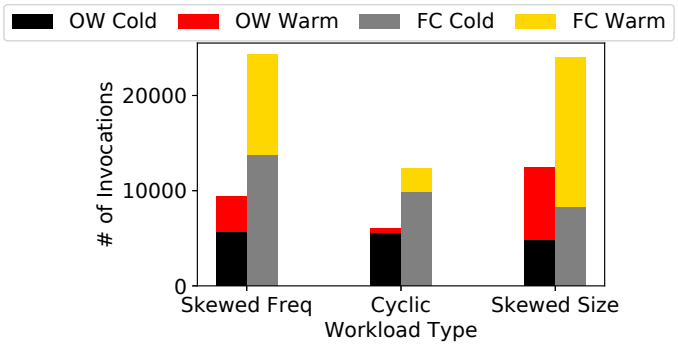


Fig. 3.7: FaasCache runs 50 to 100% more cold and warm functions, for skewed workload traces.

invoked functions. The performance of these functions is generally not affected by keep-alive policies, since any policy is going to keep them in the cache because of their high frequency. Thus, the difference between non-work-conserving policies such as TTL and Greedy-Dual is masked because of the frequent and popular functions. For instance, the average inter-arrival time for all three workloads is less than 36ms, or about 27 function invocations per second. Thus, the server is overloaded, and TTL does well even though it is not work-conserving. As the IAT grows, the effectiveness of work-conserving caching-based approaches increases compared to TTL, as we shall see in the next subsection.

We see a similar relation and behavior in the miss-ratio curves shown in Figure 3.6. Due to function heterogeneity, the cold start overheads are not strictly correlated with cache miss ratios, and thus the differences between policies is different compared to the previously described actual cold start overheads. Classic miss-ratio curves do not consider the miss *cost* (i.e., initialization cost), which is an important metric that is optimized by the Greedy-Dual approach. Thus, in general, even in object caching contexts, miss-ratio curves deviate from the actual performance—a behavior that we also observe.

Tab. 3.2: FaaS workloads are highly diverse in their resource requirements and running times. The initialization time can be significant and is the cause of the cold start overheads, and depends on the size of code and data dependencies.

Application	Mem size	Run time	Init. time
ML Inference (CNN)	512 MB	6.5 s	4.5 s
Video Encoding	500 MB	56 s	3 s
Matrix Multiply	256 MB	2.5 s	2.2 s
Disk-bench (`dd`)	256 MB	2.2 s	1.8 s
Image Manip	300 MB	9 s	6 s
Web-serving	64 MB	2.4 s	2 s
Floating Point	128 MB	2 s	1.7 s

3.6.2 OpenWhisk Evaluation

In this subsection, we evaluate the performance of the FaasCache system on real functions. We focus on the performance of FaasCache's Greedy-Dual keep-alive implementation, and compare it to the vanilla OpenWhisk system which uses a 10 minute TTL.

In contrast to the previous subsection in which we showed the average performance for different cache sizes, we will now also focus on the inverse problem: for a fixed server size, how much more load can be handled with FaasCache? By leveraging Greedy-Dual caching, FaasCache is able to reduce cold starts. This also reduces the number of *dropped* requests.

OpenWhisk buffers and eventually drops requests if it cannot fulfill them. Because FaasCache more effectively selects evictions, its higher hit rate results in functions finishing faster, allowing more functions to be executed in the same time frame.

To examine the effect of Greedy-Dual keep-alive on cold start and dropped requests, we use a workload trace comprising of four different functions: Disk-bench, ML inference, Web-serving, and Floating-point, described in Table 3.2.

In Figure 3.7, we use different kinds of *skewed* workloads: with a single function having a different frequency, a cyclic access pattern, and a skewed workload with 2 sizes. We see that FaasCache's keep-alive can increase the number of warm invocations by between 50 to

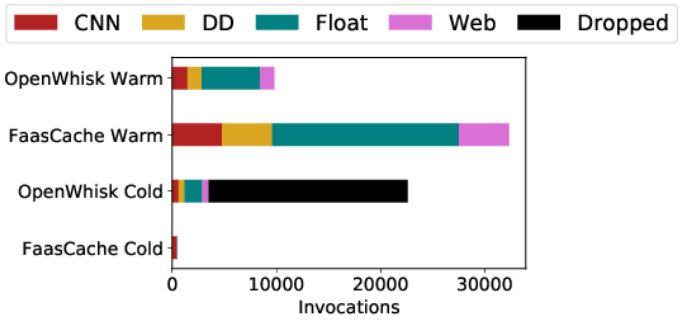


Fig. 3.8: FaasCache increases warm-starts by more than 2×, which also reduces system load and dropped functions.

100% compared to OpenWhisk's TTL. The difference in the total number of requests served (warm+cold) is because OpenWhisk drops a significant number of requests due to its high cold start overhead and resultant system load. Thus with FaasCache, the total number of requests that are served also increases by 2×.

Next, we use the skewed frequency workload and use functions from Table 3.2 to evaluate the impact on real applications. To generate the workload, the CNN, DD, and Web-serving functions have an inter-arrival time of 1500 ms, and the Floating-point function has a lower IAT of 400 ms. Figure 3.8 shows the breakdown of different function invocations for this workload on a 48 GB server. Interestingly, OpenWhisk drops a significant number (50%) of requests due to the its high cold start overheads. FaasCache increases the warm requests by more than 2×. Interestingly, the *distribution* of warm starts is also different. FaasCache's Greedy-Dual policy prioritizes functions with higher initialization times, but penalizes those with large memory footprints. Because the floating-point function has a high initialization overhead (Table 3.2), it sees a 3× increase in hit-ratio compared to OpenWhisk. *In practical terms, the improvement in keep-alive results in a* 6× *reduction in the application latency.*

Result: *FaasCache can increase the number of warm-starts by* 2× *to* 3× *depending on the function initialization overheads and workload skew. This results in lower system load, which increases the number of requests FaasCache can serve by* 2×*.*

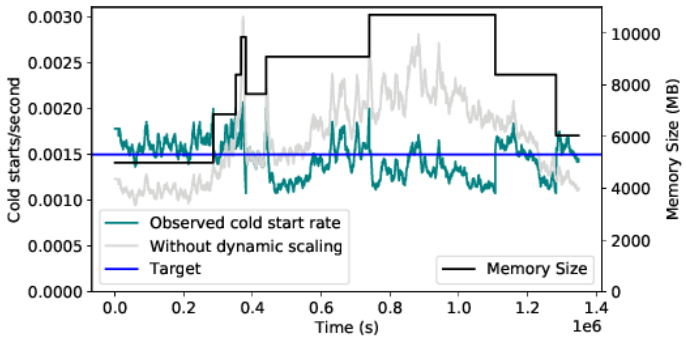


Fig. 3.9: With dynamic cache size adjustment, the cold starts per second are kept close to the target (horizontal line), which reduces the average server size by 30%.

3.6.3 Effectiveness of Provisioning Policies

All our previous results have been with a statically allocated server, and we now illustrate the effectiveness of our dynamic vertical scaling policy described in Section 3.4.2. The goal is to dynamically adjust the cache size based on the workload. Our policy seeks to keep the miss speed (cold starts per second) close to a pre-specified target. This is shown in Figure 3.9—the target is 0.0015 misses per second. In this experiment, the cache resizing is done only when the miss speed error exceeds 30%, and we can see that the cache size increases with the miss speed, and decreases with it. Without the dynamic scaling, a conservative provisioning policy would result in a constant, 10,000 MB size. In contrast, the average cache size with our proportional controller is less than 7,000 MB. This 30% reduction means that FaaS providers can reduce their provisioned resources without compromising on performance. The freed-up resources can be used to accommodate additional cloud workloads (such as co-located VMs and containers). Our dynamic scaling is extremely conservative: increasing its agressiveness by reducing the error tolerance below 30% will reduce average server size, but we seek to avoid the resultant small changes to memory-size to minimize fragmentation.

4. Load- and Locality-Aware Load Balancing

In this chapter, we investigate load-balancing policies for serverless clusters and find that the locality vs. load tradeoff is crucial and presents a large design space. Locality, i.e., running repeated invocations of a function on the same server, is a key determinant of performance because it increases warm-starts and reduces cold start overheads.

We enhance consistent hashing for FaaS, and develop CH-RLU: Consistent Hashing with Random Load Updates, a simple practical load-balancing policy which provides more than 2× reduction in function latency. Our policy deals with highly heterogeneous, skewed, and bursty function workloads, and is a drop-in replacement for OpenWhisk's existing load-balancer. We leverage techniques from caching such as SHARDS for popularity detection, and develop a new approach that places functions based on a tradeoff between locality, load, and randomness.

4.1 Related Work

Package-aware load balancing [94] identifies and uses function code dependencies (software packages) as an important source of data locality. While this is an important factor, we focus on in-memory locality of kept-alive functions, since memory capacity is much smaller than permanent storage and caching functions in memory has a very large performance impact. CPU contention and interference is a major source of performance bottlenecks for co-located functions, and adjusting CPU-shares using cgroups can provide significant

benefits [102–104]. The load-locality tradeoff we explore is complementary to these CPU scheduling optimizations. The repetitive nature of functions and their workflows can also be used to improve resource utilization and latency [105–108]: our load-balancer is stateless for the sake of simplicity and can be enhanced with these techniques if necessary.

The tradeoff between locality and performance has also been explored in the context of delay scheduling [109] for data-parallel applications like MapReduce. Load-balancing is seen as a "dispatch" problem in queuing theory, and the FaaS cluster system most closely approximates G/G/PS, since the arrivals and service times are not markovian. Techniques such as "join the shortest queue", and "least work left" [110] have been shown to be effective. The online-greedy policy evaluated in the previous section closely approximates least-work-left. However, it is difficult to implement in practice since the running times of functions is hard to predict due to their volatile arrival distribution mixtures and high variances in running time due to various system interference effects.

4.2 Background: Load Balancing

Managing the load of a cluster of servers is a common problem in distributed computing systems. Load-balancing policies typically rely on some notion of "load" of a server, such as the number of concurrently executing tasks, length of the task-queue, cpu-utilization, etc. The first broad class is *compute-oriented* load-balancers, typically used for short-running tasks and queries. Load-balancing for computational tasks is common in scenarios like web-clusters [167]. In these systems, the tasks can be executed on any server, servers in a cluster are largely fungible, and the task performance largely depends on the server-specific cpu-utilization at the time.

Load-balancing techniques have received significant theoretical attention (especially using queuing theory), as well as many practical systems [168]. From a queuing theory perspective,

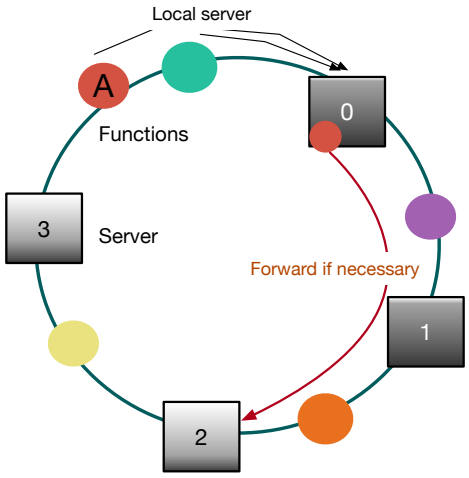


Fig. 4.1: Consistent hashing runs functions on the nearest clockwise server. Functions are forwarded along the ring if the server is overloaded.

policies such as least-work-left (LWL) and Join-Shortest-Queue (JSQ), have studied near-optimal load balancing for computing load-dependent workloads under a processor-sharing (PS) setting.

Interestingly, load-balancing for *data-oriented* systems, such as Content Delivery Networks (CDNs) [169], and distributed key-value stores (such as Amazon Dynamo [168]) must also balance the load on servers, but with data locality as a key requirement. In this context, locality refers to requests for the same object being handled by the server, or the same subset of servers if the object is replicated. We find that FaaS load balancing requires and benefits from *both* these objectives: minimizing computing load *and* maximizing locality to reduce cold starts.

4.2.1 Consistent Hashing

For data-oriented systems, a common technique to ensure locality is Consistent Hashing [70, 167]. Objects are mapped to servers based on some object id or key. Consistent hashing preserves object-server mapping even in the face of server additions and removals,

which improves locality. Figure 4.1 provides an overview of consistent hashing. Both objects and servers are hashed to points on a "ring", and objects are assigned to the next server (in the clockwise direction) in the ring. Addition or removal of servers only affects the nearby objects by remapping them to the new next server in the ring.

OpenWhisk uses a modified consistent hashing algorithm for its default load balancer. As functions are sent to servers, their expected memory footprint is added to a server-specific running counter of outstanding requests. Upon completion the memory size of a function is decremented from that server's counter. If the counter for a function's "home" server would exceed the assigned memory on the server it is forwarded along the ring. The drawback of this policy, and consistent hashing as a whole, is that the performance can be affected by the relative popularities of the different objects. A highly popular object can result in its associated server getting overloaded. This problem is exacerbated in the case of FaaS functions, as we shall demonstrate in the next section.

4.3 Load-aware Consistent-Hashing

In this section, we describe the load-balancing algorithm which is locality, stale-load, and burst aware. We assume a cluster homogeneous servers, and a new function invocation can be sent to any of the servers. Each server implements keep-alive for functions: after successful execution, the function's container is stored in server memory, and evicted based on some eviction policy.

4.3.1 Tradeoff between Locality and Load

We use consistent hashing as the fundamental principle to ensure high locality: repeated invocations of the same function occur on the same server. However, popular functions, i.e., which are invoked very frequently, can result in overloaded servers. Because function

performance is affected by server load and resource availability, focusing on locality alone can result in slow function execution.

Function popularities are also highly skewed: a small percentage account for a vast majority of invocations. With pure locality-based load-balancing, the servers of these popular functions would be severely overloaded. Functions also can run for significantly longer than simple web requests, and thus they impose more load on servers, and the cost of a wrong placement decision is higher. This, combined with bursty invocations, can significantly increase the tail latency of functions. Thus pure-locality policies such as classical consistent hashing are not sufficient, and our research question is: *Can consistent hashing be used to reduce latency due to overloaded servers?* Or put another way, can we balance the tradeoff between function locality and server-loads with consistent hashing?

Our key idea is to extend consistent hashing to take also into account server loads, the cold start overheads of different functions, and the bursty traffic that is a key characteristic of FaaS workloads. In the rest of this section, we describe our approach.

4.3.2 Key Principle: Load-based Forwarding

To balance the locality vs. server load tradeoff, we build on a new variant of consistent hashing called Consistent Hashing with Bounded Loads [170] (abbreviated as CH-BL in the rest of the paper). The key idea behind CH-BL is to use consistent hashing to locate servers for objects, and if the servers are "full", then "forward" the objects to the next server in the consistent hashing ring.

For example, in Figure 4.1, function A is originally assigned to server 0, but this "home" server is overloaded (already running many functions), and thus the function is forwarded along the ring until a suitable non-overloaded server (2) is found. Any 5-independent hashing function can be used for determining the "home" server of a function. Users can specify

the load upperbound or the capacity of the server (b), which determines the max load the server can sustain. Consistent hashing with bounded loads provides many strong theoretical guarantees on the length of the forwarding chain until the object is safely placed on a server.

Interestingly, forwarding along the ring not only avoids server overloads, but also improves locality, *even in overload scenarios.* Forwarding along the ring has the advantage that even if function is not run on its "home" server, subsequent invocations that "overflow" still have a high warm-start probability on the servers on the overflow chain. The warm-start probability is highest on the home server, and decays the farther the function is from its home server. This is more beneficial than alternative techniques such as Consistent Hashing with Random Jumps [171], which do not preserve locality and instead forwards to randomly chosen least loaded servers.

4.3.3 Server Load Information

Server load is a key metric in load-balancing policies. We need to be able to determine the *relative* suitability of one server over another, and thus many existing metrics can be used to provide information about server loads. Simple metrics such as number of running functions are insufficient, since functions can have highly variable execution times. OpenWhisk currently uses occupied-memory used by active/running invocations as a proxy for load, and is unsuitable for the same reason. Both these metrics fail to capture CPU loads and lead to scalability issues when used by the load-balancer.

Instead, we primarily rely on *system-level* load metrics, such as the standard Linux 1-minute load-average. In addition to CPU utilization, this also captures the I/O wait due to cold starts, and provides a more realistic measure of load. Traditional Linux load-average estimates the total number of processes running and ready-to-run, and we normalize the load-average by the number of CPUs. Thus, a load-average of 8 on an 8 core server (discounting hyperthreading) is normalized to 1.

An important practical consideration is that load information is often *stale*, with the degree of staleness ranging from a few seconds to several minutes. For instance, because the Linux load average is an exponential moving average, it is slow to change. Furthermore, load monitoring and reporting has delays due to how frequently the metrics are gathered at the local server, and how often they are made available to the load-balancer. We use a simple publish-subscribe-like system, where individual servers periodically (every 5 seconds) push their load information, and the load-balancer uses these published loads to make all scheduling decisions.

4.3.4 Why CH-BL Is Insufficient

The high computing load of functions, their bursty nature, and the staleness of loads, are the three major challenges to Consistent Hashing with Bounded Loads [170] that the original algorithm is not designed to meet. There are a few practical considerations and key differences between simple object/storage caching and function execution: 1. CH-BL does not take into account the heterogeneity in running times and memory size of the objects (i.e., functions). 2. The implicit CH-BL performance model is binary: running-time is assumed to be uniform as long as servers are under the load-bound. 3. The server loads evolve as a result of the actual function execution and are not just uniformly incremented as in the original algorithm. Object deletions are also not handled explicitly: we let the lazily computed load average determine whether a server meets the load-bound or not.

Importantly, we do not assume complete and consistent state information about the servers. Omniscient knowledge of the execution state of all functions running all servers can certainly be leveraged effectively to run functions on the most suitable server. However, such maintaining such global knowledge is expensive and impractical as far as storage consistency and latency are concerned. Thus, we are striving for load-balancing policies which are robust to stale, incomplete, and coarse-grained information about server states. In the rest of this

section, we shall show how the above three limitations of CH-BL can be overcome in FaaS load-balancing settings.

4.3.5 Incorporating Function Performance Characteristics

Different running time and performance characteristics of functions can be incorporated into consistent hashing. *The key problem is to determine when and which function to forward.* The forwarding policies need to be cognizant of the warm and cold running times, and the sensitivity to load of different functions.

Assume a load-bound of b, the warm time of a function is w, and the cold time is c (slow-start). The current or the home server will be "0", and the next server in the ring that the function may be forwarded-to will be denoted by "1". Running it on the "home"/local server will result in expected time $E[T_0] = (p_0 w + (1 - p_0 c)S(L_0)$, where p_0 is the cache-hit/keep-alive probability, and $S(L_0)$ is the slowdown in function if the load on the server is L_0. When a function in invoked the load balancer has the choice to either run in on the home server or forward it to the next server, where it is less likely to be found in the keep-alive cache, because the reuse-distance is much larger for the servers down the chain. Therefore we can compute the forwarding regret, $E[T_0]/E[T_1]$.

The properties of bounded-loads allows us to easily compute this value. The probability of being forwarded is small, and is $1/b$ based on Lemma 4 of [170]. The reuse-distance of the function, and hence the hit-rate on the original/home server will be larger: $p_0 > p_1 * b$. Based on our empirical observation of sub-linear performance decrease due to load (elided for space), in the worst case, the home server will be overloaded and alternative server will not be, and hence the ratio of slowdowns, $S(L_0)/S(L_1) > b$. Minimizing the regret, we get that the function should be forwarded if $L > cb/w$. Thus, the effective load upper-bound is *increased* by a factor of cold/warm time, allowing us to run more functions per server. In our empirical evaluation, we will show that this can significantly improve performance over

plain CH-BL with a function-agnostic constant load-bound. If the cold and warm times of a function are not available, then they are assumed to be equal, thus this degrades to classic function-agnostic bounded-loads.

4.3.6 Handling Bursts

Functions come in a variety of frequency classes and are also prone to unpredictable burstiness (i.e., very low inter-arrival-times for a short duration). Identifying these bursts and both keeping latency for such "popular" functions low and preventing them from negatively impacting co-located functions is critical. We have found that handling overload conditions is a key requirement and can significantly affect the tail latency.

Bursty function invocations result in two main problems. First, they cause an increase in server load beyond the actual load-bound, because load is only lazily tracked. The delayed load information can result in a popular function completely overwhelming a server, causing load "hotspots" in the cluster. The second problem is that in extreme cases, the inter-arrival-time is less than the function latency, causing concurrent invocations. Even if these concurrent invocations are run on a "local" server with the function present in the keep-alive cache, there will still be cold starts, since each invocation must run in its own container.

Our solution to these two problems caused by bursty invocations is to detect popular function bursts, "spread" these invocations around multiple servers to prevent cluster hot-spots, and use stochastic/random load updates to introduce randomness into the load-balancing.

4.3.6.1 Detecting Popular Functions with Spatial Sampling Our goal is to detect "popular" functions with low inter-arrival-times, in an online low-overhead manner. Popularity detection must take into account the changing invocation frequencies of different functions

over time, and be low-overhead. We identify the top p percentile of functions by their inter-arrival-times (IAT), or below some explicit IAT threshold, to reduce unnecessary hyperparamaters.

Our approach is general: we first build a histogram of inter-arrival-times using sampling, and then query it. We note similarities with computing reuse-distance histograms, which are the building block of miss-ratio curves. Reuse-time histograms are a simpler version of reuse-distances. Recall that reuse distance is the number of *unique* objects accessed, whereas inter-arrival-time is simply the difference in wall-clock times.

Our solution to identifying popular functions and function bursts is inspired by the popular SHARDS [152] algorithm for building reuse-distance histograms. Following SHARDS, we randomly sample invocations to track individual function IATs. This tracking is simplified by only recording the most recent access time, and then computing the IAT as an estimated moving average of the current IAT and $now - last\_access$. These values are tracked for every function, and functions in the top p^{th} percentile of IATs are considered **popular**. For the sampled functions using spatial hashing, we update their IAT. Note that this approach keeps only a small number of last-accessed-iat entries in memory: "have-been" popular functions are naturally evicted from the tracking list. Because we do not care about reuse-distances, we avoid keeping a tree of reuse-distances, resulting in a simplified SHARDS-like algorithm (see Algorithm 1).

4.3.6.2 Randomly Updating Stale Loads Popular functions represent such a large percentage of invocations yet a small number of functions, that they can be safely spread across many servers without causing cold starts. A fair load balancing algorithm must spread popular functions to ensure QoS for less frequent functions. Because load information is stale, adhering to locality and load can result in servers facing a herd-effect. Randomization is a powerful strategy to ameliorate such effects, however, we must use it judiciously because

Algorithm 1 SHARDS-inspired popular function detection. Functions with the top p percentile of IATs are 'popular'.

```
1: procedure UPDATE_SHARDS_POPULAR(func, time)
2:     P ← 100.0
3:     T ← 20.0                                   ▷ Effective sampling rate
4:     R ← T/P
5:     Ti ← abs(hash(func.name))
6:     if Ti ≤ T then
7:         if last_access_times.contains(func) then   ▷ Already in our sample set
8:             iat ← (t − last_access_times[func])/R
9:             last_access_times[func] = t
10:            iat_heap.push((iat, func))
11:        else                                    ▷ First access... iat=='inf'
12:            last_access_times[func] = t
13:            iat_heap.push((t/R, func))
14:    iats_only ← iat_heap.values()
15:    pop_thresh ← percentile(iats_only, p)
```

of the strong effects of locality in FaaS load-balancing.

Our solution is to introduce random forwarding (along the ring) which is proportional to the load of the server, such that popular functions are forwarded with a higher probability. If the (stale) load of the server is L, we update its load by adding gaussian noise with a mean of the *extra anticipated load* on the server based on the staleness and function arrival rate on the server (λ). Specifically, the $L_{\text{noisy}} = L + \mathcal{N}(\mu = \lambda, \sigma = 0.1)$, where $\mathcal{N}$ is a Gaussian random variable. For popular functions, we compare the L_{noisy} to the load-bound. For remaining functions, we continue to use the stale load L. Thus for highly loaded servers "near" the upper-bound, the extra random noise will result in the popular bursty functions being forwarded more, to avoid the herd-effect.

4.3.7 Putting it all together: CH-RLU

Our overall policy, Consistent Hashing with Random Load Updates (CH-RLU), combines all the previously described techniques and insights. When a new invocation arrives, we

Algorithm 2 Random Load Update Forwarding Function

```
1:  procedure CH-RLU-FORWARD(func, server, chain_len)
2:      b, b_max, max_chain_len ← system_params
3:      if chain_len > max_chain_len then
4:          return least-loaded-server
5:      λ ← 1.0/avg_iat                          ▷ Computed from Algo 1
6:      L = Load(server)
7:      if popular(func) then                    ▷ Computed from Algo 1
8:          L = Load(server) + N(μ = λ σ = 0.1)
9:      if L < min(cb/w, b_max) then
10:         server
11:     else
12:         CH-RLU-forward(func, next(server), chain_len+1)
```

query the popular IAT threshold to determine what class of function it is. Functions are distributed via Algorithm 2, which combines the use of SHARDS for popularity detection, cold and warm times for increasing the effective load-bound, and noisy loads. We bound the cold/warm ratio with a final load upper-bound b_max. The load bound parameters determine the locality-sensitivity: higher values of b and b_max increase locality at the risk of resource-contention delays. Similarly, higher values of p results in more aggressive random forwarding and reduces locality.

Forwarding along the chain has diminishing returns of locality, and if the function gets forwarded more than max_chain_len times, we simply run it on the least-loaded server. If the least loaded server is also overloaded, we drop the function. We have also implemented a simple PID controller with hysteresis for horizontal scaling, by using server load averages as the input control signal. This horizontal scaling is conservative, with a large dead-band of 5 minutes, and scaling is triggered only if the at least 50% of the servers are overloaded. As we shall show in the empirical evaluation, CH-RLU significantly reduces the variance in the loads among servers, and thus is more amenable to this horizontal scaling policy.

4.4 Implementation

We have implemented our consistent hashing with random load update (RLU) policy and other load-balancing policies in OpenWhisk, a popular FaaS system. Our changes amount to more than 1,700 lines of code across many OpenWhisk components, but are primarily in the load-balancer class. In this section, we describe major implementation details, as well as key performance optimizations which significantly improve OpenWhisk performance and scalability by more than 4×.

Our policies are implemented by modifying the load-balancer module of OpenWhisk (see Figure 6.3). CH-RLU is implemented by modifying the existing OpenWhisk "container sharding" policy, which also uses consistent hashing, and forwards functions using only available memory as the load metric. We use OpenWhisk's existing consistent hashing implementation, permiting an "apples to apples" comparison, and also making CH-RLU a "drop-in" replacement for the OpenWhisk default load-balancing. At the invoker level, we adapt FaasCache's GreedyDual keep-alive policy, which increases the keep-alive effectiveness compared to OpenWhisk's default non-resource-conserving TTL eviction [42].

The CH-RLU algorithm described in the previous section requires two main additional pieces of information from each invoker/server: the load averages, and the cold/warm running times of functions. Both of these are periodically (every 5 seconds) captured and stored in a centralized redis key-value store. The load-balancer in the controller reads these asynchronously: working with stale and inconsistent metrics is our key design goal. The default load-bound, b, is 1.2, and the max load, b_max is 6. Popularity threshold is set to 20%. We did not observe performance to be very sensitive to these parameters, and thus do not need to auto-tune them, and they are suitable as user-inputs.

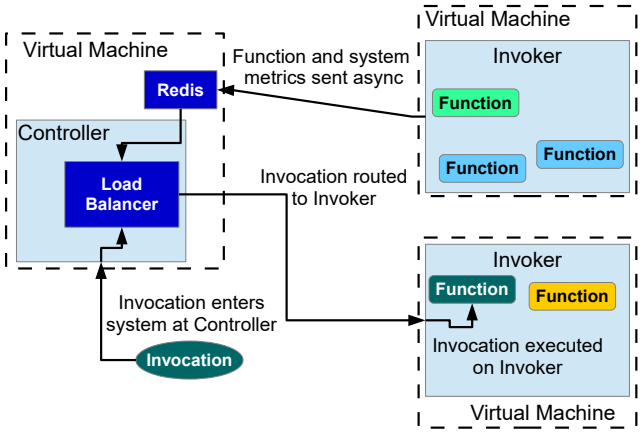


Fig. 4.2: System diagram of relevant OpenWhisk components and communication used to schedule and run function invocations.

4.4.1 Performance Optimizations For OpenWhisk

Since our goal is to run functions under high load, we ran into a large number of OpenWhisk performance and scalability bottlenecks. We found default OpenWhisk to be almost unusably slow and unstable even under reasonable load. We present their details and our actions to overcome them, hoping that the fast-growing serverless computing research field can benefit from our lessons.

In our experience, the primary source of scalability bottlenecks is running Docker containers concurrently. We found significant contention in `dockerd`, Docker's control daemon which handles all the container lifecycle events. Even at moderate loads (normalized server load average close to 1), high dockerd contention can increase tail latencies by *several minutes!*

Currently, OpenWhisk **pauses** each container after function execution, which prevents it from being scheduled by the CPU. It then resumes the container before running the next invocation of the same function (assuming a warm start). Thus each invocation requires these two additional (pause/resume) events to be handled by dockerd, which results in

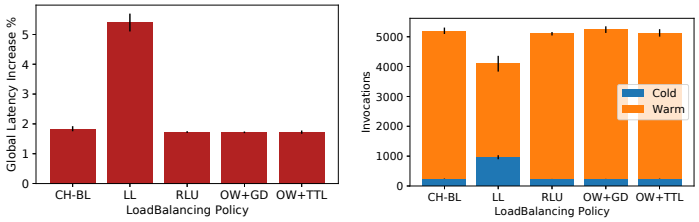
significant lock contention. Because of the FaaS programming model, the pausing is not necessary, since nothing in the container can run after a function has returned. Therefore, we remove these redundant pause/resume operations to reduce dockerd contention. This reduces the OpenWhisk overhead by 0.2 seconds *per-invocation* on average. More importantly, by reducing dockerd contention, we were able to run a much larger number of concurrent functions.

An even larger source of scalability bottleneck is **network** namespace creation time. Using the default bridge networking requires each invocation to create a new TUN/TAP network interface. We found this to be a very expensive operation because of Linux network stack overheads (several 100 ms), and because of dockerd's userspace lock (futex) contention for its networking database. We found that as the *historical* total number of containers launched grows, so does the size of the network-interface database. Dockerd reads and updates this database under the critical section, and the larger database results in higher lock contention. As a result, we were unable to use VMs/servers with more than 4 CPUs after 20 minutes of sustained load, since the dockerd contention resulted in many functions timing out (timeout was 5 minutes)!

We sidestep this problem by not using bridge networking, but instead using Docker's *host* network option and assigning each container a unique port on the host. Implementing the network change required updating the OpenWhisk runtimes used to wrap functions to monitor their specified port. This change allowed us to run functions on larger invokers and under more sustained load, and eliminated most timeouts.

Finally, after a certain request rate threshold, we found the default `nginx` OpenWhisk frontend would crash and return *502 BAD GATEWAY* for all URLs. We did not discover the cause of this problem, and simply bypassed it by letting function invocations to communicate with the controller/load-balancer directly.

CPU limits. OpenWhisk uses the *--cpu-shares* option to set container CPU priority. This



(a) Global Latency Impact (b) Invocation Throughput

Fig. 4.3: Latency and throughput under low-load. Locality-agnostic least-loaded policy has more cold starts and a higher impact on latency.

has an unintended consequence of allowing functions to use more than one CPU core while running. Major FaaS providers constrain functions to a single core unless they have extremely high memory allocations (¡1 GB). In order to stay in line with providers and prevent outsized impact on system load from some functions, we use the *--cpus* flag instead to assign each function no more than one CPU.

Together, these performance optimizations have allowed us to run OpenWhisk on invokers that are 4× larger, and serve more than 6× the load, without dropping functions due to timeouts. We plan to upstream all these performance optimizations in OpenWhisk to provide a higher-performance and lower-jitter control plane for FaaS research and production deployments.

4.5 Evaluation

In our evaluation we present the effectiveness of our load-balancing policy (RLU) using an implementation in OpenWhisk. Our primary goal is to quantify the impact of different load-balancing policies on function latencies under varying load conditions.

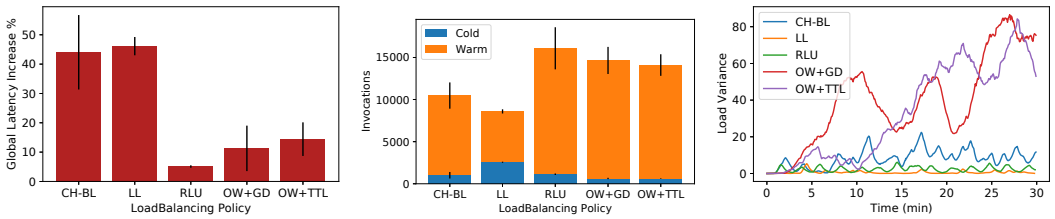
4.5.1 Evaluation Environment

HW and SW Config. We run OpenWhisk in a distributed mode across 9 VMs. 8 invokers are each in their own VM with 16 vCPUs and assigned to use 32 GB RAM for hosting functions. The final VM hosts the controller, load-balancer, and remaining services, with 12 vCPUs and 50 GB RAM to ensure it is not a bottleneck. Metrics about system load were captured every 5 seconds by calling *uptime* on each invokers VM and normalized by the number of CPUs on that system. All latency information was recorded by the client, timing the HTTP request until the request completed. We make no policy changes to the invoker eviction policy, but use the changes from FaasCache [42] for eviction decisions on the invoker.

Contenders. In addition to our proposed load balancing policy, we compare against the default OpenWhisk load balancing policy (described in Section 4.2.1) with GreedyDual (OW+GD) and 10 minute Time-To-Live (OW+TTL) eviction policies, and implement two other load balancing polices for comparison: least loaded (LL), and consistent hashing with bounded loads using stale load-averages (CH-BL). For CH-RLU and CH-BL, we set the max_chain_len=3, a high max load bound, b_max= 6, and a popularity threshold, p=20%. We did not find performance to be particularly sensitive to the load-bound: the function latencies showed little changes across load upper-bounds of $[2-8]$.

Metrics. We examine three main metrics: cold starts, the global average latency across all invocations, and the evenness with which load is spread amongst workers. The first two directly and obviously relate to end user service quality but the third is more intricate. Providers pay for servers to run functions on and don't want those resources going unused and therefore wasted. Equally, a server that is overloaded (not enough CPU or memory resources) will cause a spike in end user latency due to contention of queuing. To quantify the global impact on latency from placement decisions, we normalize each invocation's latency by the ideal (minimum) latency, take the per-function mean of these, multiply each

mean by the percentage of invocations that function had in the whole trace, and finally take the mean of those function latency means. This is essentially a weighted average of latency-increase (i.e., slowdown). It gives some balance between outcomes, for example, a rare function may get several bad placement decisions and thus increase the global latency, or a very common function generally has warm hits and does not impact latency.



(a) Latency (b) Throughput (c) Server Load variance

Fig. 4.4: At high server loads, our RLU policy reduces average latency by 2.2x at higher throughput, compared to OpenWhisk's default policy. It does so by keeping cold starts and load-variances low.

Workload. We convert 12 functions from FunctionBench [163] to run on OpenWhisk. To create a more realistic variety of functions, we create ten copies of each function with unique names, giving us 120 unique functions. Each function clone is invoked at different frequencies mimicing the arrival frequencies of the Azure trace [1]. Our load is generated using the closed-loop load generation tool Locust [172] to invoke functions, running 20 threads for low load, and 120 for heavy load stressing. Locust cannot easily have dedicated threads to invoke each function, so we convert the "frequencies" into weights and use those to randomly choose what function will be invoked next. Each thread will iteratively invoke a random function, and after its completion wait 0-1 seconds before invoking another function. Unless stated otherwise all experiments are run with the above settings, under heavy load, for 30 minutes, and results are the average of 4 runs.

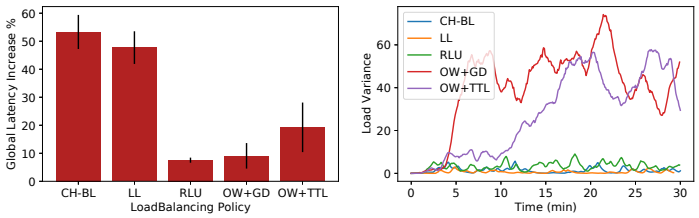
4.5.2 Load-balancing Performance

When we run them under **light load** in Figure 4.3, the policies that use a locality mechanism are essentially identical. The load on any one server is never high enough to impact co-located functions and we never have to forward invocations and incur excess cold starts, giving us a "lower bound" on load balancing. The low 1-2% latencies in Figure 4.3a we see here are due to initial cold starts for functions and the varied overhead imparted by the system analyzed earlier. The least loaded policy is significantly worse as it's lack of locality causes excessive cold starts as evidenced by the high number of cold starts in its invocation results detailed in Figure 4.3b.

Next we run the policies under our **heavy load** scenario, and get a clear distinction between how each of them performs. The two versions of OpenWhisk in Figure 4.4a only increase latency by 11% and 14% respectively which is rather good. They cannot complete with RLU whos increase is less than half of that, a tiny 5% impact on global latency. CH-BL and least loaded increase global latency by over 40%, showing terrible performance in that metric and on invocation throughput.

The wide gap between policies can be understood by comparing the load variance between their workers (Figure 4.4c). OpenWhisk's default policy is to only move a function to another server if the "home" one does not have available memory to run it. While very good for locality (getting fewer cold starts than RLU in Fig 4.4b), it creates severe imbalance on the worker loads. A few workers grow to extremely high load and their functions suffer, while others are mostly empty. RLU intelligently forwards invocations when a worker is near overload, keeping load variance low while protecting locality. Least loaded actually does the best at keeping equal load amongst workers, but at the cost of poor locality.

4.5.2.1 Handling Bursty Traffic Next we take two different bursty workloads to see how the polices handle changes in invocation patterns. The first uses the same closed-loop



(a) Global Latency Impact (b) Worker Load Variance

Fig. 4.5: RLU improves latency by 10% compared to OpenWhisk under bursty load conditions, while keeping a low worker load variance.

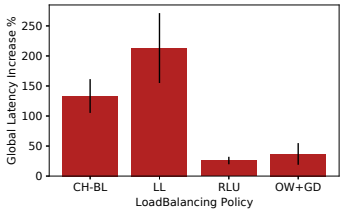


Fig. 4.6: Global latency impact under a 30-minute long rising burst load from an open-loop generator. RLU reduces latency by 17% compared to OpenWhisk.

load generation but adjust the weights by which functions are invoked. Every 30 seconds two of the top weighted functions are chosen to become bursty, and have their weights set much higher. At the end of a burst their weights are returned to normal and another two functions are chosen. As can be seen in Figure 4.5a our policy acheives a 17% lower impact on global latency than OpenWhisk with GreedyDual. RLU represents a 60% reduction to latency over OpenWhisk with its default TTL backend. The more advanced eviction decision choices have a clear effect on improving the system even when the load balancer does not optimize for it. The longer running functions in our workload have a larger effect on system load and the load balancer must be aware of this impact and either spread that heavy popular function around or move other functions off of that server. Again, OpenWhisk does not take load into account and severely overloads some servers while languishing others. We see more sky-high load variances from this bursty workload in Figure 4.5b. Policies that monitor load, our RLU, CH-BL, and least loaded keep tighter control on load variance.

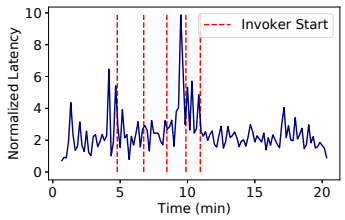


Fig. 4.7: The average normalized function latency over time for a dynamic workload. New invokers are launched at the dashed lines, keeping the latency in check.

The second busrty load is a 30 minute long-rising burst, starting with just a few invocations per second and reaching a sustained peak of roughly 18 invocations per second at roughly 25 minutes. We generated this load with a custom open-loop load tool that fires invocations but does not block waiting for completion. New invocations are continually fired in a preset pattern of function types and times. The global latency impact of this final scenario can be seen in Figure 4.6. Only the final 10 minutes of the workload place the system under extreme load, and the differences between policies reflect this. CH-BL and least loaded cannot keep up with the suddenly changing load, causing a latency increase of over 100% and 200% respectively. RLU's 25% increase in global latency is still significantly better, 30% lower, than OpenWhisk. Our policy is able to make ideal choices for function placement under a varient of realistic workload scenarios.

4.5.2.2 Scaling Lastly we want to demonstrate how our policy reacts to scaling the number of workers as demand increases. We start our cluster with only 3 invokers and increase applied load up to the heavy load scenario above. Rather than starting with the 120 threads of the heavy load with this smaller cluster, we adjust the scenario to start with a single thread and add a new one every 6 seconds, reaching the final thread count at about minute 12.

As the average invoker load increases, the controller activates a new worker and starts directing work towards it. New workers are kept under the load bound of 6 and see load

similar to our previous experiments that had a constant load. Figure 4.7 shows the function latencies (normalized to respective min. warm times). Preceeding each worker being started is a rise in overall latency, which then falls after the invoker has come online and starts taking additional load. Thus, our horizontal scaling is able to dynamically keep the function latency in check, even though it only uses coarse-grained server load metrics.

4.5.2.3 Load-balancer Overhead More complicated routing decisions naturally mean they are more computationally expensive to perform. Even so, RLU is on significantly slower making individual routing decisions, taking on average 1242.6μs to OpenWhisks' 472.3μs. Such times represent a fraction of the time spent per-request by the system and is made up for by our more optimal placements.

5. Ilúvatar: A Low-Latency FaaS Research Control Plane

Providing efficient Functions as a Service (FaaS) is challenging due to the serverless programming model and highly heterogeneous and dynamic workloads. Current open-source FaaS control planes like OpenWhisk introduce 100s of milliseconds of latency overhead, and are becoming unsuitable for high performance FaaS research and deployments. In this chapter we present the design and implementation of Ilúvatar, a fast, modular, extensible FaaS control plane which reduces the latency overhead by more than two orders of magnitude. Ilúvatar has a worker-centric architecture and introduces a new function queue technique for managing function scheduling and overcommitment. Ilúvatar is implemented in Rust in about 13,000 lines of code, and introduces only 3ms of latency overhead under a wide range of loads, which is more than 2 orders of magnitude lower than OpenWhisk.

5.1 Why a new control plane?

We believe that the FaaS control plane is an important component of the modern cloud ecosystem, and presents many optimization opportunities and interesting research questions in system design.

Performance. Because of its central role in coordinating all aspects of function execution, the control plane plays a major role in determining function performance. Managing the function execution lifecycle for hundreds of concurrent invocations imposes a *control plane overhead*, and increases the end-to-end latency. This control plane overhead can be

significant, and affects *all* function invocations, including and especially the "warm starts".

In experiments we witnessed OpenWhisk's 50 percentile latency overhead to be more than 10ms, which is already a significant increase in latency for small functions which dominate real-world FaaS workloads. Worryingly, the 99 percentile overhead is much higher, and rises to as much as 600ms, more details are in Section 5.5. To emphasize, for a median function in the Azure workload which runs for 500 ms, OpenWhisk can increase its latency by 100%. Thus, the control plane plays a crucial role in function performance. We note that these are the best-case warm-start latencies, when the function's containers are fully initialized and in memory. Since function cold starts impose such a major performance penalty (increasing latency by more than 10×), mitigating them has been a major research focus. However, because of temporal and spatial locality of access, caching and prefetching techniques can be extremely effective, and the cold start rate is often less than 1% of all invocations [42]. The majority of invocations are thus "warm", where the performance is dominated by control plane overheads.

System Design. As evidenced by the OpenWhisk architecture presented earlier, FaaS control planes are large, complex distributed systems. A simple system running web services or microservices does not have to deal with sandbox management overheads, nor with highly heterogeneous request sizes. At the other extreme, control planes managing long running containers and VMs, like OpenStack or Kubernetes, face a much lower rate of VM arrivals and departures, and can do careful and "hard" resource allocation using bin-packing [173].

Functions are highly heterogeneous, and can be seen as both latency-sensitive web requests *and* large containers requiring significant system resources for several seconds. FaaS control planes thus have to do *both* low-latency allocation *and* pack CPU and memory resources on their servers carefully to maintain high system utilization. Thus FaaS control planes are one of the more perfect microcosms of challenges in resource management and control in large scale distributed computing. Our new implementation also helps to identify

the current performance bottlenecks and new avenues of OS optimizations.

Control Plane for Experimental Systems Research. Performance-focused FaaS research is already challenging due to the extreme scale and heterogeneity of the workloads. These challenges are compounded by existing control planes like OpenWhisk that are unfortunately highly unpredictable. The control plane jitter and the extreme bimodal cold vs. warm latencies makes it difficult to do reliable and reproducible research [174], and subtle environmental and configuration effects can mask the true effects of new research optimizations. However, it continues to be a key component in developing and evaluating FaaS research [1, 42, 69, 85, 104, 175, 176].

Given the importance of the control plane, we want *predictable* performance to a large degree. In our experience, research in FaaS is often hindered by the large overheads and complexity of existing control planes. Thus, Ilúvatar is designed from the ground-up to be lightweight and provide predictable performance under different conditions. Our system implementation can potentially accelerate the development of new optimizations, clarify our understanding of performance characteristics of this relatively new stack, and provide a control plane for robust experiments. With a robust control plane, the community can share knowledge and advances, while being able to compare against a well-known and trusted baseline.

All aspects of function execution are orchestrated by a FaaS *control plane*, which are implemented by frameworks like OpenWhisk [26]. For using a FaaS service, the user interacts with the control plane for registering and invoking functions, tracking their status, etc. The control plane manages the resources of a cluster of servers, and schedules functions on to them based on its load-balancing policies.

In OpenWhisk, user requests for invoking a function go through a reverse proxy (NGINX) to the central *controller*, which implements, among other things, load-balancing (a variant of consistent hashing with bounded loads by default). The controller puts the function

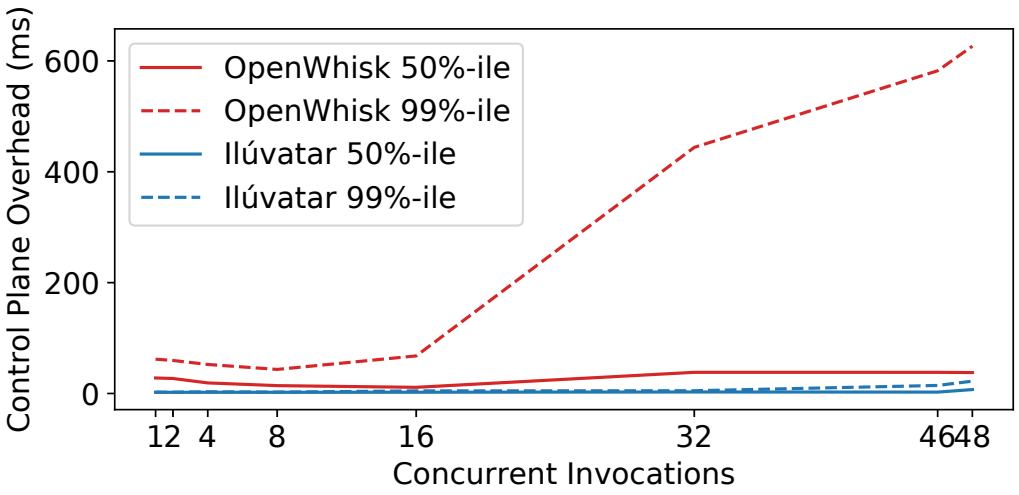


Fig. 5.1: The latency overhead of the control plane, as the number of concurrent invocations increases. OpenWhisk overhead is significant and has high variance, resulting in high tail latency. Ilúvatar reduces this overhead by 100x.

invocation request into a shared Apache Kafka [177] queue. Inside the worker, the invoker service pulls function invocations from the Kafka queue based on that worker's own resource availability. Docker containers running a Go-based control plane agent are used to isolate functions, and each worker maintains a container pool of initialized/warm containers. OpenWhisk logs function results in a CouchDB instance. Importantly, both Kafka and CouchDB are on the critical path, and add 100s of ms to invocation latency. OpenWhisk is highly modular and distributed, with many networked services. All of these, combined with the JVM GC (it is implemented in Scala), results in large and unpredictable latency spikes [69, 178], with slowdowns of more than $10,000\times$ reported [179].

We believe that the FaaS control plane is an important component of the modern cloud ecosystem, and presents many optimization opportunities and interesting research questions in system design.

Performance. Because of its central role in coordinating all aspects of function execution, the control plane plays a major role in determining function performance. Managing the

function execution lifecycle for hundreds of concurrent invocations imposes a *control plane overhead*, and increases the end-to-end latency. This control plane overhead can be significant, affecting *all* function invocations, including and especially the "warm starts". This overhead (end-to-end latency minus the function code execution time) for the PyAES function from FunctionBench [180] is shown in Figure 5.4.

The figure shows the 50 and 99 percentile overheads as the number of concurrent invocations are increased. In each case, we are invoking the function repeatedly in a closed-loop, and concurrent invocations are achieved by using multiple client threads. All invocations are warm starts. The experiment is run on a 48 core server (more details in Section 6.6), and the figure thus shows the performance at low and medium load conditions.

From Figure 5.4, we can see that the OpenWhisk latency overhead is more than 10ms, which is already a significant increase in latency for small functions which dominate real-world FaaS workloads. Worryingly, the 99 percentile overhead is much higher, and rises to as much as 600ms. We also see strange inversions in the scaling behavior: the overhead reduces for certain load-levels, and then increases again. This high overhead, high variance, and uncertain scaling behavior, results in many challenges for FaaS *providers*. Due to these issues, low-latency functions see severe performance degradation, and resource provisioning and capacity planning becomes harder due to the high variance and performance unpredictability.

Some of these latency overheads are an artifact of the architecture. The shared Kafka function queue can be a major bottleneck; and there are no explicit backpressure or load regulation mechanisms, which is compounded by the CPU overcommitment. For the sake of comparison, the figure also shows the latency overhead of Ilúvatar in the same environment. We are able to achieve a per-invocation mean overhead of less than 2ms for almost all the load conditions. Importantly, the tail overhead is also small: less than 3ms for less than 32 concurrent invocations, rising to 10ms when the system is saturated.

To emphasize, for a median function in the Azure workload which runs for 500 ms,

OpenWhisk can increase its latency by 100%. Thus, the control plane plays a crucial role in function performance. We note that these are the best-case warm-start latencies, when the function's containers is fully initialized and in memory. Since function cold starts impose such a major performance penalty (increasing latency by more than 10×), mitigating them has been a major research focus. However, because of temporal and spatial locality of access, caching and prefetching techniques can be extremely effective, and the cold start rate is often less than 1% of all invocations [42]. The majority of invocations are thus "warm", where the performance is dominated by control plane overheads.

System Design. As evidenced by the OpenWhisk architecture presented earlier, FaaS control planes are large, complex distributed systems. Due to the continually evolving needs of FaaS applications and emergence of new sandboxing techniques (such as lightweight VMs like Firecracker [34]), they are sandwiched between the scale and heterogeneity of FaaS workloads on one hand, and the deep stack of OS and virtualization components on the other.

For instance, systems for running web services or microservices do not have to deal with large and highly variable sandbox management overheads, nor with highly heterogeneous request sizes. For reducing tail latency, these systems can often rely on the OS CPU scheduler for processor sharing, but can manually perform CPU allocation at a very fine granularity [181], or use queuing theory techniques [182]. At the other extreme, for longer running containers and VMs, their control planes, like OpenStack or Kubernetes face a much lower rate of VM arrivals and departures. and can do careful and "hard" resource allocation using bin-packing [173].

Functions are highly heterogeneous, and can be seen as both latency-sensitive web requests *and* large containers requiring significant system resources for several seconds. FaaS control planes thus have to do *both* low-latency allocation *and* pack CPU and memory resources on their servers carefully to maintain high system utilization. Thus FaaS control

planes are one of the more perfect microcosms of challenges in resource management and control in large scale distributed computing.

A clean-slate control plane design helps us investigate the fundamental performance tradeoffs and challenges in this fast-evolving ecosystem. Our new implementation also helps to identify the current performance bottlenecks and new avenues of OS optimizations.

Platform for Experimental Systems Research. Performance-focused FaaS research is already challenging due to the extreme scale and heterogeneity of the workloads. These challenges are compounded by existing control planes like OpenWhisk that are unfortunately highly unpredictable. The control plane jitter and the extreme bimodal cold vs. warm latencies make it difficult to do reliable and reproducible research [174], and subtle environmental and configuration effects can mask the true effects of new research optimizations. However, it continues to be a key component in developing and evaluating FaaS research [1, 42, 69, 85, 104, 175, 176]. With OpenWhisk, function performance can be severely affected by a myriad of configuration options, such as insufficient memory for CouchDB, networking configuration, Docker configuration, etc.

Given the importance of the control plane, we want *predictable* performance to a large degree. In our experience, research in FaaS is often hindered by the large overheads and complexity of existing control planes. Thus, Ilúvatar is designed from the ground-up to be lightweight and provide predictable performance under different conditions. Our system implementation can potentially accelerate the development of new optimizations, clarify our understanding of performance characteristics of this relatively new stack, and provide a platform for robust experiments. With a robust platform, the community can share knowledge and advances, while being able to compare against a well-known and trusted baseline.

5.2 Ilúvatar Design

Ilúvatar's design is guided by our experience of OpenWhisk performance, and by our goals of providing predictable performance, modularity, and a control plane for reliable FaaS research.

5.2.1 Architecture and Overview

The Ilúvatar control plane is spread out across a load balancer and the individual workers, and sits above the containerization layers. We intend for Ilúvatar to be the narrow waist [183] in the FaaS ecosystem: with optimizations for DAG scheduling [100], state handling [61], and horizontal scaling [69] implemented above it, and sandboxing and containerization below it. This architecture was motivated by the key question: *Can fast FaaS control planes be implemented with strict layering and separation of concerns?*

We have found that most of the control plane overhead is in the workers, and hence optimizing the worker performance is our major focus. Our architecture is **worker-centric**, and places more performance and load-management responsibility on the individual workers, instead of a more "top-down" centralized approach favored by prior work such as Atoll [184] and others [185, 186]. Top-down resource management requires a consistent global view of the cluster, and is complementary to our work. Predictive techniques for load-balancing, prefetching, scheduling, function-sizing can all be effective, but we want to explore the performance characteristics and limits of *reactive* control planes that work with unmodified container runtimes.

Ilúvatar's main components are shown in Figure 5.2. Clients/users invoke functions using an HTTP or RPC API, with the main operations being `register, invoke, async_invoke, and prewarm`. Workers also provide load and status information to the load-balancer. We use stateless load-balancing, by using variants of consistent hashing with bounded loads

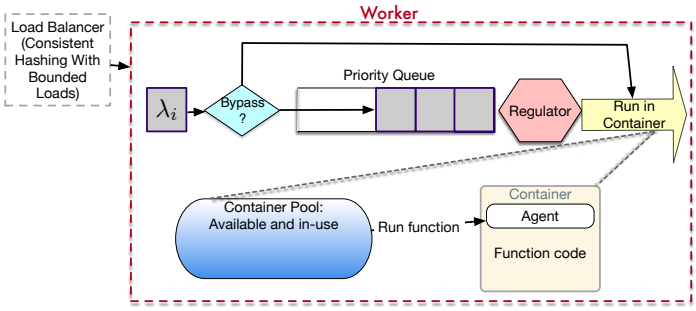


Fig. 5.2: Ilúvatar has a worker-centric architecture. A per-worker queue helps schedule functions, and regulate load and overcommitment.

(CH-BL), which have been proposed for FaaS recently [69]. This is a locality-aware scheme, which runs functions on the same servers to maximize warm starts, and forwards them to other servers only when the server's load exceeds some pre-specified load-bound.

Continuing on the worker-centric theme, the worker API is a subset and almost completely identical to the overall API, and functions can be launched directly on a worker for single-worker setups and benchmarking, without going through a load-balancer and adding unnecessary latency. The workers implement various latency-hiding and burst-mitigation techniques. All functions are launched inside containers, and dealing with the container layer is a major part of the worker. Each worker maintains a container pool of initialized containers for facilitating warm starts, and has an invocation queue for handling dynamic loads. Function characteristics such as their cold and warm execution times are captured in various data-structures and are made available using APIs for developing data-driven resource management policies.

An important contribution and component of Ilúvatar is its principled support for function overcommitment based on its queuing architecture. In many environments, like public FaaS providers, function resources cannot be overcommitted. However, the actual function resource usage is often significantly less compared to their requested "size". This difference is the motivation behind recent "right sizing" work [40,87,96,97,101], and can significantly improve

system utilization. Through its queue-based architecture, Ilúvatar supports a wide range of overcommitment scenarios, including no overcommitment, which is absent from OpenWhisk. By default, OpenWhisk does not overcommit memory, but can overcommit CPUs, which introduces performance interference and potential SLA violations for functions.

5.2.2 Function Lifecycle

New functions first must be *registered*, which entails downloading and preparing its container disk image. The container images are fetched from DockerHub or some other image repository. Container images are composed of multiple copy-on-write layers, and we prepare the images by selecting the relevant layers for the operating system and CPU architecture. The images consist of the user-provided function code and our agent, which is a simple Python HTTP server that runs in each container. Registered functions can then be directly *invoked*, which triggers launching of the function's container. The first invocation is usually a cold start, which entails launching the container image from disk, or from a previous snapshot [187, 188] if available. Each function container starts the agent which listens for and controls the actual function code execution. The agent has two simple commands, a `GET /` endpoint for simple status checking, and a `POST /invoke` to run an invocation with some arguments. When the container is ready, the worker sends an HTTP request to the agent to start the function code execution. We detect the container's readiness using an inotify callback, which is a faster and more generic mechanism for notification compared to Docker's built-in API. Finally, when the function finishes execution, the HTTP call to the container's agent returns, and the container is marked as "available" in the container pool, to be potentially used for future invocations of the same function.

Additionally, Ilúvatar introduces a standard `prewarm` API call, which starts the function's container and the agent inside of it, and adds it to the container pool. This reduces most of the cold start overhead associated with the container. Prewarming can both avoid

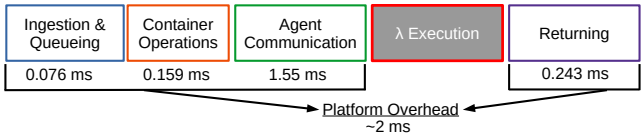


Fig. 5.3: The main components of the Ilúvatar overheads.

a "thundering herd" of cold starts on worker startup, and be an optimization in which the control plane anticipates invocations and prepares containers for them. This allows for a systematic mechanism to implement various recently proposed predictive prewarm policies [1, 112, 189].

Function Latency Breakdown. Throughout Ilúvatar and this paper, we are interested in three main performance metrics. The first is the end-to-end latency of function execution, also called the *flow time*, shown in Figure 5.3. This in turn has two main components: the control plane overhead is the latency of Ilúvatar operations, which are mainly before the start of function execution. The second component is the function execution time, which is determined by the function code, and the load on the system. The function execution time is our baseline, and we compute the *normalized* end-to-end latency by dividing the full latency by the execution time (also called the *stretch*).

A more detailed latency breakdown is shown in Table 5.1. The majority of overhead comes from the communication with the agent which is over HTTP. This is a deliberate choice, since we wanted to be compatible with existing OpenWhisk function images. This can be reduced by using faster IPC mechanisms like in Nightcore [84]. However, these faster communication approaches would reduce compatibility, especially with functions deployed inside VMs.

For OpenWhisk, a similar latency breakdown shows that a large amount of time is spent reading/writing to CouchDB (up to half a second), and the rest of the slowdown occurs in the Invoker (OpenWhisk's worker) and is primarily due to its design and implementation.

Group	Function Name	Time (ms)
Ingestion & Queuing	invoke	0.026
	sync_invoke	0.013
	enqueue_invocation	0.017
	add_item_to_q	0.02
Container Operations	spawn_worker	0.029
	dequeue	0.02
	acquire_container	0.096
	try_lock_container	0.014
Agent Communication	prepare_invoke	0.154
	call_container	1.364
	download_result	0.032
Returning	return_container	0.017
	return_results	0.266

Tab. 5.1: Latency of different Ilúvatar worker components for a single warm invocation.

Interestingly, the load-balancer/controller for OpenWhisk adds less than 3ms of latency even under heavy load, indicating that the worker-level performance is relatively more important. This further motivates our worker-centric design and evaluation focus.

5.2.3 Worker Performance Optimizations

To achieve this low latency function execution for heterogeneous and bursty workloads, Ilúvatar uses two key underlying design principles: resource caching, and asynchronous handling of function life-cycle events.

5.2.3.1 Resource Caching The cornerstone design goal of Ilúvatar is to reduce jitter, which we accomplish by removing expensive operations from the function's critical path. Instead, we cache and reuse as many function resources as possible, which minimizes the "hot path" function invocation latency significantly. This principle is applied in various worker components, which we describe below. A fast **Container Pool Keep-alive** and cached **HTTP Clients** allow for efficient warm start invocations. Pre-allocating and caching

Network Namespaces shortens cold starts—a technique first used for rapid container provisioning [132];

Container Keep-alive. The primary and exemplary application of resource caching is in the container keep-alive cache that Ilúvatar workers maintain. The containers become "warm" when their function has finished execution, and become "available" for the next invocation of the same function. We maintain a pool of all in-use and available containers for each registered function. This container cache implements classic eviction policies such as Least Recently Used (LRU), and size-aware policies like Greedy-Dual-Size-Frequency, as proposed in FaasCache [42].

Network Namespace Caching. For isolation, each container is provided with a virtual network interface and a network namespace. Through performance profiling, we've found that creating this network namespace can add significant latency to container cold starts—as much as 100ms. This is due to contention on a single global lock shared across all network namespaces [132]. To minimze this overhead, we maintain a pool of pre-created network namespaces that are assigned during container creation. The isolation is still maintained, since concurrently running containers do not share the namespace.

HTTP Clients. The worker threads communicate with the in-container agent for launching the function code. Instead of creating a new HTTP client for every invocation, we cache a client per container and use connection pooling. This affects all invocations (even warm starts), and reduces the control-plane overhead latency by up to $3ms$.

5.2.3.2 Async Function Life-Cycle Handling

The second key design principle is to handle various aspects of the function's lifecycle asynchronously off the critical path. Ilúvatar achieves this through background worker threads for certain tasks, and through its Rust implementation which heavily uses asynchronous functions, futures, and callbacks wherever possible.

Keep-alive eviction. One such aspect is maintaining the function keep-alive cache, and ensuring that new functions have enough free memory to launch without waiting on existing containers to be evicted first. Traditionally, eviction decisions would be made in an online fashion, but picking victims and waiting for their removal creates high variance in function execution times. Ilúvatar performs container eviction from the keep-alive pool periodically in the background, off the critical path. This is similar to the Linux kernel page-cache implementation. We maintain a minimum free-memory buffer for dealing with invocation bursts, and periodically sort the containers list for eviction based on caching policies from [42].

Function Queuing. An important component of Ilúvatar's architecture is a per-worker function queue. New invocations are first put into the queue, and are dispatched to the container backend by a queue monitoring thread. This allows us to tolerate bursts of invocations, and regulate the server load.

5.2.4 Container Handling

Ilúvatar uses standard Linux containers for isolating and sandboxing function execution—a "vanilla" and conventional approach. Several exciting new isolation mechanisms for cloud functions have been proposed: such as lightweight VMs [34], unikernels, WASM [32] and other language runtimes [190], etc. Importantly, the sandboxing affects the *cold start* overheads, which account for a tiny fraction of all invocations (usually less than 1%). Our control plane design and performance optimizations are independent of the sandboxing mechanism, and we address the orthogonal problem of optimizing the *warm starts*.

The basic container operations we use are: i) Create a container/sandbox with specified resource limits and disk image/snapshot, ii) launch a task inside it for the agent, and iii) destroy the container. Each container is launched with the CPU and memory resource limits. CPU limits are enforced with cgroup quotas. This limited API allows Ilúvatar to support *multiple* container backends.

By default, we use containerd [191], which is popular container library, also used by Docker. The very rich containerization ecosystem presents a large number of options, and examining their tradeoffs was a major part of Ilúvatar's design process. Importantly, the choice of containerization library impacts the cold start times, and some library operations can take considerable time (100s of ms). High-level container frameworks like Docker are feature-rich and easy to use, but are typically used for long-running containers and are not optimized for latency. Docker uses containerd under the hood, and it provides more fine-grained control and slightly better latency. Functions require a minimal containerization, and a lot of feature-complexity in these large containerization libraries can add to latency. For instance, the crun [192] library which is written in C takes about 150ms to launch a container, whereas containerd (written in Go) needs 300ms, and Docker needs 400ms.

Using containerd allows us to use the OCI container specification [78], and makes it easier to support other container runtimes. For instance, we also support the Docker container backend, which required only a minimal programming effort. Containerd operates as a separate service, and we use it's RPC-based API, which contributes to some latency as well. We contemplated writing our own optimized container runtime in Rust to avoid the overheads due to inter-process communication, extra process forks and system calls, and implement other cgroups and namespace optimizations. However, we ended up going with containerd to keep our control plane small and reusable across container runtimes. We also wanted to investigate and tackle the challenge of getting predictable performance out of higher level containerization services that are not part of the same address space.

Simulation Backend. In addition to containerd and Docker containers, we also support a "null" container backend which is useful for simulations and evaluating control plane scalability. Because of the scale and variety of FaaS workloads, using discrete event simulators for developing and evaluating resource management policies is often necessary. For instance, the recent work on FaaS load balancing [69] uses such a simulator for evaluating their policies

at scale for different subsets of the Azure workload trace. Usually, the simulation is used to augment and complement the "real" empirical evaluation of the same policies which are implemented in FaaS frameworks like OpenWhisk.

However, a major methodological and practical issue is that the policy implementations, workload generation, and analysis, all need to be duplicated across the simulator and the real system. This can lead to subtle and large divergences between the simulation and real environment. Moreover, the simulator cannot capture all the real-world dynamics and jitter, and can suffer from poor fidelity.

In order to aid researchers, Ilúvatar takes a different approach to simulations, and provides *in-situ* simulations. Our "null" container backend does not run any actual function code, but instead sleeps for the function's anticipated execution time. The rest of the control plane operates exactly as with real containers, and we still handle all other aspects of the function's lifecycle. This allows us to simulate large systems and workloads. For evaluating any particular policy, researchers can use the simulator null-backend to evaluate control-plane overheads, warm-starts, etc., without requiring a large cluster. Each Ilúvatar worker can "simulate" 100s of cores, since the CPU resources are only being consumed by the control plane, and not for running actual functions. Alternatively, a large cluster can be simulated with multiple simulated workers.

With this approach, *there is minimal difference* between the simulation and the real system. Thus an experiment can be run in-situ or in-silico, following identical code paths. The main distinction is that API calls to containerd are replaced with internal dummy function calls, and function invocations are converted to sleep statements. All control plane operations, control-flow, logging, resource limits enforcement, etc., are exactly the same as with the "real" Ilúvatar. This also helps with mocking and testing new policies.

5.3 Function Invocation Queuing

As a way to regulate and control function execution and worker load, Ilúvatar incorporates a per-worker invocation queue architecture. Function invocations go through this queuing system before reaching the container manager, which either locates the warm container and runs the function or creates a new container. Each worker manages its own queue, differentiating our design from OpenWhisk's shared Kafka queue.

Motivation. This queuing architecture is motivated by three main factors: i) the bursty nature of the workload, ii) Reducing cold starts due to concurrent invocations, and iii) to give workers additional mechanisms for controlling their load, implementing prioritization, etc. Note that once the function passes through the queue, it is effectively "scheduled" for execution by the OS CPU scheduler. The CPU scheduler of course has its own throttling and controling mechanisms, such as cgroups and the various scheduler tuning knobs. The invocation queue thus acts as a kind of a regulator or a filter before the CPU scheduler, and ideally, "feeds" it the right functions at the right rates for maximizing throughput and minimizing latency.

Because function workloads are so bursty and heterogeneous, running each function immediately can significantly increase the worker load and result in severe resource contention and increase function tail latencies. The queue also helps as an explicit back-pressure mechanism for load-balancing, admission control, and elastic scaling. The queue length is used for accurately determining the true load on the worker, which is a vital input to consistent hashing with bounded loads [69]. This reduces the staleness and noise of using system load average as the load indicator, and makes load balancing more robust.

Queuing invocations also allows us to reduce cold starts. While repeated function invocations are good and increase warm starts, *concurrent* invocations of the same function results in cold starts for all the concurrent invocations, since each invocation needs to be

run in its own container. This is also the "spawn start" [193], which causes severe latency increase of 10s of seconds in public FaaS. If there are n concurrent invocations that arrive at the same time, then the n concurrent cold starts can significantly increase the system load and affect latency of other functions. Instead, by queuing and throttling the functions, we can wait for the invocation to finish, and then use the warm container for the next function in this "herd", and so on and so forth.

5.3.1 Queue Architecture

Ilúvatar's queue architecture is shown in Figure 5.2. We have three main components. From right to left, first, we have a concurrency regulator (or just regulator), which enforces the **concurrency limit**: the upper-bound on the number of concurrently running functions. This lets functions execute "on cpu" without timesharing, and effectively determines the overcommitment ratio. Higher concurrency limits (more than the number of CPUs) means more CPU overcommitment. Note that even with overcommitment, the cgroup quotas still provide proportional allocation (thus a 2 CPU container will still get twice the CPU cycles compared to a 1 CPU container). In addition to concurrency, other factors can also be used to regulate the queue discharge rate. The regulator can be used to run functions only when sufficient resources (such as CPU bandwidth, warm containers, or even accelerators like GPUs) are available.

Ilúvatar can be deployed with a fixed concurrency limit based on the usage requirements, or use its dynamic concurrency limit mode. In the dynamic mode, we use a simple TCP-like AIMD [194] policy which increases the concurrency limit until we hit congestion, which in our case is hit if the system load average increases above some specified threshold. Other metrics are possible: looking at the increase in execution time (i.e., stretch) of the functions could also be used as a congestion metric. The concurrency limit affects the tail-latency, and more advanced policies can be implemented.

The second component is a **queuing discipline.** In the simplest case, we can use simple FCFS, and process functions in arrival order. However, because functions are heterogeneous, this is not always the most appropriate. Instead, we can use the past function execution characteristics such as their cold/warm running times for size-aware queuing such as shortest job first (SJF). We elaborate more on the queuing policies in the next subsection.

Finally, we note that queuing may increase the waiting time for small functions. We thus have a **queue bypass** mechanism, which allows certain functions to bypass the queue and immediately and directly run on the CPU. Bypass policies take the function running time and the current system state as input. Currently, we implement a short-function bypass, where functions smaller than a certain duration are immediately scheduled, as long as the system is under a load-average limit. More effective bypass policies can also consider reinforcement learning approaches, since the action space is simple (bypass or enqueue), and the system state is well defined (functions running and in-queue, etc.).

5.3.2 Queuing Policies

We implement multiple queue policies which leverage the repeated invocations of functions and use their learned execution characteristics to determining each function's priority. To accomplish this, we maintain per-function characteristics such as cold time, warm time, and inter-arrival-time (IAT). We maintain a priority queue sorted by the function priorities, which are computed using their characteristics like arrival and execution time.

FIFO is simplest and invocations are just sorted by their arrival time. For prioritizing small functions, we leverage our bypass mechanism, where the short functions can skip the queue and be scheduled directly on the CPU. Optimizing queuing policies for heterogeneous functions is challenging, and is an NP complete problem even in the offline case [195].

For improving throughput, we use shortest job first (SJF), which helps reduce the waiting

time for short functions, but can lead to starvation for longer functions if the queue never drains. As a tradeoff between function duration and arrival, Ilúvatar by default tries to minimize the "effective deadline" of a function, which is equal to the sum of its arrival time and (expected) execution time. This earliest effective deadline first (EEDF) approach balances both short functions and starvation. In both SJF and EEDF, an invocation's execution time is determined by its (moving window) warm time. New/unseen functions have their times set to zero, to prioritize their execution. If we expect to find available containers for a function, we use its (moving window) warm time as the execution time in both SJF and EEDF. Otherwise, we use its cold time—this also helps in reducing the concurrent cold starts, since the expected cold invocations of some functions in a burst separates them in the queue, and reduces the number of concurrently executing identical functions. This spreading of function invocations over time increases the warm starts and overall performance. Finally, the RARE policy prioritizes the most unexpected functions (i.e., functions with the highest IAT).

5.4 Implementation

Ilúvatar is implemented in Rust in about 13,000 lines of code, and was made open-source and easy to use. Its low latency and lack of jitter are attributable to the various low-level profile-guided performance optimizations we have implemented during the course of its development and testing. Function handling and container management in the worker make up a majority of the implementation footprint and focus. Ours is a heavily asynchronous implementation using the `tokio` library in Rust, and various function lifecycle events spawn new userspace threads and trigger callbacks. The major data structure shared by the various worker threads is the container pool, which is implemented using the `dashmap` crate, which is a concurrent associative hashmap— this provides noticeable latency improvements compared to a mutex or read-write lock. Conversely, we still use a mutex for the queue, since we found

minimal performance degradation compared to a no-queue architecture during profiling. These, and many other small optimizations, keep the Ilúvatar resource consumption small: even under a heavy and sustained load that saturates a 48 CPU server, the worker process uses less than 20% of a single CPU core.

5.4.1 Support for FaaS research

One of our major design goals is for a reliable and extensible control plane for performance-focused FaaS research. We now describe some of the Ilúvatar features and our experiences in extending it.

Performance Metrics. We keep track of all internal and external function metrics (such as their cold/warm execution time histories, inter arrival times, memory footprints, etc.) and provide them to all components of the control plane, and also to external services. One of Ilúvatar's implementation goals was to reduce the reliance on external services for system monitoring etc. We thus track key system metrics like CPU usage, load averages, and even CPU performance counters and system energy usage using RAPL and external power meters. These metrics are collected using async worker threads, and provide a single consistent view of the system performance. Additionally, we also use and provide Rust-function tracing for fine-grained performance logging and analysis. We use the `tracing` crate to instrument the passage of invocations through the control plane components, and obtain detailed function level timing information, which is used for identifying control plane and container-layer bottlenecks.

Adding New Policies and Backends. Using function and system metrics allows for easy development of data and statistical learning based resource management policies to be implemented. Our baseline policy implementations for keep-alive eviction, queuing, load-balancing, are all easily extensible using Rust traits, polymorphism, and code generation. In our experience, adding new policies is relatively straight-forward, even for new-comers. For

example, all the priority-based queuing policies (SJF, EEDF, RARE, etc.) were implemented by extending the base FCFS policy. Implementing and testing these policies took less than a few dozen lines and about four hours for a graduate student unfamiliar with the code-base.

The default container runtime backend is `containerd`, but the interface is small, and supporting new backends is relatively easy. We added Docker support in about 400 lines and one person-day of development effort.

Load-generation and Testing. In the spirit of providing a full-featured system for FaaS experimentation, we have developed a load-generation framework. It can do closed and open loop load generation, and be parameterized by the number and mixture of functions, their IAT distributions, etc. The testing framework can use functions from FaaS suites like FunctionBench [180], or custom sized functions that run lookbusy [196] for generating specific CPU and memory load. The open-loop generation produces a timeseries of function invocations, which is helpful for repeatable experiments. The functions' IAT distributions can be exponential, or be derived from empirical FaaS traces like the Azure trace [1].

For the Azure trace, we start by randomly sampling functions and computing the CDF of their IATs. We compute the expected load level in the system using Little's law, by finding the expected number of concurrent invocations for each function and adding them for all functions. This expected load can be significantly different from the capabilities of the system under testing (for example, 100 concurrent functions will overload a 12 core system). Therefore, we can scale the individual function IAT CDFs to find a suitable load. This also allows us to change the relative popularities of individual functions, and conduct fine-grained sensitivity experimentation (like examining system performance when the popularity of one single function changes, etc.). We can generate larger traces by layering, and merging the traces from multiple smaller workloads.

For synthetic functions (using lookbusy), we use their distribution of running times and memory consumption when generating the workload. When using real functions from a

benchmark-suite like FunctionBench, for each randomly sampled function, we use its average execution time (from the full trace), and assign it the closest function in the suite. For example, if the average running time of a candidate function in the Azure trace is 8 seconds, we represent it using the ML-training function, which has the closest running time of 6 seconds.

5.5 Experimental Evaluation

We have extensively tested Ilúvatar's performance characteristics throughout its development. Here, we present a limited set of its key performance attributes and focus on new insights into FaaS performance. All our experiments are conducted on a 48 core Intel Xeon platinum 8160 CPU, and we restrict the worker to 32 GB memory, running Ubuntu 20.04 using Rust version 1.67.0 and Tokio library version 1.19.2. We are interested in evaluating latency overheads and Ilúvatar's suitability as a low-jitter research control plane. This evaluation focuses exclusively on the performance of the worker, where we think most per-invocation latency improvement opportunities exist. Many effective load-balancing policies have been published, but their impact on latency is limited to balancing decision time and warm start ratio. Our stateless controller's overhead is consistent at less than 0.5ms, and we can thus ignore its latency contribution, for ease of exposition. Our CH-BL based load-balancer maximizes locality and provides 99% warm starts, and we focus on single-worker performance to remove unnecessary confounding factors.

5.5.1 Control Plane and Function Performance

In this subsection, we focus on the latency overheads of Ilúvatar under different workloads and configurations. For these experiments, we do not use any queuing, use a single worker, and focus on the most basic Ilúvatar configuration.

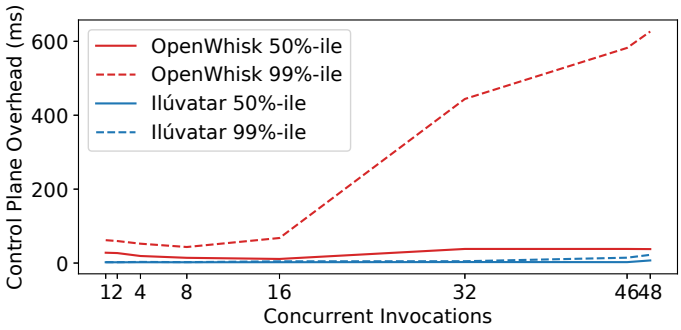
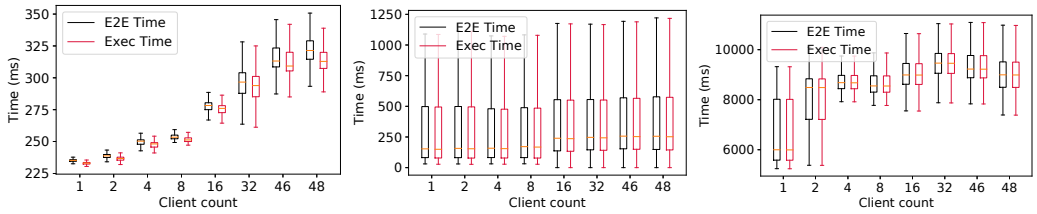


Fig. 5.4: The latency overhead of the control plane, as the number of concurrent invocations increases. OpenWhisk overhead is significant and has high variance, resulting in high tail latency. Ilúvatar reduces this overhead by 100x.



(a) PyAES (b) JSON (c) Video

Fig. 5.5: End-to-end latency and execution times for different functions as we increase the concurrency levels.

We start by examining the control plane overheads under a closed-loop load for 30 minutes generated by different number of client threads. The control plane overhead CDF for the AES function is shown in Figure 5.4. With 48 concurrent client threads, all the CPUs are fully utilized by function execution. Even in this saturated case, the 90 percentile overhead is less than 20ms. Just below this saturation limit, with 46 threads, the 90 percentile overhead drops to less than 10ms, and the average is less than 3ms.

We now provide a more detailed breakdown of the function latency. In Figure 5.5, we look at the end to end (E2E) function latency (i.e., flow time) and execution time of different representative functions under different loads. The flow time is impacted by the control plane overhead and the function code execution time. Both these factors are affected by the

system load, which in turn is affected by the concurrency level. The difference between the E2E and the function execution time is the control plane overhead, which is small for all functions and at all load levels.

Interestingly, a significant source of latency variance is the function execution time itself. For the small, CPU-intensive PyAES function (Figure 5.5a), the inter-quartile-range is 60ms, which is 20% the average execution time. Both the execution time (and hence the E2E latency) and the variance also increases with the system load. This variance is also determined by the non-determinism in the function code. For instance, the JSON function (Figure 5.5b) parses a random json file on every invocation, and thus has a higher natural variance in its execution time. Finally, the video processing function is long and CPU intensive: it downloads and converts a video to grayscale. This magnifies the CPU contention, and the function latency increases from 6 to 9 seconds under heavy load.

The notable increase in execution time for all three functions is a result of high CPU cache miss percentage and a reduction in the instructions per cycle (IPC). We also observed poor cache locality with an increasing number of CPU cores. When the same workload was run on half the number of CPUs (by disabling the rest of the CPU cores), the cache miss percentage significantly dropped (by more than 50%), along with a proportionate reduction in the latency variance. This highlights and emphasizes the deeper architectural challenges of FaaS, which were also shown by [197].

Result: *Ilúvatar overheads are small even under heavy load. Function code non-determinism and system load have a higher impact on the function execution times.*

Cold starts. So far, we have focused on warm-start performance which dominates function workloads. Ilúvatar also incorporates a few optimizations for cold starts. Specifically, we are interested in quantifying the impact of the different container backends (containerd and Docker), and the network namespace caching optimizations. The end to end cold times for various functions are shown in Figure 5.6: this includes both container startup time and

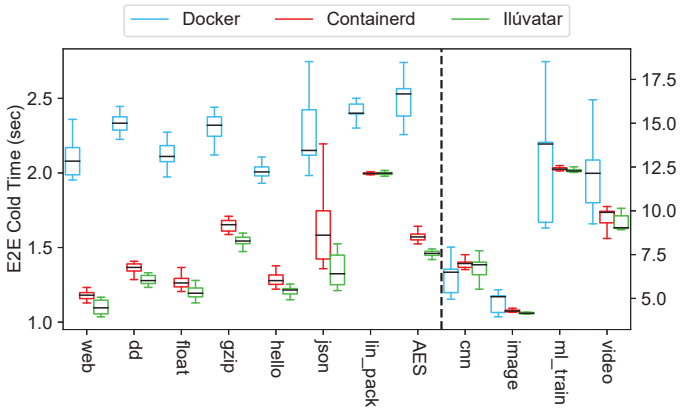


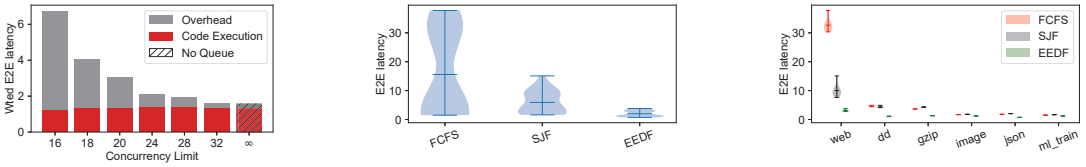
Fig. 5.6: Most functions benefit from using a lower-level containerization and OS object caching on cold starts.

function initialization overheads. In general, smaller functions face a larger impact due to the cold starts, since it represents a higher percentage of their total flow time.

For small functions (left axis of the figure), using containerd (without network namespace caching) reduces the cold start by more than 40%, indicating a clear advantage of using a lighter container runtime. Introducing the namespace caching further reduces the cold start times by 15% compared to unoptimized containerd which creates a new network namespace for each new container. After using the namespace cache, each function invocation sees upwards of $100ms$ improvement in their cold start time. The effects also hold for larger functions (right axis of Figure 5.6), where Docker increases both the average and variance of the latency.

5.5.2 Queuing Performance

Having seen Ilúvatar performance in closed-loop micro-benchmarks, we now investigate the impact of its various queuing components and policies. We use an open-loop load-generator, with a random selection of 21 functions from the Azure traces, and pair them with different functions based on their closest running times. This "stationary" workload has an



(a) Overcommit

(b) Distribution of function latencies

(c) Latency breakdown

Fig. 5.7: Queuing performance on the stationary Azure workload. Size-based policies can provide significant latency benefits.

average 40 requests per second for 30 minutes. This represents an extremely heterogeneous workload in terms of function durations and IATs. Additionally, we also show results from a "bursty" workload generated in the same way, but with one function generating a burst of 18 requests per second for one minute. In this open-loop testing, we prewarm the function containers to prevent excessive cold starts immediately at the start of the workload. The number of containers to prewarm for each function is determined using Little's law by using their average rates and execution times.

Metrics. We use multiple performance metrics to understand and compare different policies. Since functions can differ in execution time, we always normalize their total latency (flow time) by their execution time in an unloaded system. As shown in the previous figures 5.5, even with 1 closed-loop thread, the execution time has variance. For normalization, we use the *average* execution time with 1 thread for all the functions. Second, function popularities can also vary widely. We thus compute the *weighted* latency, where each function's normalized latency is weighted by the number of its invocations in the trace. Thus, the weighted latency represents the latency *per-invocation*.

Saturation Testing. We are primarily interested in how the queuing impacts the waiting time (which is part of the control plane overhead), and the function performance. The analysis of queuing is interesting only in saturated scenarios, where there is enough extra load on the system and not all invocations can immediately run on the CPU. We find this

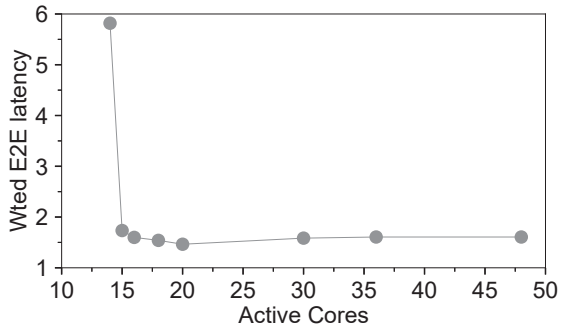
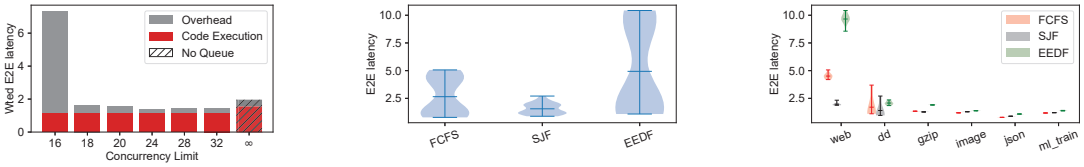


Fig. 5.8: The per-invocation function latencies for different system sizes (# CPUs). We see a sharp inflection point at 16 CPUs, and use that in our queuing evaluation.

saturation point by weak scaling, and decreasing the number of CPU cores available to Ilúvatar (by disabling CPU cores using hot-unplug). The weighted and normalized latencies for different number of CPUs is shown in Figure 5.8, which shows the performance *without queuing*. We see that for our baseline trace, increasing the number of available CPU cores has diminishing returns: the per-invocation latency doesnt benefit when CPUs are increased from 18 to 48. However, we also see a sharp inflection point at 16 cores: decreasing the size to 14 cores results in a very high, almost 6× slowdown. At 16 cores, our workload saturates the system, and we use this system configuration for all our queuing analysis. We note that the alternative is to scale the workload up and run on on all 48 cores. However, as we have shown previously through Figure 5.5, the poor hardware locality results in higher variance in the function execution times, and introduces more performance variance. This variance often masks the control plane jitter, which is of more interest to us.

Impact of Overcommitment. Many frameworks like OpenWhisk inadvertently overcommit CPUs by running more functions than available CPU cores. Ilúvatar can control the degree of overcommitment through its concurrency limit queue regulator. Figure 5.7a shows the effect of this overcommitment, when the EEDF (earliest effective deadline) queue policy is used. The worker is limited to CPU cores, so higher concurrency limits represent different



(a) Overcommit

(b) Distribution of function latencies

(c) Latency Breakdown

Fig. 5.9: Small and bursty functions can get disproportionately impacted due to queuing. A little overcommitment can go a long way to reduce latency.

degrees of overcommitment. As the concurrency limit is increased, we see a reduction in the queuing time (which is a major part of the control plane overhead). For instance, the queuing overhead is negligible when overcommitment level is 2 (i.e., 32 concurrency limit). However we can see a tradeoff: the increased concurrency risks performance interference, and the code execution time also slightly increases (by 4%). For comparison and as a baseline, we also show the "no queue" configuration which is pure processor sharing and there is no limit on CPU overcommitment. Queuing also reduces cold starts due to concurrent invocations. Without queuing, the number of cold starts increased by more than 3×.

For the bursty workload, the impact of overcommitment is even more drastic, as shown in Figure 5.9a. A slight increase in concurrency limit can reduce the weighted latency by more than 3×, indicating that overcommitment is more effective for burstier workloads. Interestingly, the latency improves by 20% with queuing as compared to the "infinite overcommitment" no queuing case. This is due to the increase in function execution time due to uncontrolled CPU contention and interference, which the queue helps ameliorate.

Result: *CPU overcommitment can reduce queuing times, but come with risk of increased performance interference. Ilúvatar's queue design provides a new effective "knob" for managing this tradeoff.*

Queuing Policies and Fairness. Next, we look at the performance impact of the different

queuing policies themselves. We are interested in the impact on the latencies of the different functions. Figure 5.7b shows the normalized latencies of different functions with the different queuing policies. This scenario has a significant amount of queuing: the concurrency limit is set to 16 (the number of CPUs). The function-size aware policies like SJF and EEDF provide much lower latency compared to the standard FCFS: the average latency is reduced by more than $2-3\times$.

A breakdown of the latency of individual functions in Figure 5.7c helps understand this stark performance difference. The queuing in FCFS increases the total time of the extremely small "web" function (13ms running time), which increases its latency by $30\times$. The small-function prioritization by SJF and EEDF reduces this significantly.

The impact of queuing for the bursty workload is even more interesting, as shown in Figure 5.9b. EEDF's average latency is $2\times$ higher than simple FCFS, while SJF is 60% lower than FCFS. Investigating the per-function breakdown again in Figure 5.9c again points to the contribution of the small web function, which is *also the bursty function*. The bursty invocations trigger the cold start mitigation, which deprioritizes them, and increases the queuing time, which disproportionately impacts the small functions.

Result: *Incorporating both function size and arrival times can improve function latency and fairness significantly. Very small functions see a higher % increase due to queuing.*

Ilúvatar vs. Little's law vs. Simulation. Finally, we want to show Ilúvatar's suitability for performance modeling, capacity planning, and as a research control plane for developing and evaluating FaaS resource management policies. We compare the number of concurrent function invocations and queue length (EEDF) with the expected load according to Little's law, computed using average arrival rates and execution times of all functions of our stationary trace. We see that the real system metrics, even with all the inherent burstiness in the Azure trace, and the function execution and control plane jitter, are on average very close to the Little's law estimate. This strongly indicates that our performance is indeed

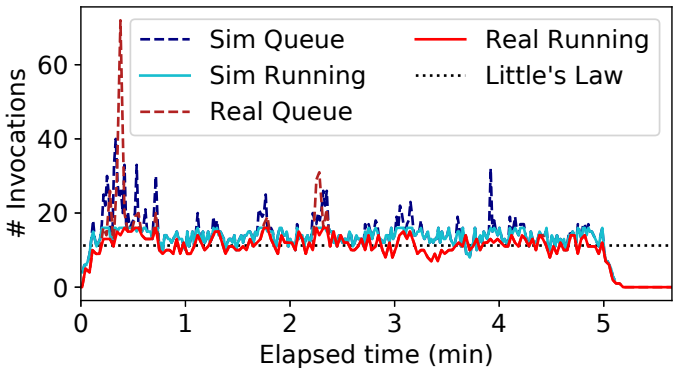


Fig. 5.10: Ilúvatar running in-silico closely models the in-situ performance. Making it a viable exploration opportunity supplementing real experiments.

predictable even with highly heterogeneous workloads.

Additionally, Figure 5.10 also shows the output of our "simulation" container backend described in Section 5.2.4. This backend doesn't run actual function code, but exercises all other control plane aspects. We use constant average function execution times (without accounting for variance and stochasticity) for all invocations. Even though this simulation setup doesn't capture real-world variability and the impact of server load on function performance, we see that the simulation is also fairly closely aligned with the real experiment output. This shows that Ilúvatar's integrated simulation framework captures sufficient system dynamics and provides high-fidelity simulations. This can significantly accelerate FaaS research, especially advances in reinforcement learning based scheduling, which requires high-quality simulations for learning policies.

5.6 Related Work

Alongside the closed-source FaaS control planes from the major cloud providers, a variety of other FaaS control planes exist. Open-source production faas OpenWhisk [26], OpenFaaS [27], nuclio [28], kNative [29], and funcX [25]. Others were made for with targeted research goals in mind [72, 81, 84, 132, 184].

Ilúvatar occupies a somewhat unique spot in the crowded FaaS landscape because of its focus on warm starts and some key constraints in our system design. Techniques for reducing cold start overheads, like snapshots, language isolation, unikernels, all sit "below" the control plane, and can be complemented with fast control planes. At the other extreme end, the predictable nature of serverless workloads has been used to great effect for predictive load-balancing, prefetching, sizing, etc. Ilúvatar is mostly reactive and is worker-centric, and tries to make minimal assumptions about workload predictability and focuses on more general optimizations that can work for arbitrary workload patterns.

FaaS Control Planes. SOCK [132] is closely related to Ilúvatar, and makes similar observations about network namespace overheads, and introduced storage and cgroup optimizations for serverless optimized containers. SOCK is based on OpenLambda [72] and achieves great cold start performance with Zygotes that are cloned into new containers. These optimizations to the container runtime are also applicable to Ilúvatar and are complementary. Using the standard containerd interface allows us to use multiple current and future container backends, and is a deliberate tradeoff.

Nightcore [84] is an integrated control plane and runtime system for low-latency microsecond-scale microservices. It essentially implements containerized RPC, and uses fast message passing between the control plane and the agent. Its special container runtime precludes generic "black box" functions, and it provides a weaker isolation model by running functions concurrently within the same container. In the microservice context, container management and scheduling, dealing with heterogenenous functions, and other challenges are not relevant.

Atoll [184] is a fast and highly scalable control plane, and hugely benefits from pre-allocation and prediction. It has a two level load-balancing setup with functions scheduled to a cluster group which then places them on a worker. Ilúvatar's design and contributions are orthogonal to Atoll's more top-down and predictive approach, and we focus on the "low-level" worker problems.

Open-source control planes like OpenWhisk, OpenFaaS [27], nuclio [28], and kNative [29], are widely used to provide functions as a service. They tackle the competing demands of modularity and features, along with supporting function executions in generic environments. Many FaaS systems use Kubernetes as the resource and container management layer, and its complexity and high latency further inhibits deep understanding and optimizations. OpenWhisk's cold and warm performance has been analyzed in many prior works such as [198] and also as part of other systems [42, 69, 176, 199]. OpenWhisk scheduling design and improvements can be found in [69, 200]. Tighter latency requirements exist when deploying functions at the edge, and OpenWhisk's use on lower powered devices presents even more latency troubles [201–204]. Interestingly, public cloud latencies are also significant, of the order of 50 ms [205], hinting that the problems also extend their control planes.

Function Scheduling. Concurrent to our efforts, queuing of function invocations has been proposed in [179], which implements various size-aware policies like SJF. Surprisingly, and perhaps due to OpenWhisk overheads, their function slowdowns are extremely high: of more than $10,000\times$. An earlier theoretical queuing analysis of flow and stretch metrics is also presented in [206]. In contrast to Ilúvatar's worker-centric design, a centralized core-level allocation design is presented in [185]. In FaaS clusters, the tradeoffs in load balancing and early/late binding are evaluated in [186]. Locality [69] and ML-based [106] techniques for FaaS load-balancing take advantage of the high temporal locality and predictability of the FaaS workloads. Our effort is more focused on reactive systems, and adding predictive allocation will only improve it.

OS scheduler improvements can also improve FaaS workloads [207]. Regulating Linux CPU cgroups shares is also effective in overcommitment [104]. Evaluating the effectiveness of these scheduling improvements when juxtaposed with queuing will be interesting. Scheduling function workflows and DAGs are a growing area [98–100], and we focus on single-invocation optimizations.

6. Black-Box GPU Acceleration for Serverless

6.1 Serverless GPU Challenges

The current resources offered by serverless platforms severely limit the types of function workloads it can support. Memory is limited to a few GB and vCPU access is restricted to a single, time sliced, core [208]. Yet any function with a dataset that exceeds these limits or a problem space that needs parallel computing for timely results cannot be served. GPU acceleration alleviates both of these bottlenecks and has the double benefit of improving existing workloads. We can't get this acceleration without cost unfortunately, several considerations need to be made for their benefits to shine.

Tab. 6.1: Attaching a GPU adds significant time to container startup overhead. All times are in seconds.

Function	GPU	No-GPU	Cold start Slowdown
Imagenet	8.581	6.907	1.25x
Roberta	16.374	14.015	1.17x
Ffmpeg	2.044	0.775	2.64x
FFT	2.648	3.677	0.73x
Isoneural	2.586	1.662	1.56x
Lud	2.125	0.736	2.89x
Myocyte	2.145	0.980	2.19x
Needle	2.292	1.453	1.58x
Pathfinder	1.997	1.029	1.95x

6.1.1 GPUs Containers

Serverless invocations are run inside isolated sandboxes, and creation of these typically happens on the critical execution path. Such "cold starts" add significant latency to an invocation in comparison to actual function execution time. Runtimes for functions are on the order of tens of milliseconds [43], yet container startup may be several seconds: an order magnitude delay.

Creating a new container with an attached GPU is even more expensive. Table 6.1 compares the startup time for new Docker containers with and without a GPU. Cold start times grow up to 3x, adding an average of 1.5 seconds of latency to an already costly process [36, 133, 135, 209]. We captured and broke down time spent starting a new ML inference container with and without an attached GPU in Figure 6.1. The bottom timeline with attachment has an additional second of startup from an Nvidia library hook performing kernel work. Once that is accomplished, the control plane agent starts loading user function code, which uses a more complicated startup procedure to prepare the device and allocate resources, taking an additional 1.5 seconds. All serverless platforms use a warm-start pool of containers to amortize the cold start penalties across future invocations. We created first such container pool for avoiding GPU cold starts using our memory multiplexing and manipulation techniques.

6.1.2 Serverless Scheduling

The workloads in FaaS are highly heterogeneous and represent a unique orchestration challenge in cloud computing. Memory usage, execution duration, and the inter-arrival-time between invocations can vary by several orders of magnitude [43].

Previous GPU-enabling works do not address the potential for worker-level queuing of accelerated Serverless functions. The first proposal [210] used Docker and only shows

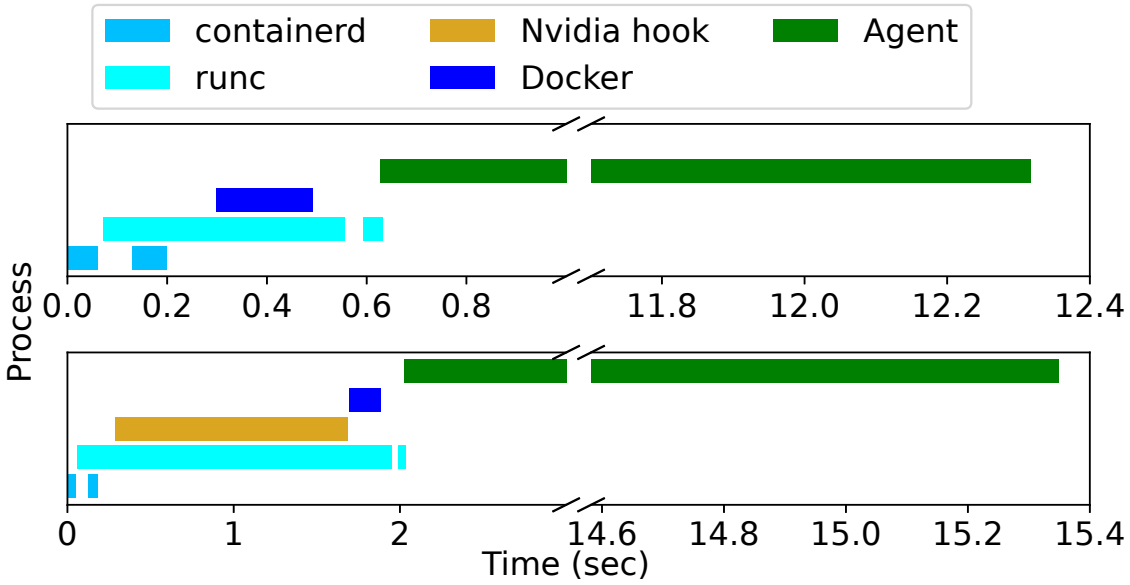


Fig. 6.1: Time spent in a cold start without (top) and with (bottom) a GPU attached to a container hosting TensorFlow inference code. GPU attachment adds over a second to initialization time, and user code setup of the GPU increases agent startup inside the container.

performance improvement against CPU runtimes, neither queuing nor any drawbacks from switching computation are considered. Several examples [48, 211, 212] do schedule ML inference tasks and GPU resources for high GPU utilization, but do so by breaking apart each function to control kernel launches, thereby preclude whole classes of applications from running.

Most scheduling serverless work targets cluster load balancing via locality [66, 69, 94, 186], and avoid queuing at workers. If necessary they use First-Come-First-Serve (FCFS), which works well scheduling on CPUs because it can rely on a large pool of containers to avoid cold starts and expects little queuing of invocations. Unfortunately, a FCFS policy neither take into account the need for fairness between functions under queuing, nor optimal use of limited device resources. Moving data between host and device to change the running application is slow, as seen by [213, 214] and we show below in Section 6.6.1. A new scheduling policy that considers locality of data on device and locality of functions in a smaller container warm pool are needed for viable GPU performance.

GPU batching has been effective [51, 52, 215] in improving GPU utilization and latency when executing ML tasks. The inputs of multiple invocations are merged into one *batch* by the control plane, computed together, and split apart to distribute results. Such white-box solutions eschew isolation by making assumptions about the workload, the fact that inputs can be manipulated and how function code can make use of them. Scientific applications among others are not amenable to batching as their inputs cannot be merged in an attempt at more efficient computation. Black-box scheduling that works with all application classes is required for generic adoption of accelerators in cloud serverless platforms.

6.1.3 Balancing Workloads

To demonstrate that our system achieves fairness with high performance and locality, we run a trace from our experiments (Sec. 6.6.2) and compare the CPU vs GPU performance.

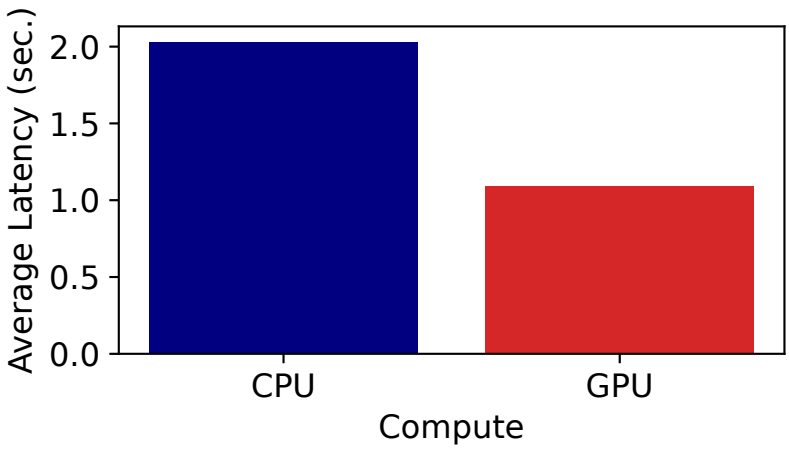


Fig. 6.2: Average invocation latency for a trace is 2x better on a small GPU platform running our desigin compared to a CPU-only system.

Tab. 6.2: The functions in Tables 6.1 and 6.3 come from several sources. They are a subset of the ones we ported to Ilúvatar.

Collection	Functions
ML [180]	Imagenet, Roberta
PyHPC [216]	FFT, Isoneural
Rodinia [217]	Lud, Myocyte, Needle, Pathfinder
Other	Ffmpeg [218]

This representative workload is run on two systems: one with 48 CPU cores, and the other with two Nvidia P100 GPUs. The functions are restricted to only use one of the two compute types, and the outcome is in Figure 6.2. The GPU system cannot run as many functions concurrently, requiring queuing, but still achieves 50% lower latency over its CPU-only counterpart.

6.2 Background

6.2.1 GPU Support for Serverless

Many applications that have adopted FaaS for its on-demand scaling will also benefit from acceleration, as shown by Table 6.3. Machine learning inference requires iterations

Tab. 6.3: Functions' get great performance benefits from running on GPU over CPU. All times are in seconds.

Function	GPU	CPU	GPU Speedup
Imagenet	2.253	5.477	2.44x
Roberta	0.268	5.162	19.26x
Ffmpeg	4.483	32.997	7.37x
FFT	0.897	11.584	12.92x
Isoneural	0.026	0.501	19.57x
Lud	2.050	70.915	34.6x
Myocyte	2.784	39.277	14.11x
Needle	1.979	144.639	73.09x
Pathfinder	1.472	134.358	91.3x

of matrix multiplication and transformations, common tasks for which modern GPUs are designed around. Imagenet and Roberta respectively see a 3x and 20x reduction in latency thanks to acceleration, common numbers for ML improvements. Video encoding via *ffmpeg* has become the a popular task on AWS Lambda, which can leverage specialized hardware found in most GPUs to get a minimum 7x speedup. Scientific computing has started to be used in FaaS [56, 131, 219, 220], but needs acceleration to quickly run expensive algorithms. One such ubiquitious algorithm, Fixed-Fourier-Transform (FFT), sees 13x improvement, and larger, complete applications (Needle, Pathfinder, etc.) are 80-90x faster with acceleration. These are performance gains and application opportunities which are currently being left on the table, and which we are the first to enable.

6.2.2 GPU Sharing Mechanisms

Spatial Sharing. Nvidia has created several technologies for sharing their GPUs that split compute and memory between clients. Nvidia MIG [221] (Multi-Instance GPU) pre-partitions device resources at manufacturing time, and one or more of these virtualized GPU partitions can be assigned to a VM or container via direct device assignment. Once slice(s) of the GPU are assigned, the only way to adjust allocations is to cold start a new

container. Fixed sizes place arbitrary restrictions on the dynamic resource allocations we want to be able to give to functions. Over-allocation causes fragmentation, and the opposite approach will reduce performance or could break functions due to insufficient resources. Idle functions cause immediate underutilization, which could be alleviated via temporal sharing complicating fixed hardware partition management even further.

Multi-Process Service [222] (MPS) allows multiple processes to make share the device concurrently and has been proposed for FaaS [211]. An MPS server and the hardware device perform resource partitioning based on configuration at application start time. MPS is explicitly designed to let cooperative processes share GPU resources, and documentation specifies that it is intended to work with OpenMP/MPI applications. If any process encounters a critical error, all processes connected to the MPS server will crash, meaning one faulty serverless function will break all functions using that GPU.

Temporal Sharing. There has been much work by others in the virtualization of accelerators [213, 214, 223] to allow for temporal sharing. Virtualization allows total control over device resources and scheduling, but imposes significant performance overheads [224]. These approaches [213,214] must duplicate application state in host memory, and allocate/de-allocate all resources when switching between applications.

The GPU driver itself accepts individual compute *kernels* from applications and has a device-internal scheduler which maps them to available compute blocks as they arrive. Kernel schedulers for domain-specific optimizations [211, 225–227] have been designed to coordinate kernels from several applications to improve on device scheduler performance. These operate only on compute kernels to prevent contention and assume active concurrent workloads are coordinated elsewhere to not exceed the device memory. Other schedulers [48, 212, 225] include manipulating and oversubscribing memory as they schedule kernel launches, but they violate the black-box principles we target by breaking apart the application to have tighter control. Interposition and disaggregation techniques [223, 228] allow for multiplexing

memory and compute similar to our design, thanks to the level of control gained by replacing the GPU driver, and are black-box because they do not replace the actual application.

6.3 Related Work

Cold Starts. Overheads from serverless computing primarily come from cold start delays, and research has tackled this problem from several directions. Reducing the time spent starting a new container has taken two approaches: snapshotting for shorter startup time [36,81], or deploying functions inside faster-starting containers [32,34,134]. Alternatively, avoiding cold starts entirely has also proven popular and successful. Predicting the need for a container and pre-warming it has been explored using various techniques [43, 229, 230]. Keeping containers around longer to maximize their performance [42], or load-balancing to guarantee more warm-hits [64, 66, 69]. Bursty functions can cause load imbalance and queuing on systems, and intelligent queuing can avoid additional latency [231].

Serverless GPU. One of the first works to integrate GPUs is [210] using rCUDA [223] to connect disaggregated GPUs in a cluster to containers. It only looks at the performance effect on individual function invocations, not exploring the resource management, queuing, or heterogeneous load issues in FaaS. DGSF [228] combines disaggregation and API remoting to improve utilization, and has the remote node load-balances GPUs.

Software level scheduling of GPU kernels (device compute requests from programs) has been explored by several works, and requires GPU code be extracted separately and not be in a function with CPU code. Kernel-as-a-Service [48] treats individual kernels as first-class serverless functions, managing kernel scheduling and memory allocations directly from the platform. Paella [212] also breaks apart model inference tasks into CUDA kernel launches to enable control plane software scheduling of GPUs and their resource management. Both of these automatically move memory off of the device when kernels are done, and don't have

to worry about applications holding onto device memory when idle.

FaST-GShare [211] profiles ML workloads to monitor how much of the GPU it utilizes, then uses this information to schedule inference tasks on GPUs to maximize utilization. ML inference tasks have fixed sized memory and kernel usages (known tensor sizes, etc.) and this is an effective approach. Other applications can have arbitrary and changing requirements, especially when one considers that function arguments are the main determiner of resource usage, so this idea breaks down when shifting to black-box applications.

FaaS Scheduling. Most FaaS scheduling research has focused on load balancing the cluster, and has tackled it on a variety of directions [13,66,69,94,186,200]. Existing platforms such as OpenWhisk [26] use FCFS queuing on workers and expect the load balancer will avoid queue delays entirely. Only one other work has explored worker-level queuing, comparing it with processor multiplexing during overload scenarios [65].

Serverless Use Cases. The number of serverless use cases than use or can benefit from acceleration is continuously growing. Encoding of videos [53, 116] and analytics on live video streaming [50, 54] are perfect matches for both serverless's scaling and the workload's need for high compute parallelization. Machine learning inference has need for low-latency results, and has seen much work in serverless [51, 52, 215], and can achieve lower latency with acceleration. The exposure of supercomputing resources to run scientific workloads by [25] highlights the demand for parallelization of functions. Such science workloads vary from biomedical research [55, 130], linear algebra [56, 131], to optimization algorithms [33].

6.4 GPU Scheduling Design

A GPU scheduler requires a different execution model than traditional CPU serverless scheduling, coming with new constraints. CPU scheduling knows a function will run on a single vCPU with a fixed maximum memory allocation. Performance cannot be impacted

by concurrent executions, as resources are isolated by the host. The control plane always knows how many items are able to run and where resource limitations exist before starting execution.

Switching device types upends these well-explored assumptions. Concurrent dispatches to a GPU can compete for both compute and memory resources, interrupting each other or even causing potential failures. If there is insufficient memory, do we need to move memory off the device to relieve pressure, or do we need to remove a container? Locality is still important, as a function will have better performance if a container for it already exists, but if the memory is off-device the benefit is reduced. On systems with multiple GPUs, we must be aware that an existing container we have may not match the GPU that's available to run and that we cannot alter this attachment. Finally, it must provide fair access to GPU time and prevent starvation amongst all functions.

6.4.1 Overview

Our scheduling system relies on three primary components, each outlined in Figure 6.3, and we shortly describe each here. Because we need to enforce fairness at the function level, invocations are put into sub-queues unique to their function (1). From these, our larger queue selects one to dispatch from (2) and execute next. It queries the GPU monitor (3) (Sec. 6.4.6) to verify sufficient GPU resources are available to run the invocation. Once this has been confirmed, the function can start executing and make use of the device. The monitor piece also actively monitors GPU compute and memory utilization (5) which is used to avoid device contention and overhead. The queue as necessary uses this information to direct the control plane agent and driver shim (4) (Sec. 6.4.5) inside containers to manage device memory.

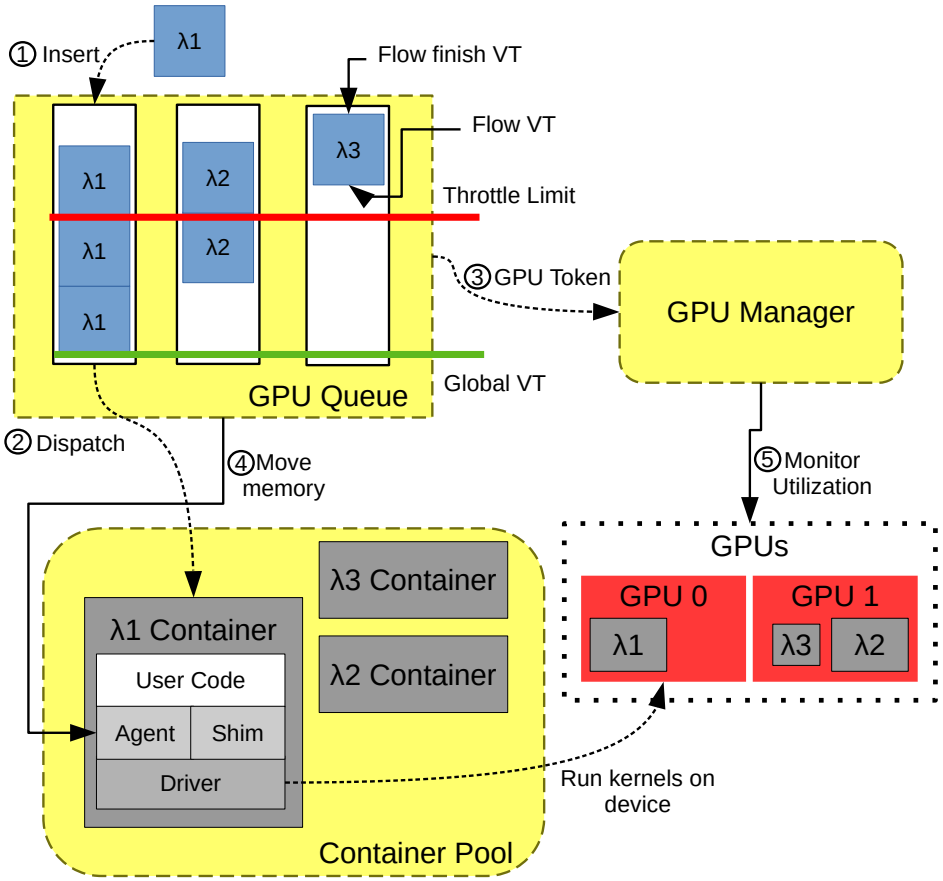


Fig. 6.3: MQFQ-Sticky system design.

6.4.2 Multi-Queuing

Next we delve into detail about how this new queue design can both ensure fairness and handle multiple invocations to run on one GPU. The design of the GPU queue piece is adapted from I/O handling and queuing policies that seek to accomplish similar goals to ours, specifically Mutli-Queue Fair Queuing (MQFQ) [232]. Modern I/O devices support multiple hardware input queues that are concurrently served from, and numerous distinct processes can submit tasks to the device. Multi-queuing maintains state and several per-flow metadata values that avoid contention and ensure fairness between functions (called *flows*), which are listed in Table 6.4.

New items are again inserted into their respective flow (1), and when a flow is non-empty, it is considered *active* and only when active can be selected for dispatching. Each flow is given a *virtual time* (VT) to be used to determine how many times flows have been served in relation to each other. From each VT, a global minimum VT ($Global\_VT$) across active flows is kept via a minimum indicator structure, or *mindicator*.

In MQFQ, each flow is managed by a separate thread, and dispatch independently of each other. As a flow is dispatched from, its VT is incremented and with these the $Global\_VT$ also increases over time. Fairness is enforced by ensuring that no flow is allowed to get T time head of $Global\_VT$. Once a flow exceeds the throttle limit, it is *throttled* and cannot be dispatched from until once again its $VT + T < Global\_VT$. When $T == 1$, we get perfect fairness as any active flow cannot get more service time than any other flow. In practice, T is larger to reduce coordination contention on shared data structures, while keeping flows within fairness bounds. Throttled flows are returned to active status as $Global\_VT$ grows and then may dispatch again.

After the last item from a flow is removed and completes, it transitions to *inactive*, and accordingly no longer contributes to $Global\_VT$. To prevent a flow from accruing credits,

Tab. 6.4: List of symbols in MQFQ-Sticky's design.

Symbol	Description
VT	Virtual Time, device wall clock service time of a flow
$Global\_VT$	Minimum VT across all active flows
T	Amount any flow's VT can exceed $Global\_VT$ before being throttled
D	Device concurrent invocations, can be fixed or dynamic with a maximum
TTL	Time-to-Live for an empty flow to become inactive

upon its transition from inactive to active we update $flow.VT = max(flow.VT, Global\_VT)$.

Lastly, a device can only support a limited amount of function parallelism before causing device-level throttling. To handle this, we limit the number of per-device concurrent dispatches with D tokens representing permission to run on a device. Removing an item from a flow and sending it to the device only proceeds if a token (3) is acquired.

6.4.3 MQFQ-Sticky

Multi-Queuing is an excellent design for systems that must throttle unequal flows and manage external device contention. It is insufficient to simply port the MQFQ pattern to GPUs and expect ideal performance. The dissimilarity of their workloads must be taken into account and integrated into a new wholly new system. GPU functions have different compute and memory footprints, execution runtimes, and importantly require state on-device for good performance: all of which diverge from disk assumptions. GPUs cannot support the 100+ active dispatches of solid state disks [232], especially with the unknown utilization of invocations. We modify the original MQFQ design to account for these significant deviations to get maximum performance out of our accelerator.

Variable Execution Time. The first difference it device dispatches no longer having the same execution time. Without adjustment, this favors long-running functions who would

get more wall time on-device because their VT increase is identical to smaller functions. We therefore switch the VT increment to use each function's average execution time, directly correlating service time with time on device. Shorter functions are allowed more *invocations* than their long counterparts, but each now get equally fair device time.

Device Footprints. Because each function uses different amounts of compute and memory during execution, we cannot have a fixed D as disks do. We track memory usage of running containers and GPU compute utilization to adjust D dynamically with load to minimize contention and execution overhead. A shim that tracks memory usage is described in Section 6.4.5, and GPU manager for compute usage in Sections 6.4.6.

Data Locality. Function's benefit not only from having an existing container available to run in, but also from having their data on-device when they do to run. Flows that become active have their data moved onto the device in anticipation of use (4), if space is available. When a container is about to execute, we move all its data to the GPU to ensure it will have optimal performance. Throttled and inactive functions immediately have their data moved off-device, because we don't expect them to run in the near future. Data monitoring and movement details are described in Section 6.4.5.

The bursty and heterogeneous nature of function arrivals can cause a flow to be empty, but we can often expect a new invocation for it in the near future. To keep data local, we add a time-to-live (TTL) timeout to each flow to prevent it becoming inactive. An empty flow remains *active* from the time of its last invocation's completion until the expiration of the TTL, then we mark it inactive. This TTL uses each function's inter-arrival-time as a base, informed by the bursty nature of FaaS that invocations will either come frequently or after long empty periods [43].

6.4.4 Dispatch Algorithm

The central hub of our queuing and scheduling design is the dispatching step (2), where a flow is chosen to execute next. We make our dispatcher both the primary source of locality for our queue, but also the place for fairness enforcement. Rapidly switching between functions does not allow for reuse of data on device, but always selecting the same flow leads to starvation.

At the start of dispatching, we update the state of each flow in *update_state* according to the $Global\_VT$ in line 3. The first flow state update checks to see if it is and has been empty for longer than the TTL, signifying that it hasn't seen any invocations for some time. It is marked inactive and loses any locality we had maintained, as we don't anticipate it will be used in the near future. Next, flows who match $flow.VT + T \geq Global\_VT$ have exceeded the throttling threshold and are immediately throttled. Any flow that doesn't fall into these categories are kept active, may be dispatched from if non-empty, and hold onto resources.

Once states has been updated, in line 3 we consider for dispatch all flows both active and non-empty. From this subset we must choose a single flow, using a piecewise function on D for best performance. When $D == 1$, the flow with the most number of enqueued items is picked. When $D > 1$ we still select on queue length, but on 3 break ties in the favor of the flow with the least number of currently executing invocations. This encourages multiple flows to progress and reduces the chance of a cold start caused by concurrent execution of the same function. In both cases we are draining the most backed up queues and enabling flow stickiness. If no token is received, we return `None` as no GPU is available to run an invocation. The flow that is ultimately chosen is returned at 3, a token to run on GPU is acquired, and the flow's oldest item is removed and sent out to be executed.

Algorithm 3 Flow Dispatching Algorithm

```
1: procedure DISPATCH
2:     Global_VT ← mindicator.min()
3:     chosen ← None
4:     for flow ∈ flows do
5:         update_state(flow, Global_VT)
6:     cand ← filter(flow.active and flow.len > 0, flows)
7:     sort(cand, on : length)
8:     if D ≠ 1 then
9:         sort(cand, on : in_flight)
10:    chosen ← top(cand)
11:    token ← get_token(chosen)
12:    if token == None then
13:        return None
14:    mindicator.update_vt(chosen)
15:    return chosen.pop()
16:
17: ▷ Update state of flow, given the global VT
18: procedure UPDATE_STATE(flow, Global_VT)
19:    if flow.is_empty and flow.in_flight == 0 then
20:        if Date.Now() − flow.last_exec ≥ TTL then
21:            ▷ Flow has expired
22:            flow.state ← Inactive
23:    else if flow.VT − Global_VT < T then
24:        ▷ Flow has exceeded threshold
25:        flow.state ← Throttled
26:    else
27:        flow.state ← Active
```

6.4.5 Memory Overcommitment

Each container has a custom shim that intercepts calls to the GPU driver, specifically those for initialization and memory allocations. Requests by functions to allocate physical memory are captured in and converted into UVM (virtual device memory) allocations, allocation metadata is stored, and the result is returned to function. When some flow becomes active, the queue, via our agent inside each container, directs the shim to move the memory for the flow's containers onto the device in anticipation of use (4). In the event

memory pressure is too high, this is delayed until an invocation is about to be dispatched, and only the container about to execute migrates memory. When a function's flow is inactive or throttled, we again use the shim to move container memory off the device to free up space. Because our shim tracks memory metadata, we can know how much memory a container needs on-device, and our GPU monitor ensures we don't over-subscribe memory with running invocations.

6.4.6 GPU Load Management

The GPU monitor is responsible for two key mechanisms: tracking GPU assignment and monitoring GPU utilization (5). Creating a GPU context uses physical memory we can't control, so the monitor only allows a fixed number of containers to exist at one time. New containers are launched on the GPU with the most available slots and mapped to the device they are assigned. When is required but not available, we use this mapping to evict a container to make room.

GPUs have limited memory and compute, and if we don't monitor how much function's use, they can interfere with each other and increase execution latency. This limited number of concurrently executing functions is captured in D, and is the primary mechanism of the control plane to prevent interference. D can be either a fixed maximum value, or dynamically adjusted by the control plane at runtime.

Dynamically setting D is dependent on the two main resource limitations of GPUs, on-device physical memory and compute. Because we intercept memory allocations, we can closely track device memory usage of containers thanks to our driver shim, and only allow a new dispatch when the needed container won't overload the device's physical memory. Ensuring there is available compute is not as straightforward to manage as memory due to unpredictable function characteristics. An ML inference task for example could have known compute usage, who's input and weight tensors will have fixed compute uses based on some

execution graph. Other types of GPU functions can launch compute kernels unpredictably, based on the application's internal control flow. The actual size and number of kernel launches often vary with function arguments, making an a priori utilization intractable. Accepting this, we choose to externally monitor device utilization and launch new invocations when headroom is deemed sufficient to support another dispatch. To avoid a thundering herd of launches, we increment the tracked usage by a factor of $1/D$, and let the next monitoring update capture actual usage. When both resources are deemed available for a dispatch, we "increase" D by allowing the item to start executing. Our active management of both resources changes the nature of D from a fixed parallelism limit to a dynamic runtime control knob.

6.5 Implementation

6.5.1 Function Queue

Queuing and dispatching happen inside the FaaS worker where the function will be executed. We implemented our queuing policy and GPU monitoring inside the Ilúvatar [233] worker in 2000 lines of Rust.

To keep dispatch time low, we have a dedicated thread inside the worker running Algorithm 3. This thread will pause and wait for a signal when GPU tokens aren't available or no invocations are available to run. New invocations sent to the worker are enqueued by separate threads, which also book-keep the flow's VT as necessary. Function flows and VT tracking are kept in one data structure, and protected behind Read/Write locks. Once invocations are dequeued, they are sent to asynchronous threads to let the dispatcher thread prepare future dispatches. Completed invocations signal the dispatcher thread, as their completion means new invocations can likely be started with their released resources. The dispatcher thread picks up these changes when it runs next and doesn't become a bottleneck,

having to micromanage individual flows.

This differs from original MQFQ [232] design that was distributed across every CPU core, and relies on lockless and highly-scalable data structures to eliminate contention. We find our adjustments enough to allow concurrent queuing and dispatching without blocking, at the rates our worker will be handling.

6.5.2 GPU Driver Shim

We control and monitor GPU memory by intercepting calls to the native driver using a custom shim. This is written in 500 lines of C code, and to ensure nothing accesses the GPU without us being aware of it, we preload it before launching function code inside each container.

Memory Oversubscription. We use CUDA's Unified Virtual Memory (UVM) to oversubscribe device memory. When using UVM, the application sees a unified host-device memory space, and memory pointers are valid in both spaces. The CUDA driver moves and ensures coherency of UVM memory between the host and device as use and pressure demands, mimicking disk-based swap space in traditional OS computing. Our shim intercepts all calls to the driver for allocations for physical memory made via `cuMemAlloc`, and makes a UVM allocation of the same size using `cuMemAllocManaged`. We record the size and memory pointer position, then return the pointer to the application, which is left none the wiser to our interposition. It can use this memory as if it were physically allocated, reading, writing, or copying it to the host using traditional driver calls. We also intercept and forward UVM allocations, recording their metadata for our use too.

Memory Movement. We fortunately do not need to rely on the CUDA driver to move memory, who's memory swapping strategies we cannot control. It exposes "madvise" and memory prefetching capabilities, which we combine with the shim's metadata to have

total control over memory. When we want a container's memory moved onto the device, we direct the shim to call `cuMemAdvise` to tell the driver we prefer its memory on the device, and `cuMemPrefetchAsync` to order it moved there. After we choose a flow for dispatch, we send this command asynchronously to the container it will execute in. Doing this in a non-blocking manner allows us to overlap prefetching with the control plane marshalling invocation arguments to send to the container. Not having to block while waiting for memory to be moved saves significant time on the critical path. When a flow is throttled or memory is needed to run other functions, we direct the shim to use those same API calls move memory off the device.

Utilization monitoring. We track both GPU compute utilization and per-container and total device physical memory usage. Using NVML [234] bindings in Rust, we query compute utilization and record its instantaneous and moving average, for use in determining D. We query this information every 200 ms to balance having up-to-date information while avoiding excessive CPU utilization on the host. To track memory usage, our shim includes a report of memory allocations still held by the application to the worker alongside other invocation results. This data updates the container's record in the worker, which then tracks memory pressure on the device. When a container needs to run, or a flow is made active, we can evict containers belonging to inactive flows to make room.

6.6 Experimental Evaluation

Here, we showcase the many parts of our design in an extensive series of experiments that fully analyze its impact. All experiments were run on identical systems running Ubuntu 20.04 on kernel version 5.4, with a 48 physical core Intel Xeon Platinum 8160 with hyperthreading disabled, 250 GB of RAM, and an Nvidia P100 GPU running driver version 470.239.06. This isn't the latest GPU hardware, and emphasizes that our design can work with a variety of hardware, doesn't require advanced features, and is easily scalable and adaptable to other

systems.

6.6.1 UVM Shim

The piece underpinning our entire design is our driver shim that captures and rewrites memory allocations made by function code. If we cannot oversubscribe memory or can only do so by significantly increasing code execution time, our system design is untenable.

Memory Management. Our shim enables two abilities, GPU memory overcommitment and control plane management of memory placement. To show the impact of both we run 16 copies of the FFT function from Table 6.2, each using 1.5 GB of device memory that, in aggregate, oversubscribes our GPU's memory by 50%. We then invoke them individually and iteratively 20 times such that all will be executed before starting executing the first again. Before and after invocations, we direct our shim to prepare function memory in various ways. The impact of these different memory policies are displayed in Figure 6.4, with average time spent in-shim shown in red and function code execution in black. With such high oversubscription, UVM driver alone controlling data placement, the `None` case, execution time is 40% worse than the optimal seen in Table 6.3. Execution time is higher when memory must be paged in on-demand from the host as kernels access it, and old memory paged out. Using just the driver madvise mechanisms, `Madvise`, tells the CUDA memory system we would prefer memory on the card, and after execution to prefer it off. This actually is worse than `None`, as madvise calls to the driver don't actually move any memory, we just waste time sending the memory directives with no benefit to execution time. Active prefetching with `Prefetch To` asynchronously copies memory to the device before invocation, and `Prefetch` additionally moves data off device after execution. If we only tell the driver to move memory *onto* the device, it must spend time page swapping to fulfill our request, and potentially move pages we still need. Combining removal of idle memory *and* prefetching memory in anticipation of use reduces function latency by 33%

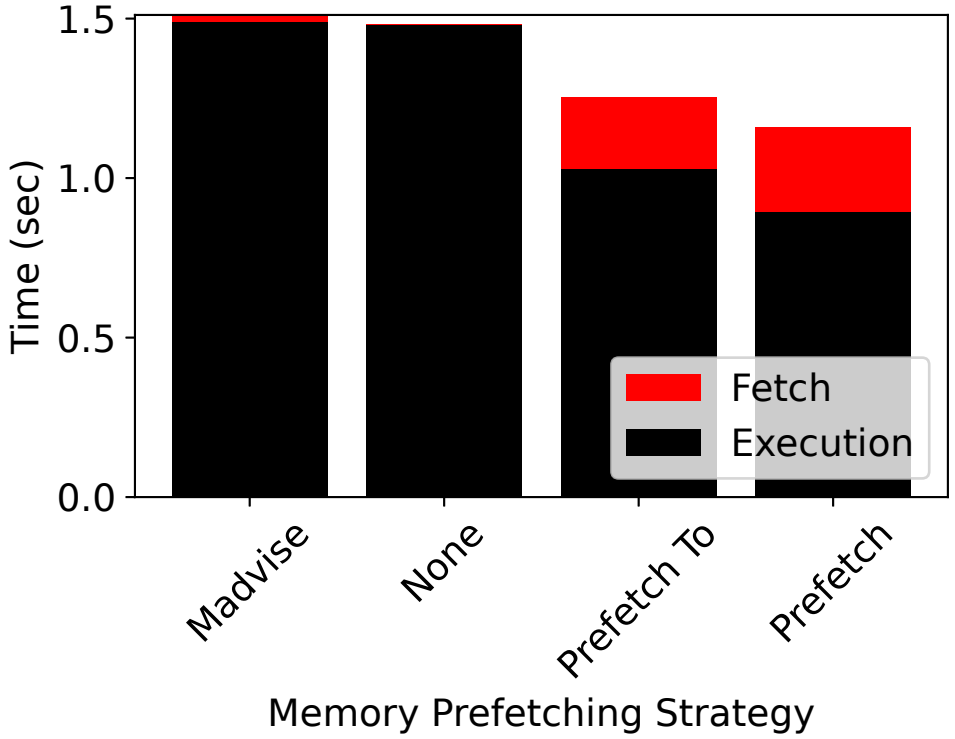


Fig. 6.4: More intelligent memory management improves execution latency. `Prefetch To` moves memory on-device before a function container executes. `Prefetch` additionally moves it off again when the container will be idle.

over `None`. We send prefetching directives off the critical execution path, thereby reducing the latency to just execution time, which now matches the ideal execution time as listed in Table 6.3.

Interception Overhead. We intercept driver calls and replacing physical memory allocations with UVM to enable memory management, but this change comes with a price. To determine the impact, we run each function 10 times, with and without our driver shim enabled, tracking their latency. We never adjust where the function's memory is, so any overhead will be caused by interceptions and usage of UVM allocations. Figure 6.5 collates a variety of different function types and how they were affected. Most functions see zero added execution time from the shift to UVM, our interception being undetectable. The

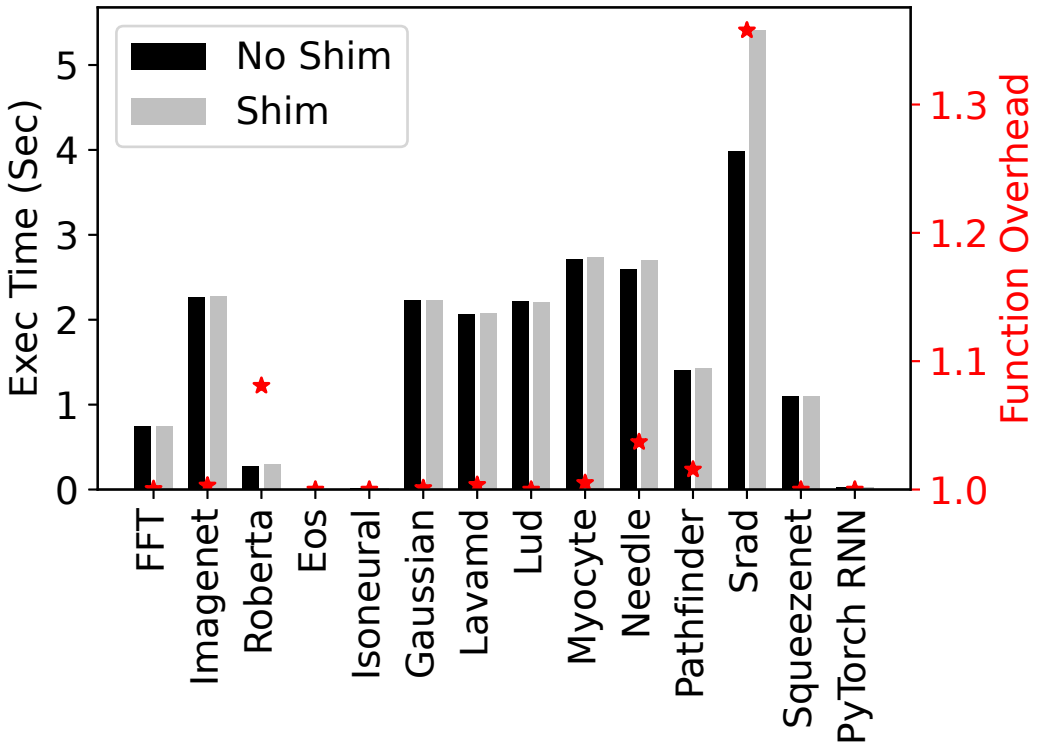


Fig. 6.5: Functions see little to no impact from our interception and substitution of allocation calls. This matches performance promised by Nvidia for UVM applications.

rest see single-digit percentage increases, with `Srad` standing out with a 30% overhead in execution time under the shim. These results are in line with Nvidia's own reporting on the performance change when migrating applications to UVM [235]. With our driver having such little impact, it is ideal for enabling a warm container pool of GPU containers and moving function memory off device when idle.

6.6.2 Queuing Knobs

Knowing we can efficiently manipulate GPU memory, we move to testing our queuing policies built on top of them. Our queue design allows for a number of hyperparameters that are either fixed by configuration or dynamic at runtime. Here we explore the effects of

each knob in isolation to see their effect on performance.

Each experiment is run with the same trace composed of 18 functions, run for 10 minutes, have roughly 2 invocations per second, and presented results are the average of 5 repeated runs. Function were randomly chosen from Azure trace in the same manner as previous works [69, 233], and the trace generated by randomly sampling from an eCDF computed from the Azure data. To map a function name to actual GPU code, we select the closest matching execution duration from Table 6.3 that doesn't exceed that time.

Container Pool. To show the effectiveness of both MQFQ-Sticky's sticky dispatching and having a container warm pool are at decreasing cold hits, we increase the number of functions to 65 while keeping the invocations per second similar. Experiment duration is increased to one hour to prevent first-time invocations of functions, which will always be cold starts, from dominating the results. The high number of functions stresses the need for having a warm pool, as is clear from the results in Figure 6.6. A simple FCFS queuing policy that has no warm pool (e.g. pool size of 1) sees colossal 90% cold hit rate. Comparatively with the same lack of pool, MQFQ-Sticky without concurrency, represented by the blue line, only has 10% of invocations run cold. MQFQ-Sticky's improvement comes from its locality and overrun techniques, as FCFS cannot reorder invocations and must frequently evict container to run the next item. When the pool is increased to 32 containers, only 2% of MQFQ-Sticky invocations are cold, which continue to decline with higher sizes. FCFS also benefits from the warm pool, with both policies having equal cold hits when the pool size is maximized.

When we increase concurrency (D), two scenarios come up: either different functions run concurrently with each other, or a function runs concurrently with itself. The latter causes higher cold starts as we need one container per dispatched invocation of the same function. Green and orange lines in Figure 6.6 show the impact of D on cold starts. Cold hits are higher with concurrency as expected, but are mitigated once the warm pool size grows to 32.

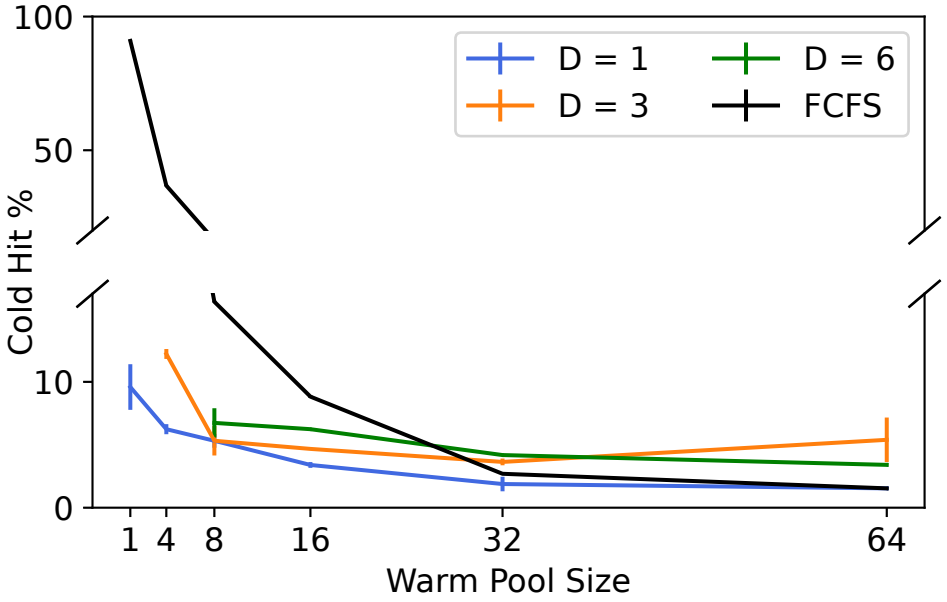


Fig. 6.6: MQFQ-Sticky greatly reduces cold hits compared to FCFS, and is improved with a large container pool size. More cold hits are also caused when D (concurrency) is raised, needing private containers to serve concurrent invocations.

Unfairness. The limited resources on GPUs encourages us to prefer running a function several times because its resources are already on-device. Adjusting the maximum overrun T to find the ideal balance point between fairness and stickiness is important. Figure 6.7 compares the average invocation latency as we increase T. Using a fixed increment to VT (i.e. 1.0 in Figure 6.7) is a simple solution and forces functions' to have an equal *number* of invocations. Such a policy ignores functions' variable execution times, favoring those with long execution times, and as seen in many other scheduling domains leads to unfairness and poor performance. When $T \leq 1$, each flow is immediately throttled after a completed dispatch, forcing it to lose on-device locality, and has a terrible impact on latency. As T increases, latency is improved by up to 25% at $T == 10$, locality causing invocations to complete faster and increasing flows drainage.

The service average by which we increase a flow's VT has a significant impact queuing.

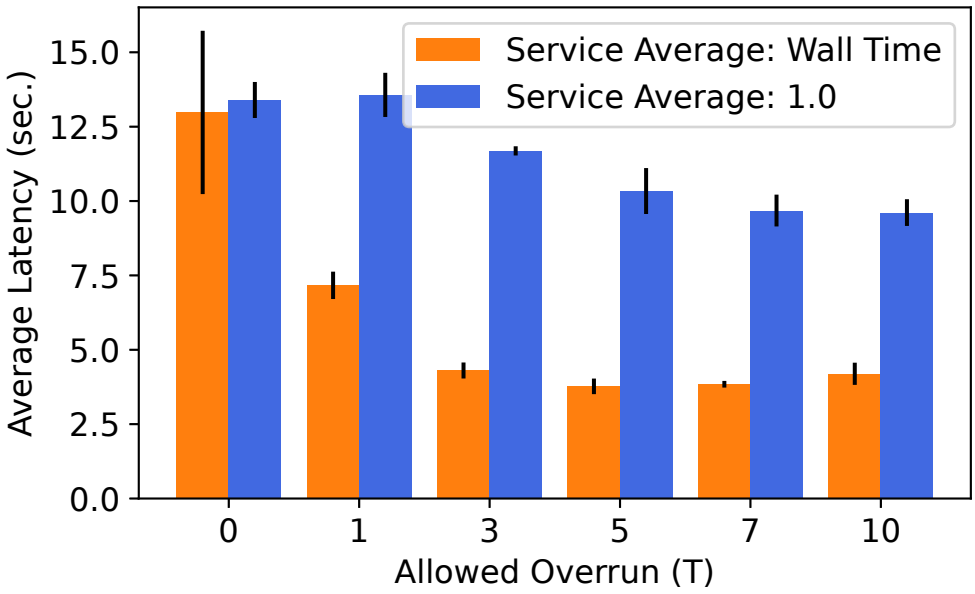


Fig. 6.7: Adjusting T allows flows to overrun one another, increasing data locality and therefore performance. The performance changes when a function's GPU wall time is used to change VT, or the increment is fixed.

Using function execution time (Wall Time) as the service average changes VT to represent *time* on device per-function. Execution time is highly variable in FaaS unlike MQFQ's disk I/O domain, and ignoring it can lead to unfairness. With $T \approx 5$, latency is 70% better than with immediate throttling, and 60% better at this point over the fixed service average. When T becomes too large, device time is hogged by specific flows, unfairness becomes extreme, and we see worse invocation latency.

Active Flow TTL. A larger T is not the only way tool design has to favor locality, we can also keep a flow *active* even while empty. Empty flows remain active until a TTL expires, then they're made inactive and have resources moved off device. Figure 6.8 shows the improvement to both execution time as compared to ideal warm performance (improved by local resources), and latency (additionally improved by priority dispatching) as the TTL grows. Setting any global TTL (the solid line), at even a small 0.1 seconds, improves latency and overhead by 25% and 50% respectively. Increasing the TTL to up to 4 seconds sees

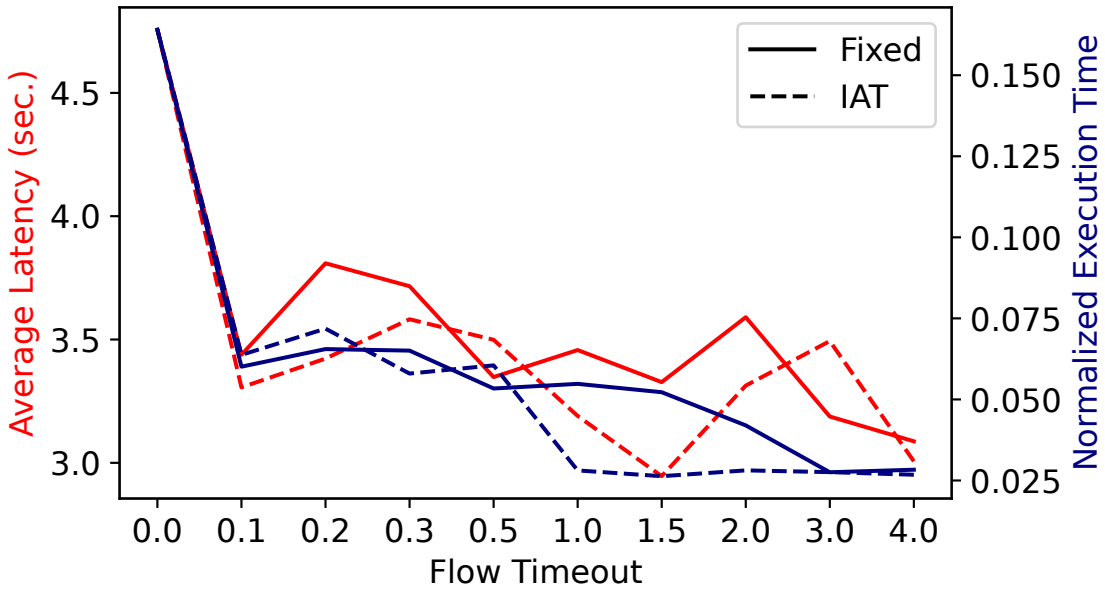


Fig. 6.8: Enabling a time-to-live for flows prevents them from becoming inactive, to keep resources warm on-device. This improves both latency and on-device execution time for functions. Note the non-linear scale on the X axis.

significant, but diminishing, returns.

We can also base the TTL on each function's inter-arrival-time (IAT), rather than have a global fixed time. This method sets the TTL for each flow to the IAT multiplied by the value on the X axis, plotted as the dashed line. Switching generally performs better, and performs best when $TTL = IAT \times 1.5$, reducing invocation latency by 40% over no-TTL. These results fit the heterogeneous and bursty nature of FaaS functions with varied IATs. Even popular functions with low IAT can have idle periods, and keeping their resources warm for short periods allows fast handling when they re-appear. Setting the TTL too high at $IAT \times 3$ prevents them from transitioning to inactive, leaving their VT artificially low, jumping them unfairly ahead in the queue.

Concurrency. Running concurrent invocations is vital to achieving high GPU utilization, and by improving invocation throughput aids user latency. Figure 6.9 examines the effect

on execution overhead as the concurrency D is changed and is made dynamic. When D is fixed, the gray line, overhead is directly correlated with D, added up to 2x overhead from GPU contention. The remaining lines represent the GPU utilization percentage at which we dynamically adjust D but have a fixed maximum (D_max), represented by the X axis value. All see a small increase in overhead when $D\_max == 2$, equal between all values. Higher D_max sees them separate but reach a plateau of maximum overhead, as the GPU manager limits D as it detects high device contention.

Latency for invocations also changes as D is adjusted, throughput at the expense of individual invocation overheads. Latency in Figure 6.10 is decreased across the board when $D\_max == 2$, and nearly 20% better than baseline when utilization monitoring is at 80%. This trend does not continue, caused by queuing at the conservative level of 50% and encroaching execution overhead at higher levels. The scalability of D hinges on a variety of factors: function workload composition, device compute capability, and device memory size. Larger devices can support more functions, but this can be offset by an equally expensive function to run.

6.6.3 MQFQ-Sticky Performance

Now that we've shown how MQFQ-Sticky's performance is affected by hyperparameters, we look at the optimal configuration and compare to other policies. We show three alternative policies as comparison, `FCFS Naïve`, `FCFS`, and `Batch`. `FCFS Naïve` dispatches in a first-come-first-serve order and has no container pool, `FCFS` uses our warm pool and shim to move function memory on-device before executing, and move it back off after each invocation has completed. Thirdly, a variant we call `Batch` also inserts invocations into per-function flows, and on dispatch empties the entire *flow* containing the oldest *item* to execute all removed items serially, but prevents new items from tagging along. It also uses the warm pool and memory movement, only does this after the completion of an entire batch.

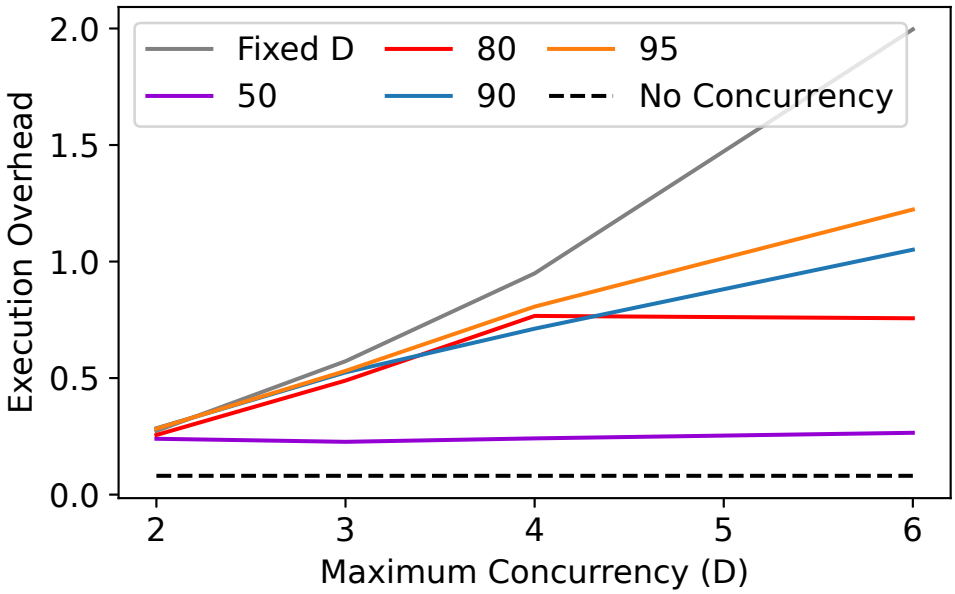


Fig. 6.9: Execution overhead grows as concurrency is increased. The gray line uses a fixed device concurrency, and the remaining lines represent the GPU utilization below which we allow a new dispatch.

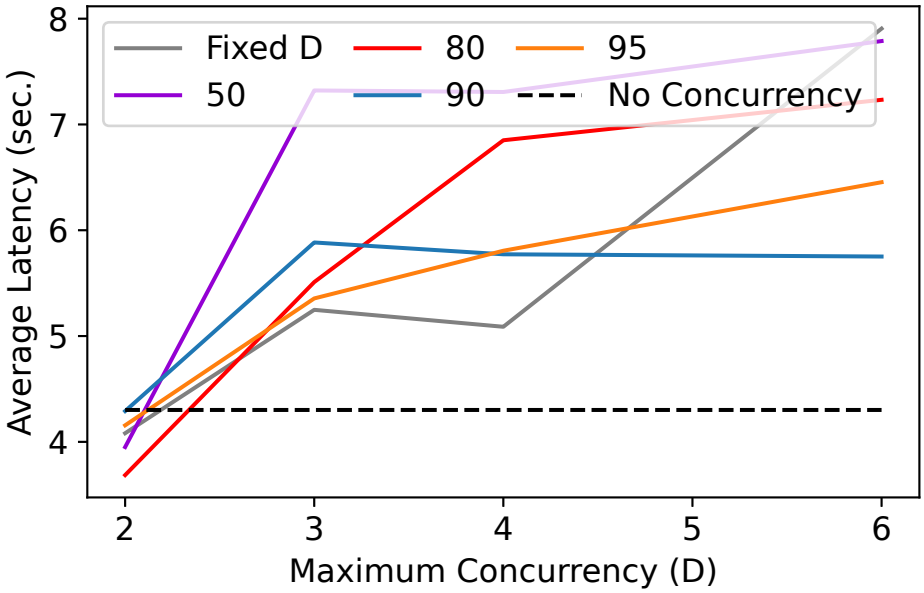


Fig. 6.10: Latency for invocations is affected by concurrency. Increasing D when utilization is low improves latency, but if the threshold is too high, significant queuing delays occur.

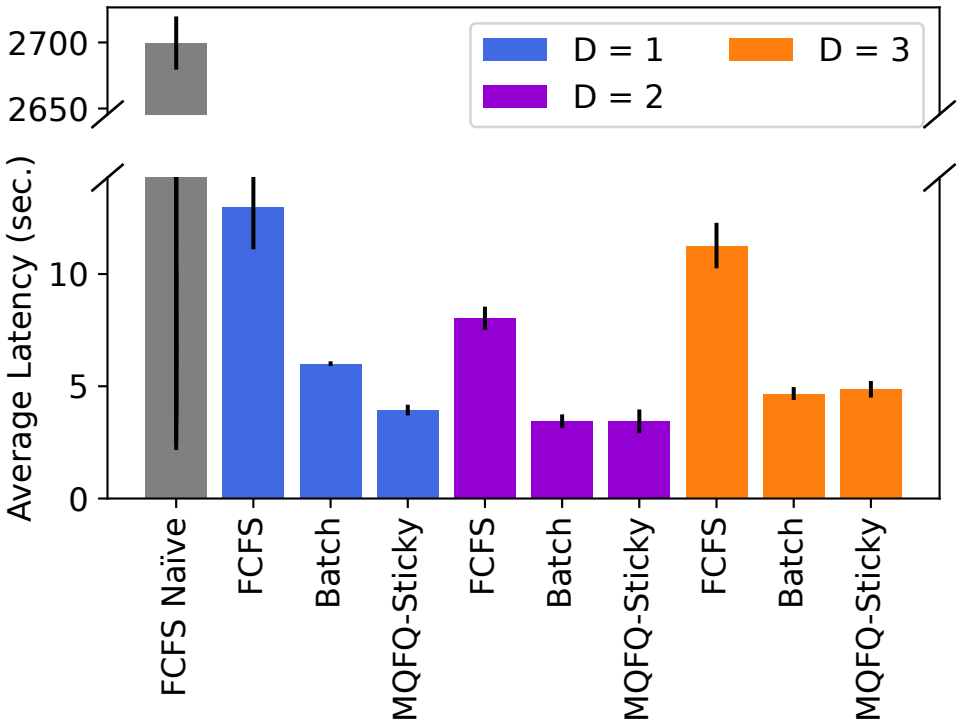


Fig. 6.11: Latency of various queue policies compared.

Starting with the worst-performing policy in Figure 6.11, `FCFS Naïve` has extreme wait times caused by frequent cold starts caused by the lack of warm pool as we saw in Fig. 6.6. Adding the pool and memory management in `FCFS` gives over two orders of magnitude improvement in latency, without increasing the concurrency. MQFQ-Sticky outperforms `FCFS` by 3x with a 3.9 vs 13-second average respective latency thanks to its locality favoring overrun mechanism. Increasing D improves MQFQ-Sticky latency by a further 15% and actually makes `Batch` have equal performance. This is caused by errant lucky items at the end of batches jumping ahead in the queue, in violation of fairness principles. When D is set too high, the device cannot handle the higher concurrency, and all policies suffer degradation from compute contention.

MQFQ-Sticky Fairness. TODO: Text for Figure 6.12

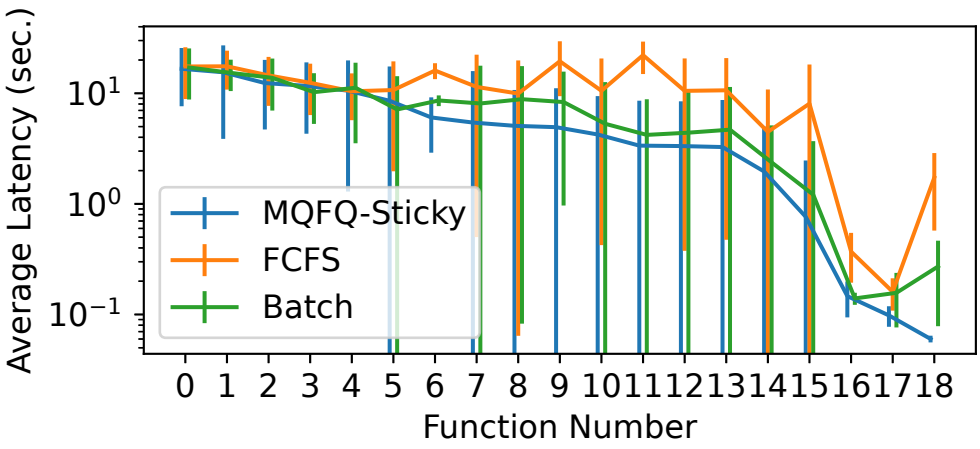


Fig. 6.12: Per-function latency comparison between FCFS and MQFQ-Sticky.

Multiple GPUs. Our system easily scales to orchestrating and dispatching across multiple GPUs. We run the same trace as in previous experiments and show the comparison in Figure 6.13 after we add a second, identical, GPU to our hardware. Two GPUs not only allows us to run $D \times 2$ invocations, but also do on-the-fly load balancing between them to avoid compute contention with higher D. As a baseline, the multi-GPU blue dashed line has 60% lower latency without device concurrency. Scaling D to 6, we see reductions in latency up to *87%* – whereas the single device is quickly overloaded and has worse overall performance. Execution overhead in red increases dramatically with D when only one GPU is available, in the worst case a nearly 6x increase. However, with two GPUs in the red dashed line, once $D \geq 2$ the load balancing mitigation lowers overhead by 45%, and a maximum of 66%. Reduction of this overhead is a significant contributor to lower latencies as well.

Dynamic Compute Selection.

Just because an item can run on GPU, doesn't mean it has to. We took a third of the functions in our trace, those having a speedup from running on GPU of 3x or less, and configured them to only run on CPU. These functions may have longer execution times,

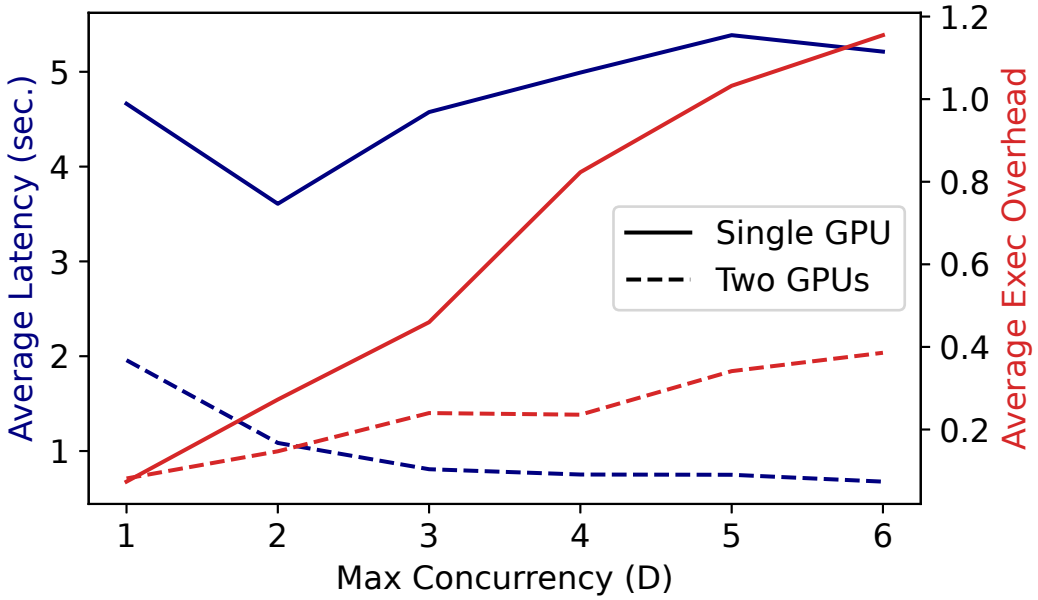


Fig. 6.13: Dispatching to multiple GPUs greatly improves latency and execution overhead compared to a single GPU.

but avoid queuing for the GPU and run immediately on the system's plentiful CPU. The effect on system performance from this shift can be seen in Figure 6.14. Average latency drops from 3.4 to 0.99 seconds, a 72% decrease. This dramatic change isn't just from a few functions avoiding the queue, their removal also reduces pressure on GPU resources. MQFQ-Sticky can more easily maintain locality for the smaller selection of functions and reduce overhead they see. Removing the need to context-switch for certain functions that see little benefit from acceleration can help both categories of function.

6.7 Discussion

Security. Nvidia UVM maintains a per-process page table entries for allocated memory similar to OS PTEs. If a process tries to access memory outside these allocations, it will incur a segmentation fault. Another process will have different mappings and will likewise error trying to access memory from other processes. UVM therefore provides the same security

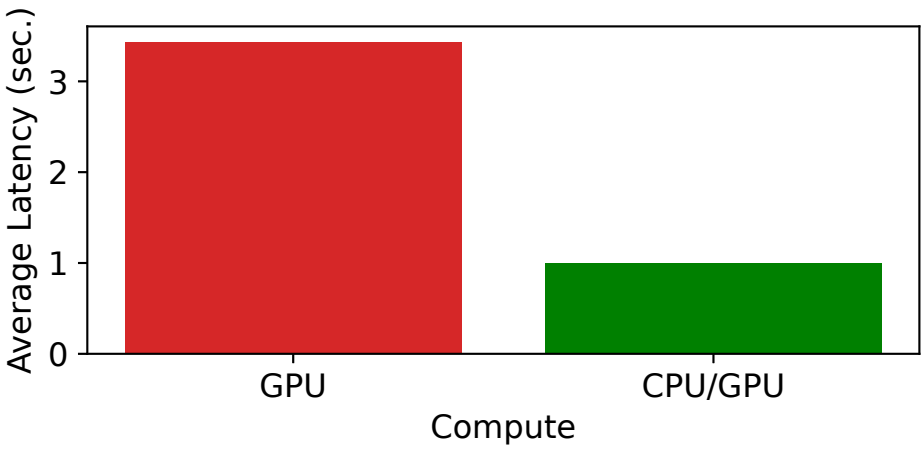


Fig. 6.14: Dynamically selecting what compute a function runs on can reduce GPU queuing and improve global latency.

guarantees as given by traditional RAM virtual memory managed by the OS/hypervisor.

Scalability. We cannot continue to create containers attached to GPUs so long as host memory is available. An open CUDA context holds some device physical memory [236] that cannot be moved by our design, and would eventually consume all usable device memory. On top of that, the device will eventually run out of PTEs to map device-to-host pages.

7. Conclusion and Future Work

7.1 Future Work

This thesis is highlighted by the design and development of the Ilúvatar serverless control plane. I want to keep using Ilúvatar as a springboard for further work both in extension of research described here, and some that is wholly new.

7.1.1 Work Stealing Scheduling

Load balancers cannot scale to serve billions of invocations while keeping an oracle-level knowledge of a cluster of thousands of workers. We must accept some load imbalance on workers – scenarios that become especially common in bursty FaaS scenarios. Work stealing [237–239] allowing workers to adjust load amongst themselves. Idle nodes can *steal* invocations from overloaded machines and run them to locally improve both their utilization and global latency. This can also be done in a locality-preserving manner, by borrowing the ideas from chapter 4. A worker can steal work "up" the consistent hashing ring in addition to the load balancer flowing invocations "down" when a worker is overloaded.

Distributed scheduling [240, 241] has been explored, and the ability of nodes to make on-the-fly decisions has proven highly beneficial. They are naturally more aware of local load and container pool conditions, and can make immediate and more optimal balancing choices.

7.1.2 Polymorphic Functions

Chapter 6 of this thesis introduced scheduling of GPU functions, but also the idea that a function can run on different compute types. It used Python functions capable of running on traditional CPU or be accelerated by a GPU. The last experiment, shown in Figure 6.14, only selected functions to accelerate that saw significant speedup, choosing to run the rest on CPU. I wish to extend these two ideas, that functions

1. Are *polymorphic*, in that without changing code they are capable of running on several distinct compute platforms

2. And that we can *opportunistically accelerate* them by scheduling on devices such as GPUs to improve latency.

The latter goal of opportunistic acceleration is a complex scheduling problem, where a worker can place an invocation on any supported compute considering local conditions. It will have to accurately estimate the latency of running an invocation, knowing a GPU may have faster execution time, but could face queuing delays that would make us prefer the plentiful CPU. Heterogeneous function characteristics (see Table 6.3) become more complex as runtimes change from multiplexing the GPU. Calculating this estimate will need an accurate model of how functions are affected by GPU memory and compute sharing. The worker will also have to be future-looking, knowing that a GPU cold start is an expensive one-time cost that will ultimately lower latency. A simple calculation looking at warm CPU time vs cold GPU time in most cases will always choose the former, yet a popular function will benefit from moving to GPU.

The functions looked at so far a polymorphic because they rely on libraries that implement algorithms for both compute platforms. Serverless can take advantage of its access to function source code to take a more agnostic approach. Many functions are small and take up rare

CPU space, but a GPU has *thousands* of cores to run such tasks. Previous work [242–244] has shown that code can be transpiled or generated to run on GPU, CPU, and more. I want to explore the possibilities of translation, alternate mechanisms to accomplish the same goal, and integrate them with the demonstrated scheduling and memory-manipulation techniques in chapter 6.

7.1.3 Serverless for Distributed Computing

Platforms allow chaining functions via directed acyclic graphs (DAGs) to make larger *applications* where outputs of a function are passed by the platform as arguments to the next one(s) in the DAG. Concurrent computing interactions are limited – two concurrent invocations cannot communicate directly with one another and all data sharing must be done via remote storage. Several platform-managed FaaS dataplanes have been designed to improve data sharing between concurrently running [60, 62] or the input and outputs of DAG functions [38, 39, 66]. Functions should not need to use intermediaries to communicate, especially when computing on a shared problem or dataset.

I want to explore enhancements to the serverless stack to let them interact like true parallel applications ubiquitous to cloud and high-performance computing. This can be taken in several theoretical directions, the first of which is a Hadoop-style [245] cluster scheduler that creates many workers operating on a shared dataset. Applications for these cluster distributed systems are programmed with API hooks to abstract away how they interact, as the platform coordinates placement and data. The other direction is to support MPI [246] functions, where many processes communicate via discrete messages to share data or synchronize themselves. MPI applications also utilize an API that handles all such communication details, abstracting the application away from the actual implementation. Both application types are designed to move between implementations with no code changes, making them easily portable to a serverless context. The platform would just have to

interpose as the expected API and coordinate the possible hundreds of concurrent containers needed to match the scale such applications run at.

7.1.4 FaaS Security

The trusted computing base (TCB) in serverless computing is extremely large, especially when compared to other forms of cloud computing. To start with, a user uploads *unencrypted source code* to the platform for it to use. This code is then run on the same virtualization stack use in cloud, with the addition of a complex containerization system. A new control plane exists, with pieces spread across nodes of varying functionality that move private arguments and outputs between them. Lastly, libraries and language runtimes are also controlled by the platform, with the only guarantee being version compatibility. The platform also has a security concern in that they're running *arbitrary* untrusted code on their hardware.

These challenges make one question if secure FaaS is even possible, and once security measures are considered, new performance problems appear. Cold start costs are significantly higher, with a 512 MB memory enclave taking up to 30 seconds to prepare [247]. A dynamic memory enclave has notable issues when trying to adjust size, but using a fixed allocation wastes memory via fragmentation. Others have pointed out that a function may service multiple end-users [248, 249], and that enclaves themselves cannot be trusted after an invocation and must be "cleaned" in some way.

There are also multiple types of trusted execution environments (TEEs) with varied system designs as well [250–253]. We must ask *where does the control plane live?* and *what is in the TCB?* Currently, the control plane has an agent inside each container that connections function user code via a pseudo-"ABI" (application binary interface) to the outside worker. Presumably this must be made part of the TCB and efficiently be able to move arguments and results between worker and function.

Ilúvatar is in an ideal position to answer the control plane question. It could dynamically support multiple TEEs, already being able to handle several isolation mechanisms, adding TEEs is just a matter of engineering work. Then select the optimal TEE for a function becomes a consideration of a number of factors. Functions have different use cases leading to separate security and performance goals. A CPU-bound piece of code will want access to features that a web service won't, putting both in the same TEE negatively affects the entire system. This break from a one-size-fits-all strategy aligns with the FaaS ethos and unlocks new optimization strategies. The multiple pieces of Ilúvatar can also be lifted to handle complex split system designs used by some TEEs [252, 253]. They place parts of the control plane in the TCB to manage TEEs, and others in untrusted user space for the remainder of required work

7.2 Conclusion

This thesis detailed the new cloud computing paradigm called *serverless computing*, and examined the control plane used to make it a reality. It also went into the challenges faced by this new service, proposing several algorithms, designs, and techniques to improve resource usage and enable new applications to run on it.

Single-worker resource optimization was explored in Chapters 3 and 6, and validated the criticality of warm-starts to low latency in FaaS. Chapter 3 described a cache management design called *FaasCache* that uses function characteristics to better manage in-memory containers for better performance. Knowing that some containers will provide better value if kept available longer, it prunes the cotnainer pool of those less useful to provide better locality and maximize memory utilization. A novel resource, GPU acceleration, was proposed in Chapter 6. Several combined mechanisms allowed the efficient and fair scheduling of black-box functions to receive acceleration from GPUs. Memory is oversubscribed and multiplexed, allowing the control plane can adjust allocations on-demand to prevent device

exhaustion and overhead. It also creates a novel GPU container pool that allows the first locality measures for accelerators. These are tied together with new queue design that favors data locality for performance, but ensures no function faces starvation.

A cluster approach at managing worker load imbalance and even overloading was laid out in Chapter 4. Heterogeneous and bursty Faas workloads are a unique challenge in cloud management, and we created an algorithm to respond accordingly. CH-RLU starts by favoring locality and preferring to always run a function on the same worker. At the same time, it identifies those which might overwhelm workers, and spreads their invocations across several workers in a locality-friendly manner. Overload scenarios are prevented by tracking worker load, estimating the impact of dispatches, and redirecting dispatches when load becomes too high. The locality focus combined with preventing worker exhaustion lower latency significantly over other FaaS load balancing policies.

Chapter 5 detailed a new serverless control plane design called Ilúvatar we created to fix problems with existing offerings. It can be used to accelerate FaaS research and make explore new possibilities that couldn't be done before.

Serverless computing promises to be an efficient and valuable system in cloud computing. In this thesis we have demonstrated several techniques that can greatly enhance the resource management and utilization of FaaS control planes. The combined large scale, breadth of research topics, and heterogeneous workloads of FaaS leaves many more avenues to explore. By all measures it is expected to grow rapidly, meaning new features will need to be developed, performance issues addressed, and features developed to extend the classes of programs that are efficient on it.

BIBLIOGRAPHY

[1] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. *arXiv:2003.03423 [cs]*, June 2020. arXiv: 2003.03423.

[2] Amazon Cloud. AWS Lambda, 2022.

[3] Google Cloud. Cloud Functions, 2022.

[4] Azure. Azure Functions, 2022.

[5] Alibaba Cloud. Alibaba Cloud Function Compute, 2023.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[7] Docker, 2015.

[8] Kubernetes, 2015.

[9] Open Source Cloud Computing Infrastructure - OpenStack, 2024.

[10] Alexander Fuerst, Ahmed Ali-Eldin, Prashant Shenoy, and Prateek Sharma. Cloud-scale vm-deflation for running interactive applications on transient servers. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 53–64, 2020.

[11] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 583–594, 2022.

[12] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 1–16, 2021.

[13] 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.

[14] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020.

[15] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, et al. Xfaas:

Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 231–246, 2023.

[16] Claudio Cicconetti, Marco Conti, and Andrea Passarella. A decentralized framework for serverless edge computing in the internet of things. *IEEE Transactions on Network and Service Management*, 18(2):2166–2180, 2020.

[17] Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. Serverless functions in the cloud-edge continuum: Challenges and opportunities. In *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 321–328. IEEE, 2023.

[18] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. Fog function: Serverless fog computing for data intensive iot services. In *2019 IEEE International Conference on Services Computing (SCC)*, pages 28–35. IEEE, 2019.

[19] I Wang, Elizabeth Liri, and KK Ramakrishnan. Supporting iot applications with serverless edge clouds. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–4. IEEE, 2020.

[20] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 797–813, 2022.

[21] Per Persson and Ola Angelsmark. Kappa: serverless iot deployment. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 16–21, 2017.

[22] Sergio Trilles, Alberto González-Pérez, and Joaquín Huerta. An iot platform based on microservices and serverless paradigms for smart farming purposes. *Sensors*, 20(8):2418, 2020.

[23] Marco Bacis, Rolando Brondolin, and Marco D Santambrogio. Blastfunction: an fpga-as-a-service system for accelerated serverless computing. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 852–857. IEEE, 2020.

[24] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. A case for function-as-a-service with disaggregated fpgas. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 333–344. IEEE, 2021.

[25] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 65–76, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Apache OpenWhisk: Open Source Serverless Cloud Platform, 2020.

[27] OpenFaaS : Server Functions, Made Simple. `https://www.openfaas.com`, 2020.

[28] Automate the Data Science Pipeline with Serverless Functions. `https://nuclio.io/`, 2023.

[29] Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. `https://knative.dev/docs/`, 2023.

[30] Saurav Chhatrapati. Towards achieving stronger isolation in serverless computing. 2021.

[31] The Container Security Platform — gVisor, 2024.

[32] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[33] Arda Aytekin and Mikael Johansson. Harnessing the power of serverless runtimes for large-scale optimization. *arXiv preprint arXiv:1901.03161*, 2019.

[34] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.

[35] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[36] 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*, pages 467–481, 2020.

[37] Lambda Warmer: Optimize AWS Lambda Function Cold Starts, 2018.

[38] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. Ofc: an opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 228–244, 2021.

[39] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa:̂ A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 122–137, 2021.

[40] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and

Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, pages 248–259, 2021.

[41] Hanfei Yu, Hao Wang, Jian Li, and Seung-Jong Park. Harvesting idle resources in serverless computing via reinforcement learning. *arXiv preprint arXiv:2108.12717*, 2021.

[42] 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, pages 386—-400, New York, NY, USA, 2021. Association for Computing Machinery.

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

[44] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. Understanding, predicting and scheduling serverless workloads under partial interference. In *Proceedings of the International conference for high performance computing, networking, storage and analysis*, pages 1–15, 2021.

[45] Jonatan Enes, Roberto R Expósito, and Juan Touriño. Real-time resource scaling platform for big data workloads on serverless environments. *Future Generation Computer Systems*, 105:361–379, 2020.

[46] Xue Li, Peng Kang, Jordan Molone, Wei Wang, and Palden Lama. Kneescale: Efficient resource scaling for serverless computing at the edge. In *2022 22nd IEEE International*

Symposium on Cluster, Cloud and Internet Computing (CCGrid), pages 180–189. IEEE, 2022.

[47] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. λ-nic: Interactive serverless compute on programmable smartnics. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 67–77. IEEE, 2020.

[48] Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy Katz, and Joseph Gonzalez. Kernel-as-a-service: A serverless interface to gpus. *arXiv preprint arXiv:2212.08146*, 2022.

[49] Anubhav Guleria, J Lakshmi, and Chakri Padala. Emf: Disaggregated gpus in datacenters for efficiency, modularity and flexibility. In *2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–8. IEEE, 2019.

[50] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–17, 2021.

[51] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

[52] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10), 2022.

[53] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with

serverless computing: A measurement study. In *Proceedings of the 29th ACM workshop on network and operating systems support for digital audio and video*, pages 61–66, 2019.

[54] Sebastián Risco and Germán Moltó. Gpu-enabled serverless workflows for efficient multimedia processing. *Applied Sciences*, 11(4):1438, 2021.

[55] Ling-Hong Hung, Dimitar Kumanov, Xingzhi Niu, Wes Lloyd, and Ka Yee Yeung. Rapid rna sequencing data analysis using serverless computing. *bioRxiv*, page 576199, 2019.

[56] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 281–295, 2020.

[57] Yuxin Yuan, Xiao Shi, Zhengyu Lei, Xiaohong Wang, and Xiaofang Zhao. Smpi: Scalable serverless mpi computing. In *2022 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 275–282. IEEE, 2022.

[58] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. Fmi: Fast and cheap message passing for serverless functions. In *Proceedings of the 37th International Conference on Supercomputing*, pages 373–385, 2023.

[59] Marcin Copik, Alexandru Calotoiu, Konstantin Taranov, and Torsten Hoefler. Faaskeeper: a blueprint for serverless services. *arXiv preprint arXiv:2203.14859*, 2022.

[60] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[61] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[62] Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. Flexlog: A shared log for stateful serverless computing. 2023.

[63] Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu. λdnn: Achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers*, 71(2):450–463, 2021.

[64] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan Narayanaswamy. Fireplace: Placing firecraker virtual machines with hindsight imitation. *Proceedings of Machine Learning and Systems*, 3:652–663, 2021.

[65] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Practical scheduling for real-world serverless computing. *arXiv preprint arXiv:2111.07226*, 2021.

[66] Mania Abdi, Sam Ginzburg, Charles Lin, Jose M Faleiro, Íñigo Goiri, Gohar Irfan Chaudhry, Ricardo Bianchini, Daniel S. Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. ACM, May 2023.

[67] Gabriel Aumala, Edwin Boza, Luis Ortiz-Avilés, Gustavo Totoy, and Cristina Abad. Beyond load balancing: Package-aware scheduling for serverless platforms. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 282–291. IEEE, 2019.

[68] Youngsoo Lee and Sunghee Choi. A greedy load balancing algorithm on serverless platforms maximizing locality.

[69] Alexander Fuerst and Prateek Sharma. Locality-aware Load-Balancing For Serverless Clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC 2022, New York, NY, USA, 2022. Association for Computing Machinery.

[70] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[71] InfluxDB Time Series Data Platform — InfluxData, 2024.

[72] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX workshop on hot topics in cloud computing (HotCloud 16)*, 2016.

[73] Amazon Cloud. Aws lambda pricing, 2022.

[74] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, volume 2, 2008.

[75] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[76] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[77] Maxime C Cohen, Philipp W Keller, Vahab Mirrokni, and Morteza Zadimoghaddam.

Overcommitment in cloud services: Bin packing with chance constraints. *Management Science*, 65(7):3255–3271, 2019.

[78] The Linux Foundation Projects. Open container initiative, 2015.

[79] FunctionBench. `https://github.com/ddps-lab/serverless-faas-workbench`, 2019.

[80] Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues.

[81] 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*, pages 559–572, 2021.

[82] Xingda Wei, Tianxia Wang, Jinyu Gu, Yuhan Yang, Fangming Lu, Rong Chen, and Haibo Chen. Booting 10k serverless functions within one second via rdma-based remote fork. *arXiv preprint arXiv:2203.10225*, 2022.

[83] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From warm to hot starts: leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 58–64, 2021.

[84] 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*, pages 152–166, 2021.

[85] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. *USENIX ATC*, page 14, 2018.

[86] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[87] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. COSE: Configuring Serverless Functions using Statistical Learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 129–138, July 2020. ISSN: 2641-9874.

[88] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[89] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys*, Jan 2022.

[90] Ali Raza, Ibrahim Matta, Nabeel Akhtar, Vasiliki Kalavri, and Vatche Isahagian. Sok: Function-as-a-service: From an application developer's perspective. *Journal of Systems Research*, 1(1), 2021.

[91] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2020.

[92] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. Survey on serverless computing. *Journal of Cloud Computing*, 10(1):1–29, 2021.

[93] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. A holistic view on resource management in serverless computing environments: Taxonomy, and future directions. *arXiv preprint arXiv:2105.11592*, 2021.

[94] Gabriel Aumala, Edwin Boza, Luis Ortiz-Avilés, Gustavo Totoy, and Cristina Abad. Beyond load balancing: Package-aware scheduling for serverless platforms. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 282–291, 2019.

[95] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 194–204. IEEE, 2021.

[96] Zhiyuan Guo, Zachary Blanco, Mohammad Shahrad, Zerui Wei, Bili Dong, Jinmou Li, Ishaan Pota, Harry Xu, and Yiying Zhang. Decomposing and Executing Serverless Applications as Resource Graphs, December 2022. arXiv:2206.13444 [cs].

[97] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *2021 USENIX Annual Technical Conference (USENIXATC 21)*, pages 805–820, 2021.

[98] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 194–204, July 2021. ISSN: 2575-8411.

[99] 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. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, May 2022.

[100] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. QoS-Aware Resource Management for Multi-phase Serverless Workflows with Aquatope, December 2022. arXiv:2212.13882 [cs].

[101] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 78–93, San Francisco California, November 2022. ACM.

[102] Amoghvarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 19–24, 2019.

[103] Amoghavarsha Suresh and Anshul Gandhi. Servermore: Opportunistic execution of serverless functions in the cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 570–584, 2021.

[104] 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.

[105] Erika Hunhoff, Shazal Irshad, Vijay Thurimella, Ali Tariq, and Eric Rozner. Proactive serverless function resource management. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 61–66, 2020.

[106] Hanfei Yu. *FaaSRank: A Reinforcement Learning Scheduler for Serverless Function-as-a-Service Platforms*. PhD thesis, University of Washington, 2021.

[107] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, pages 356—370, New York, NY, USA, 2020. Association for Computing Machinery.

[108] Bartłomiej Przybylski, Paweł Żuk, and Krzysztof Rzadca. Data-driven scheduling in serverless computing to reduce response time. *arXiv preprint arXiv:2105.03217*, 2021.

[109] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[110] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation*, 64(9-12):1062–1081, 2007.

[111] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Speedo: Fast dispatch and orchestration of serverless workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 585–599, 2021.

[112] 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*, pages 753–767, 2022.

[113] Rohan Basu Roy, Tirthak Patel, Richmond Liew, Yadu Nand Babuji, Ryan Chard, and Devesh Tiwari. Propack: Executing concurrent serverless functions faster and cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 211–224, 2023.

[114] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 symposium on cloud computing*, pages 445–451, 2017.

[115] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos

Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 475–488, 2019.

[116] 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*, pages 263–274, 2018.

[117] Konstantinos Konstantoudakis, David Breitgand, Alexandros Doumanoglou, Nikolaos Zioulis, Avi Weit, Kyriaki Christaki, Petros Drakoulis, Emmanouil Christakis, Dimitrios Zarpalas, and Petros Daras. Serverless streaming for emerging media: towards 5g network-driven cost optimization: A real-time adaptive streaming faas service for small-session-oriented immersive media. *Multimedia Tools and Applications*, pages 1–40, 2022.

[118] Zekun Wang, Pengwei Wang, Peter C Louis, Lee E Wheless, and Yuankai Huo. Wearmask: Fast in-browser face mask detection with serverless edge computing for covid-19. *arXiv preprint arXiv:2101.00784*, 2021.

[119] Unai Elordi, Luis Unzueta, Jon Goenetxea, Estíbaliz Loyo, Ignacio Arganda-Carreras, and Oihana Otaegui. On-demand serverless video surveillance with optimal deployment of deep neural networks. In *VISIGRAPP (4: VISAPP)*, pages 717–723, 2021.

[120] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, pages 1–4, 2016.

[121] Sundar Anand, Annie Johnson, Priyanka Mathikshara, and R Karthik. Low power real time gps tracking enabled with rtos and serverless architecture. In *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, pages 618–623. IEEE, 2019.

[122] Priscilla Benedetti, Mauro Femminella, Gianluca Reali, and Kris Steenhaut. Experimental analysis of the application of serverless computing to iot platforms. *Sensors*, 21(3):928, 2021.

[123] Justin Hu, Ariana Bruno, Brian Ritchken, Brendon Jackson, Mateo Espinosa, Aditya Shah, and Christina Delimitrou. Hivemind: A scalable and serverless coordination control platform for uav swarms. *arXiv preprint arXiv:2002.01419*, 2020.

[124] Razin Farhan Hussain, Mohsen Amini Salehi, and Omid Semiari. Serverless edge computing for green oil and gas industry. In *2019 IEEE Green Technologies Conference (GreenTech)*, pages 1–4. IEEE, 2019.

[125] Muhammed Oguzhan Mete and Tahsin Yomralioglu. Implementation of serverless cloud gis platform for land valuation. *International Journal of Digital Earth*, 14(7):836–850, 2021.

[126] Song Zhang, Xiaochuan Luo, and Eugene Litvinov. Serverless computing for cloud-based power grid emergency generation dispatch. *International Journal of Electrical Power & Energy Systems*, 124:106366, 2021.

[127] Jesse Donkervliet, Javier Ron, Junyan Li, Tiberiu Iancu, Cristina L Abad, and Alexandru Iosup. Servo: Increasing the scalability of modifiable virtual environments using serverless computing.

[128] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[129] Pablo Gimeno Sarroca and Marc Sánchez-Artigas. Mlless: Achieving cost efficiency in serverless machine learning training. *arXiv e-prints*, pages arXiv–2206, 2022.

[130] Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. Serverless computing provides on-demand high performance computing for biomedical research. *arXiv preprint arXiv:1807.11659*, 2018.

[131] Sebastian Werner, Jörn Kuhlenkamp, Markus Klems, Johannes Müller, and Stefan Tai. Serverless big data processing using matrix multiplication as example. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 358–365. IEEE, 2018.

[132] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. USENIX ATC:14, 2018.

[133] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Vadim Sukhomlinov, and Naren Nayak. Agile Cold Starts for Scalable Serverless. *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, page 6, 2019.

[134] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[135] Ping-Min Lin and Alex Glikson. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. *arXiv:1903.12221 [cs]*, March 2019. arXiv: 1903.12221.

[136] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. In Search of a Fast and Efficient Serverless DAG Engine. *arXiv:1910.05896 [cs]*, October 2019. arXiv: 1910.05896.

[137] Bishakh Chandra Ghosh, Sourav Kanti Addya, Nishant Baranwal Somy, Shubha Brata Nath, Sandip Chakraborty, and Soumya K. Ghosh. Caching Techniques to Improve

Latency in Serverless Architectures. *arXiv:1911.07351 [cs]*, November 2019. arXiv: 1911.07351.

[138] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshu Gandhi. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. page 10, 2020.

[139] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[140] Kai Cheng and Yahiko Kambayashi. Lru-sp: a size-adjusted and popularity-aware lru replacement algorithm for web caching. In *Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000*, pages 48–53. IEEE, 2000.

[141] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *USENIX FAST*, volume 3, pages 115–130, 2003.

[142] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[143] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, page 15, 1997.

[144] N. Young. The K-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, June 1994.

[145] Ludmila Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. In *HP Labs Technical Report 98-69 (R.1)*, 1998.

[146] Ludmila Cherkasova and Gianfranco Ciardo. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In G. Goos, J. Hartmanis, J. van Leeuwen,

Bob Hertzberger, Alfons Hoekstra, and Roy Williams, editors, *High-Performance Computing and Networking*, volume 2110, pages 114–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. Series Title: Lecture Notes in Computer Science.

[147] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *International Conference on High-Performance Computing and Networking*, pages 114–123. Springer, 2001.

[148] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An online-model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798. USENIX Association, July 2020.

[149] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 351–364, 2016.

[150] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

[151] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 55–67, 2017.

[152] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.

[153] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield.

Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, 2014.

[154] Google Cloud Functions Tips and Tricks. `https://cloud.google.com/functions/docs/bestpractices/tips`, 2020.

[155] Aws lambda predictable start-up times with provisioned concurrency. `https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/`, Dec 2019.

[156] Azure functions warm-up trigger. `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-warmup`, 2019.

[157] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[158] Neal E Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.

[159] PID controllers. `https://en.wikipedia.org/wiki/PID_controller`.

[160] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 33:1–33:17, New York, NY, USA, 2019. ACM.

[161] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–26, 2012.

[162] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An online-model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference*, pages 785–798, 2020.

[163] Jeongchul Kim and Kyungyong Lee. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, July 2019. ISSN: 2159-6182.

[164] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[165] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive ttl-based caching for content delivery. In *Proceedings of the 2017 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, pages 45–46, 2017.

[166] Bo Jiang, Philippe Nain, and Don Towsley. On the convergence of the ttl approximation for an lru cache under independent stationary request processes. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(4):1–31, 2018.

[167] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11-16):1203–1213, 1999.

[168] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[169] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[170] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 587–604. SIAM, 2018.

[171] John Chen, Benjamin Coleman, and Anshumali Shrivastava. Revisiting consistent hashing with bounded loads. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3976–3983, 2021.

[172] Locust. Locust: A modern load testing framework. `https://locust.io/`.

[173] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[174] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 265–276, New York, NY, USA, 2009. ACM, Association for Computing Machinery.

[175] 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*, pages 1–14, 2022.

[176] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient Request Isolation in FaaS, May 2022. arXiv:2205.11458 [cs].

[177] Apache Kafka: Open Source Distributed Event Streaming Platform. `https://kafka.apache.org/`, 2020.

[178] Prateek Sharma. Challenges and opportunities in sustainable serverless computing. *HotCarbon 2022: 1st Workshop on Sustainable Computer Systems Design and Implementation*.

[179] Paweł Zuk, Bartłomiej Przybylski, and Krzysztof Rzadca. Call Scheduling to Reduce Response Time of a FaaS System. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 172–182, September 2022. ISSN: 2168-9253.

[180] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[181] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for $\mu second-scaletaillatency.In16thUSENIXSymposiumonNetworkedSystemsDesignandImplementation(N-360, 2019$.

[182] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[183] Lucian Popa, Ali Ghodsi, and Ion Stoica. HTTP as the narrow waist of the future internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, Monterey California, October 2010. ACM.

[184] 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*, pages 138–152, 2021.

[185] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized Core-granular

Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 158–164, Santa Cruz CA USA, November 2019. ACM.

[186] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 289–305, San Francisco California, November 2022. ACM.

[187] 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*, pages 559–572, 2021.

[188] Lixiang Ao, George Porter, and Geoffrey M Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 730–746, 2022.

[189] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference*, pages 1–13, Delft Netherlands, December 2020. ACM.

[190] Rodrigo Bruno, Serhii Ivanenko, Sutao Wang, Jovan Stevanovic, and Vojin Jovanovic. Graalvisor: Virtualized polyglot runtime for serverless applications, 2022.

[191] Containerd: An industry standard container runtime. `https://containerd.io/`.

[192] crun: A fast and low-memory footprint OCI Container Runtime fully written in C. `https://github.com/containers/crun`, 2020.

[193] Sashko Ristov, Christian Hollaus, and Mika Hautz. Colder than the warm start and warmer than the cold start! experience the spawn start in faas providers. In *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing*

and Evaluating Algorithms for Distributed Systems, ApPLIED '22, page 35–39, New York, NY, USA, 2022. Association for Computing Machinery.

[194] Yang Richard Yang and Simon S Lam. General aimd congestion control. In *Proceedings 2000 International Conference on Network Protocols*, pages 187–198. IEEE, 2000.

[195] Michael A Bender, Soumen Chakrabarti, and Sambavi Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA*, volume 98, pages 270–279, 1998.

[196] Devin Carraway. Lookbusy – a synthetic load generator. `http://www.devin.com/lookbusy/`.

[197] 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*, pages 1063–1075, Columbus OH USA, October 2019. ACM.

[198] Sebastián Quevedo, Freddy Merchán, Rafael Rivadeneira, and Federico X. Dominguez. Evaluating Apache OpenWhisk - FaaS. In *2019 IEEE Fourth Ecuador Technical Chapters Meeting (ETCM)*, pages 1–5, November 2019.

[199] Joel Scheuner, Simon Eismann, Sacheendra Talluri, Erwin van Eyk, Cristina Abad, Philipp Leitner, and Alexandru Iosup. Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications, May 2022. arXiv:2205.07696 [cs].

[200] Dong Kyoung Kim and Hyun-Gul Roh. Scheduling Containers Rather Than Functions for Function-as-a-Service. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 465–474, May 2021.

[201] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. An evaluation of open source serverless

computing frameworks support at the edge. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642-939X, pages 206–211, 2019.

[202] Tobias Pfandzelter and David Bermbach. tinyFaaS: A Lightweight FaaS Platform for Edge Environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24, Sydney, Australia, April 2020. IEEE.

[203] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation - IoTDI '19*, pages 225–236, Montreal, Quebec, Canada, 2019. ACM Press.

[204] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. Lass: running latency sensitive serverless computations at the edge. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 239–251, 2021.

[205] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 51–62, Storrs, CT, USA, November 2021. IEEE.

[206] Pawel Zuk and Krzysztof Rzadca. Scheduling Methods to Reduce Response Latency of Function as a Service. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 132–140, September 2020. ISSN: 2643-3001.

[207] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. Sfs: Smart os scheduling for serverless functions. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 584–599. IEEE Computer Society, 2022.

[208] Amazon. AWS Lambda Limits.

[209] Johannes Manner, Martin EndreB, Tobias Heckel, and Guido Wirtz. Cold Start Influencing Factors in Function as a Service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, Zurich, December 2018. IEEE.

[210] Diana M Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on gpu virtualization. *Journal of Parallel and Distributed Computing*, 139:32–42, 2020.

[211] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 635–644, 2023.

[212] Kelvin KW Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 595–610, 2023.

[213] Hangchen Yu, Arthur M Peters, Amogh Akshintala, and Christopher J Rossbach. Automatic virtualization of accelerators. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 58–65, 2019.

[214] Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3):1–37, 2017.

[215] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, Atlanta, GA, USA, November 2020. IEEE.

[216] HPC benchmarks for python. `https://github.com/dionhaefner/pyhpc-benchmarks`.

[217] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[218] FFmpeg, 2024.

[219] Aji John, Kristiina Ausmees, Kathleen Muenzen, Catherine Kuhn, and Amanda Tan. SWEEP: Accelerating Scientific Research Through Scalable Serverless Workflows. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion - UCC '19 Companion*, pages 43–50, Auckland, New Zealand, 2019. ACM Press.

[220] Josef Spillner, Cristian Mateos, and David A. Monge. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. In Esteban Mocskos and Sergio Nesmachnow, editors, *High Performance Computing*, volume 796, pages 154–168. Springer International Publishing, Cham, 2018.

[221] NVIDIA Multi-Instance GPU, 2023.

[222] Multi-Process Service :: GPU Deployment and Management Documentation, 2023.

[223] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231. IEEE, 2010.

[224] Hangchen Yu and Christopher J Rossbach. Full virtualization for gpus reconsidered. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2017.

[225] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. 2024.

[226] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16, 2017.

[227] Jiho Kim, John Kim, and Yongjun Park. Navigator: Dynamic multi-kernel scheduling to improve gpu performance. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[228] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J Rossbach. Dgsf: Disaggregated gpus for serverless functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–750. IEEE, 2022.

[229] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. Mitigating cold start problem in serverless computing: A reinforcement learning approach. *IEEE Internet of Things Journal*, 10(5):3917–3927, 2022.

[230] Ana Ebrahimi, Mostafa Ghobaei-Arani, and Hadi Saboohi. Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions. *Journal of Systems Architecture*, page 103115, 2024.

[231] Bowen Yan, Heran Gao, Heng Wu, Wenbo Zhang, Lei Hua, and Tao Huang. Hermes: Efficient cache management for container-based serverless computing. In *12th Asia-Pacific Symposium on Internetware*, pages 136–145, 2020.

[232] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. Multi-Queue fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, 2019.

[233] Alexander Fuerst, Abdul Rehman, and Prateek Sharma. Ilúvatar: A fast control plane for serverless computing. 2023.

[234] Nvidia management library. https://developer.nvidia.com/nvidia-management-library-nvml.

[235] Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2017.

[236] How much device memory per device context creation. https://forums.developer.nvidia.com/t/predictable-how-much-device-memory-per-device-context-creation/41983/4, 2016.

[237] Chun-Xun Lin, Tsung-Wei Huang, and Martin DF Wong. An efficient work-stealing scheduler for task dependency graph. In *2020 IEEE 26th international conference on parallel and distributed systems (ICPADS)*, pages 64–71. IEEE, 2020.

[238] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 341–342, 2010.

[239] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[240] Qinqin Tang, Renchao Xie, Fei Richard Yu, Tianjiao Chen, Ran Zhang, Tao Huang, and Yunjie Liu. Distributed task scheduling in serverless edge computing networks for the internet of things: A learning approach. *IEEE Internet of Things Journal*, 9(20):19634–19648, 2022.

[241] Kevin Exton and Maria Read. Raptor: Distributed scheduling for serverless functions. *arXiv preprint arXiv:2403.16457*, 2024.

[242] Samuel Ginzburg, Mohammad Shahrad, Michael J Freedman, Zhaoduo Wen, Sidharth Kumar, Binyu Zang, Ken Gordon, Xiaochuan Tang, Balaji Vembu, Zbigniew T Kalbarczyk,

et al. VectorVisor: A binary translation scheme for Throughput-OrientedGPU acceleration. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1017–1037, 2023.

[243] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[244] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 119–134, New York, NY, USA, 2023. Association for Computing Machinery.

[245] Apacha Hadoop, 2024.

[246] Open MPI: Open Source High Performance Computing, 2024.

[247] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 44–54, 2019.

[248] Seong-Joong Kim, Myoungsung You, Byung Joon Kim, and Seungwon Shin. Cryonics: Trustworthy function-as-a-service using snapshot-based enclaves. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 528–543, 2023.

[249] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable enclaves for confidential serverless computing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4015–4032, 2023.

[250] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[251] Jianqiang Wang, Pouya Mahmoody, Ferdinand Brasser, Patrick Jauernig, Ahmad-Reza Sadeghi, Donghui Yu, Dahan Pan, and Yuanyuan Zhang. Virtee: A full backward-compatible tee with native live migration and secure i/o. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 241–246, 2022.

[252] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[253] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. HyperEnclave: An open and cross-platform trusted execution environment. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 437–454, 2022.

[254] Best Ever Blueberry Cookies - OMG Chocolate Desserts, 2024.

CURRICULUM VITAE

Ingredients [254].

- $1\ \frac{1}{2}$ cups flour
- 1 teaspoon corn starch
- $\frac{1}{4}$ teaspoon salt
- $\frac{3}{4}$ teaspoons baking powder
- $\frac{1}{2}$ cup unsalted butter
- $\frac{3}{4}$ cup sugar
- 1 egg
- 1 teaspoon vanilla
- 4 oz. white chocolate chips
- $\frac{3}{4}$ cup frozen blueberries

Instructions.

- In a small bowl whisk together dry ingredients: flour, corn starch, salt and baking powder, set aside.
- Cream butter and sugar on high speed for about 2 minutes, until light and creamy. Add egg and vanilla and mix to combine.
- Running your mixer on low, mix in dry ingredients mix.
- Fold in white chocolate chunks and blueberries, gently.
- The dough will be thick and sticky.
- Cover and refrigerate for 2 hours.
- When ready to bake preheat the oven to 350 F and line baking sheets with parchment paper.
- Scoop one heaping tablespoon of dough and roll gently to make a ball.
- Arrange cookie balls onto baking sheet leaving 2 inches apart, because the cookies will spread while baking.
- Bake 16-18 minutes, until edges start to brown.
- Cool on baking sheet for 5 minutes, then transfer on a rack to cool completely.

Alex Fuerst

(440) - 669 - 5865
fuersta.2013@gmail.com | linkedin.com/in/alex-fuerst | github.com/aFuerst | afuerst.github.io

OBJECTIVE

Systems engineer with several years' development experience, and expertise in low-level virtualization, operating systems, and high-level distributed systems, who can analyze and communicate findings to general audiences. Looking to tackle the inefficiencies and problems of modern computing in cloud-scale environments.

EDUCATION

Computer Engineering, PhD — Expected May 2024
Indiana University, Intelligent Systems Engineering — Major GPA 3.9

Computer Engineering, MS — Dec 2023
Indiana University, Intelligent Systems Engineering — Major GPA 3.9

Computer Science, Bachelor of Science — May 2017
Xavier University — Major GPA 3.6

Diploma with Honors — May 2013
Medina High School — GPA 3.6

PUBLICATIONS

5. **Alexander Fuerst**, Abdul Rehman, and Prateek Sharma. Ilúvatar : A Fast Control Plane for Serverless Computing. *International ACM Symposium on High-Performance Parallel and Distributed Computing [HPDC] 2023*. Acceptance Rate = 21%

4. **Alexander Fuerst**, and Prateek Sharma. Locality-aware Load-Balancing For Serverless Clusters. *International ACM Symposium on High-Performance Parallel and Distributed Computing [HPDC] 2022*. Acceptance Rate = 19%

3. **Alexander Fuerst**, Stanko Novakovic, Inigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-Harvesting VMs in Cloud Platforms. *International Conference on Architectural Support for Programming Languages and Operating Systems [ASPLOS] 2022*. Acceptance Rate = 20%

2. **Alexander Fuerst**, and Prateek Sharma. FaasCache: Keeping Serverless Computing Alive With Greedy-Dual Caching. *International Conference on Architectural Support for Programming Languages and Operating Systems [ASPLOS] 2021*. Acceptance Rate = 18.8%

1. **Alexander Fuerst**, Ahmed Ali-Eldin, Prashant Shenoy, and Prateek Sharma. Cloud-scale VM-deflation for Running Interactive Applications On Transient Servers. *International ACM Symposium on High-Performance Parallel and Distributed Computing [HPDC] 2020*. Acceptance Rate = 22%

EXPERIENCE

Google, Inc. — *Mountain View, California*
Software Engineering Intern — May 2023 - August 2023

- Delved into performance of advanced Linux and KVM-based virtualization technologies under cloud workloads
- Explored techniques to seamlessly upgrade VMM and hypervisor with zero downtime to guest OS and applications
- Modified Linux kernel, KVM, and Cloud Hypervisor VMM to test possibilities for seamless upgrade
- Developed proof-of-concept experiments to show feasibility of designed techniques

Microsoft Research — *Redmond, Washington*
Research Intern — May 2021 - August 2021

- Analyzed modern hypervisor and control plane's performance under strenuous runtime conditions
- Modified Azure control plane, hypervisor, and guest OS to improve cluster virtual memory management

- Collaborated with Azure team to alleviate production contentions and plan rollout across clusters
- Improved resource utilization by 20% in Azure without impact to hosted VMs or applications

Indiana University *Bloomington, Indiana*
Research Assistant August 2021 - Present

- Performed advanced research in cloud resource management, serverless computing, and virtualization
- Developed novel techniques and algorithms improving resource utilization and latency in cloud control planes
- Designed experiments to showcase techniques' effectiveness and transform results into actionable data
- Published several first author papers and gave presentations at high-impact conferences

Indiana University *Bloomington, Indiana*
Associate Instructor August 2019 - May 2020

- Assisted with Engineering Cloud Computing & Distributed Computing Engineering course work
- Created assignments and exams given to students
- Hosted lab and office hours to discuss project design and assist with student questions

Hyland Software *Westlake, Ohio*
Developer 1 & 2 June 2017 - July 2019

- Developed features and wrote tests for a cloud application capable of handling thousands of daily users
- Troubleshot complex issues of a multi-service .Net SaaS application running in production
- Refactored monolith application to run as microservices and support autoscaling inside Kubernetes
- Modernized application CI/CD pipeline to improve time-to-deployment for features and enable rollback

Xavier University *Cincinnati, Ohio*
Teaching Assistant August 2016 - December 2016

- Work with students during class exercises
- Host office hours answering questions and giving guidance on assignments

Xavier University *Cincinnati, Ohio*
Student Technician Tier II August 2013 - August 2017

- Troubleshoot complex technology issues and provide onsite service and repair for faculty, staff and public computing
- Provide software, hardware and network problem resolution
- Handle tickets escalated from Tier I

Salisbury University *Salisbury, Maryland*
NSF REU Researcher May 2016 - August 2016

- Applied emerging parallel computing models using GPU and CPU parallelism with NVIDIA's CUDA
- Tackled data and compute-intensive problems in geographic information systems
- Presented findings to GIS professionals and Salisbury Faculty

Critchfield, Critchfield & Johnston, Ltd. *Medina, Ohio*
Paralegal Intern August 2012 - June 2013

- Prepared and delivered documents to county offices
- Finalized legal binders for delivery to clients

PROJECTS

Ilúvatar FaaS Control Plane

An open-source, fast, jitter-free control plane for Serverless function execution written in ∼23k lines of Rust. Ilúvatar provides a 3x reduction in overhead compared to popular open-sourced examples under normal load, and under high load has a 100x reduction in p99 latency. Additionally, it enables unique usability and extensibility designed to accelerate FaaS research.

FaasCache

Introduced caching insights into the Function-As-A-Service paradigm. Enhanced the open source FaaS application OpenWhisk using Greedy-Dual caching. Reduced cold-start overhead for functions by up to 3x and can reduce constrained system resources by up to 30%. These high cache reuse results allow for increased ability to serve functions and lower latency for users.

CompuCell3D Tissue Modeling Parallel Rendering

CompuCell3D is a 3D modeling software for large-scale cellular, tissue, and biochemical simulation. The modeling steps used an OpenMP Cellular Potts Model algorithm, but the 3D rendering of cell states and positions was done serially. This project re-wrote the rendering code in ∼1000 lines of OpenMP C, achieving a near-linear scaling with the increase in threads. Overall, some simulations were accelerated by up to 50% over the serial implementation.

Dynamically Typed Racket Compiler

A scratch built compiler supporting a subset of statically typed and dynamically typed Racket.

Tensorflow NanoParticle Simulator

Implementation of the Lennard-Jones potential in a simulated cube and electrostatic forces of colliding ions in a confined nano-channel. THe simulator Achieved performance similar to MPI/C++ code performing the same simulation.

Jae OS

Just Another Educational Operating System. A port of the Kaya OS project to the new μARM emulator. Wrote the student guide and the canonical implementation of Jae OS.

Kaya OS

Wrote a complete operating system from scratch. The final product, in addition to support a multitude of peripheral devices, successfully ran eight concurrent processes, each running in their own virtual address space.

Parallel GIS Raster Calculator

Developed a tool combining CPU based parallelism and NVIDIA's CUDA technology for GPU calculation for performing GIS raster calculations. Achieved 2 - 5 times performance increase over traditional analysis tools due to GPU performance.

Eagle Scout Project

Installed commemorative plaques on veterans' graves at local cemetery. Led a group of 15 scouts to plan and accomplish this project.

SKILLS

Programming	Debugging, Problem Solving, Code Optimization, Git, Agile
Languages	Rust, Python, C, C++, C#, Bash, PowerShell, SQL, LaTeX
Infrastructure	Kubernetes, Docker, Redis, Octopus Deploy, Ansible, AWS, Azure
Technologies	Linux, KVM, GPUs, GDB, OpenMP, MPI, Tensorflow, SQL Server

PRESENTATIONS

- Ilúvatar : A Fast Control Plane for Serverless Computing. HPDC 2023. Slides
- Locality-aware Load-Balancing For Serverless Clusters. HPDC 2022. Slides Video
- Memory-Harvesting VMs in Cloud Platforms. ASPLOS 2022. Slides Video
- FaasCache: Keeping Serverless Computing Alive With Greedy-Dual Caching. ASPLOS 2021. Slides Video
- Cloud-scale VM-deflation for Running Interactive Applications On Transient Servers. HPDC 2020. Slides Video

AWARDS

Reserve Champion, Baked Goods Division	Monroe County Fair 2023
HPDC Travel Grant	Travel grant to HPDC 2023
ACM Travel Grant	Travel grant to ASPLOS 2022
John F. Niehaus Scholarship	Xavier University
John F. Niehaus Award	Xavier University
Eagle Scout	Boy Scouts of America
National Honors Society	Medina High School
National Technical Honors Society	Medina County Career Center

COURSE WORK

- Engineering Cloud Computing
- Engineering Distributed Systems
- Graph Analytics
- Deep Learning Systems
- Engineering Compilers
- Programming Languages
- Engineering Operating System
- Simulating Nanoscale Systems
- High Performance Computing
- Computational Modeling for Virtual Tissues
- Databases

ACTIVITIES

Parish Pastoral Council, St. Paul's Catholic Center	Member, 2023-2024
Young Adult Ministry, St. Paul's Catholic Center	Member, 2022-2023
	Vice Chair, 2023-2024
Computer Science Club, Xavier University	Treasurer, 2015-2017
	Vice President, 2016-2017
Dean's Advisory Council, Xavier University	Member, 2015 – 2017