UNIVERSITY OF CALIFORNIA SAN DIEGO

Framework and Platform Support for General Purpose Serverless Computing

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Lixiang Ao

Committee in charge:

Professor George Porter, Co-Chair
Professor Geoffrey M. Voelker, Co-Chair
Professor Alin Deutsch
Professor Patrick Pannuto
Professor George Papen

2022

University of California San Diego

2022

DEDICATION

Dedicated to my family, for their unconditional love and unwavering support through the years.

EPIGRAPH

Two things awe me most,
the starry sky above me
and the moral law within me.

Immanuel Kant

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First I am extremely grateful to my advisors Geoff Voelker and George Porter. I have learned invaluable lessons from them on both research and life. My PhD journey at UCSD would not be possible without Geoff. His attention to detail has taught me dedication. His optimism and patience have helped me through trying times. His wisdom and passion have navigated me through uncharted waters. After five years of working with him, I am still amazed how he can provide insightful answers to almost every question I ask. George has been encouraging and supportive in many ways. His intriguing proposals and unique ideas have greatly inspired me on my research projects. From him I learned the power of conveying complex ideas using just simple words.

I would like to thank the rest of my thesis committee members, Alin Deutsch, Pat Pannuto, and George Papen, for their thought-provoking questions and in-depth discussions during my thesis.

The collaborative and supportive atmosphere at SysNet group has made my PhD a much smoother ride. I would like to thank Aaron Schulman, Stefan Savage, Alex Snoeren, Yuanyuan Zhou, and Yiying Zhang for their guidance and advice. I am grateful to Cindy Moore, our system administrator. Cindy has provided tremendous help whenever I need to set up servers for research projects and classes, a privilege that not every system research group enjoys.

I would like to thank my labmates in the Foundry and the rest of SysNet, especially Vector, Shelby, Nishant, Rajdeep, Liz, Evan, Chengcheng, Yudong, Bingyu, Yizhou, and Zesen. I am fortunate to have some of the smartest and most approachable fellow graduate students around, from the day I first arrived in UCSD till the day of my final defense, through every paper acceptance and rejection, and in every celebration at BJ's. They have made my PhD experience a memorable one.

Last but not least, I would like to thank my parents and my wife Yuanyuan Yu. They have always believed in me and supported me. They have taught me to understand each other and care for each other, and showed me what it means to be a family.

Chapter 2, in full, is a reprint of the material as it appears in the Proceedings of the 9th ACM Symposium on Cloud Computing. “Sprocket: A Serverless Video Processing Framework.” Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the 11th ACM Symposium on Cloud Computing. “Particle: Ephemeral Endpoints for Serverless Networking.” Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. The dissertation author was the co-primary investigator and author of this paper.

Chapter 4, in full, is currently being prepared for submission for publication of the material “Sophon: Efficient Container-grained Serverless Scheduling.” Lixiang Ao, George Porter, and Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this material.

Chapter 5, in full, is a reprint of the material as it appears in Proceedings of the 17th European Conference on Computer Systems. “FaaSnap: FaaS Made Fast Using Snapshot-based VMs.” Lixiang Ao, George Porter, and Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this paper.

VITA

2009–2013	Bachelor of Engineering, University of Electronic Science and Technology of China
2013–2016	Master of Engineering, University of Electronic Science and Technology of China
2016–2022	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Lixiang Ao, George Porter, Geoffrey M. Voelker: FaaSnap: Faas Made Fast Using Snapshot-based VMs. ACM EuroSys 2022.
Lixiang Ao, George Porter, Geoffrey M. Voelker: Sophon: Efficient Container-grained Serverless Scheduling. *In submission.*
Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, George Porter: Sprocket: A Serverless Video Processing Framework. ACM SoCC 2018: pp. 263-274.
Shelby Thomas, **Lixiang Ao**, Geoffrey M. Voelker, George Porter: Particle: Ephemeral Endpoints for Serverless Networking. ACM SoCC 2020: pp. 16-29.
Landon Cox, **Lixiang Ao**: LevelUp: A thin-cloud approach to game livestreaming. ACM SEC 2020: pp. 246-256.
Sandeep D'souza, Victor Bahl, **Lixiang Ao**, and Landon P. Cox. "Amadeus: Scalable, Privacy-Preserving Live Video Analytics." arXiv preprint arXiv:2011.05163 (2020).
Li, Zhiying, Ruini Xue, and **Lixiang Ao**: Replichard: Towards Tradeoff between Consistency and Performance for Metadata. ICS 2016: p.25
Ruini Xue, **Lixiang Ao**, Zhongyang Guan: COMET: Client-Oriented Metadata Service for Highly Available Distributed File Systems. SBAC-PAD 2015: 157-164
Ruini Xue, Shengli Gao, **Lixiang Ao**, Zhongyang Guan: BOLAS: Bipartite-Graph Oriented Locality-Aware Scheduling for MapReduce Tasks. ISPDC 2015: 37-45
Ruini Xue, **Lixiang Ao**, Shengli Gao, Zhongyang Guan, Lupeng Lian: Partitioner: A Distributed HDFS Metadata Server Cluster. CyberC 2014: 167-174
Ruini Xue, Zhongyang Guan, Shengli Gao, **Lixiang Ao**: NM2H: Design and Implementation of NoSQL Extension for HDFS Metadata Management. CSE 2014: 1282-1289

ABSTRACT OF THE DISSERTATION

Framework and Platform Support for General Purpose Serverless Computing

by

Lixiang Ao

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor George Porter, Co-Chair
Professor Geoffrey M. Voelker, Co-Chair

Serverless computing is a new computing paradigm that aims to simplify building cloud applications. Existing serverless computing is designed for certain types of workloads, such as stateless web request handlers that require little coordination across instances. In this thesis I explore the potential of general purpose serverless computing by providing new features at both the application framework level and the serverless platform level.

I first describe Sprocket, a parallel video application framework based on serverless computing. Based on my work with Sprocket, I observed several key limits of existing serverless offerings, including lack of cross-instance networking, inefficient resource scheduling, and the cold start problem. I develop three systems for the serverless platform: Particle, Sophon, and FaaSnap, that tackle networking, scheduling, and cold start challenges, respectively. These systems reduce the gap between the requirements of emerging serverless applications and the capabilities of current serverless offerings, paving the way for future general purpose serverless computing.

Chapter 1

Introduction

Cloud computing has transformed the computing landscape. According to Castro et al. [15], 67 percent of enterprise computing spending is on the cloud. Compared to traditional on-premise infrastructures, cloud computing simplifies infrastructure management, increases capacity, reduces the capital expenses, and potentially reduces overall costs. The advantages of cloud computing come from two reasons. First, users do not purchase any hardware resources, and the cloud resources are usually shared among cloud tenants, which reduce capital expenses and lowers the resource price. Second, cloud providers operate at large scale, which enables extremely large scaling and capacity. This economy of scale of the cloud helps reduce costs for both providers and users.

The IaaS (Infrustructure-as-a-Service) model is the most popular form of cloud computing for many enterprises and businesses. In IaaS, hardware resources are virtualized and provided as a service to the users. Cloud vendors use hypervisors to multiplex resources like CPU, memory, network and storage to multiple users. The users in effect rent resources like virtual machines, virtual networks, and virtual disks, and use their own operating systems, runtimes, and libraries to host their applications. IaaS is a simple form of low-level resource offering, yet it works well in many scenarios. About 33 percent of all cloud revenue comes from IaaS [75]. The success of the IaaS model can be attributed to several reasons. First, the simple resource interface allows for straightforward transitioning from traditional on-premise infrastructures to the cloud, which reduces the cost for the users. Second, the low-level hardware interface is largely standardized, making it easy to switch between cloud vendors, which helps users avoid vendor lock-ins and facilitates competition among cloud vendors.

The only difference between IaaS and traditional on-premise infrastructures is the ownership of hardware resources. In IaaS, users no longer need to own the hardware. On the other hand, the operating systems, runtimes, libraries, and applications stay the same. Although the simple interface of IaaS makes the transitioning to the cloud easy, it imposes several limits and disadvantages.

The first limit of IaaS is that, although users no longer need to worry about hardware, they still need to manage networking and storage devices, operating systems, language runtimes, libraries, etc. This software management overhead is not negligible. The second limit is that hardware resource allocation is coarse-grained, which prohibits users from making more precise resource allocation. For example, even when a virtual machine's CPU is 10% utilized, the user needs to pay for the full price for renting the CPU time. The third limit is that, by using low-level interfaces, the cloud vendors are unable to provide some high-level features like fine-grained auto-scaling, layer-7 load-balancing, distributed consensus, etc., because they lack application-level semantics.

Serverless computing

In early 2015, Amazon Web Services (AWS) introduced Lambda [11], a "serverless" cloud computing offering. In Lambda, users do not manage servers. Instead, they provide application code in the form of "functions", which will be invoked when events like web requests or API calls happen. Users do not manage servers, runtimes, storage, or the network. Because of the "function" interface, it is also called function-as-a-service (FaaS).

The function instances are stateless, meaning there is no persistent state across requests. Each instance only handles one request. This approach allows easy autoscaling. The platform simply creates as many concurrent instances as concurrent requests. Fine-grained autoscaling makes it easy to handle workloads that fluctuate. Compared to IaaS, FaaS alleviates users from the overheads of managing servers and other resources like operating systems, libraries, and networking. The cloud provider owns and manages these resources and keeps them up to date.

Serverless has a fine-grained pay-as-you-go billing model. It only charges the CPU time spent on executing actual application code. The billing model is attractive to many users whose application workloads fluctuate over time: in IaaS users tend to be conservative in workload estimation and reserve more resources to prevent service degradation due to under-provisioning. But this often leads to wasted resources: users are billed for idle CPU cycles. In serverless, users usually do not need to worry about resource provisioning since resources are fine-grained and auto-scaled according to the current workload. Compared to IaaS, serverless instances can usually be launched in less than a second so that it can react to fluctuations in load in almost real time. IaaS virtual machines, however, take tens of seconds to minutes to launch and shutdown. These features make serverless a good fit for **request handler** applications like web request handlers or IoT event handlers.

Most serverless offerings are intended for request handlers. Request handlers process one request at a time, and each request is isolated from other requests. Serverless instances do not need to talk to each other, and there is no need for coordination across instances. Besides request handler applications, however, there is a growing interest in extending serverless computing to a more general purpose model. ExCamera [31], PyWren [44], and gg [30] are distributed video encoding, parallel tasks, and compiling applications on serverless. In these applications, multiple instances are involved for handling a request, and data transfer across instances are needed. In addition, there is usually a need for coordination across instances. This type of application is in stark contrast to *request handler* applications. In this dissertation I use the term **general purpose serverless**, or GPSL, to refer to this type of application.

GPSL benefits from serverless's pay-as-you-go billing model as well as the fine-grained auto-scaling capability. Users can launch thousands of serverless functions instantly, allowing the application to handle sudden bursts of workloads.

However, existing serverless computing offerings impose several limitations, and these limitations are restrictive for GPSL. Serverless intrinsically eliminates the notion of addressable servers, which is a reasonable simplification for event handling programs since an event processor does not need an external network address, but it can be inconvenient for GPSL. These applications usually require sizable amounts of data to be sent between the instances; for example, in MapReduce applications, large data transfers often occur in the "shuffle" phase. Without addressable servers, the instances are unable to directly talk to each other. Some data processing systems work around this limitation by using intermediate stores or caches to let nodes intercommunicate indirectly. However, these approaches are sub-optimal since data has to traverse much more network and storage paths in the process.

In serverless, users do not need to keep idle servers running as in IaaS. Instead, when requests arrive, underlying environments need to be set up to handle the request. Two types of delays can happen in this scenario, scheduling delays and cold start delays. The delays will slow down performance and reduce responsiveness of applications, especially for GPSL applications that rely on many concurrent instances, and the slow "stragglers" instances can become performance bottlenecks.

Thesis statement

In this thesis, I will demonstrate that application framework level and platform level features greatly simplify implementation, reduce network overheads, and improve scheduling and cold start performance of GPSL computing. I first describe Sprocket, a serverless application framework for building parallel video processing pipelines. I identify what common building blocks are needed for developing such complex applications on serverless. I observe the potential benefits of building GPSL applications. I also describe a few problems of existing serverless offerings, which include lack of direct networking, inefficient resource scheduling, and cold start problems. I then describe Particle, lightweight networking for serverless, Sophon, container-grained scheduling, and FaaSnap, a snapshot-based virtual machine for fast cold start to solve the problems, respectively.

Sprocket

Serverless is able to provide massive parallelism in a short amount of time thanks to the stateless function abstraction. This is a good match for parallel data processing systems where large amounts of CPU resources are required. Existing cloud-based parallel data processing systems are based on IaaS, where the users use a cluster of virtual machines to process large amounts of data. This architecture suffers from the inherent drawbacks of IaaS: users have to manage the cluster of machines, the network, and storage layers, which increases the operational costs. Users also need to pay for CPU time even when the virtual machines are idle, and this can be particularly challenging when the workload fluctuates a lot. The users need to reserve extra resources so that the system can handle high workloads. When the workload is low, it is hard to downsize the resources since the virtual machines can take several minutes to shutdown. Users end up paying for resources they are not using.

Serverless can be a good replacement for IaaS in this scenario. When there is no request, no serverless instances would be used. And when requests arrive, serverless instances can be created on-demand to process the requests. However, although the processing power is available, processing large-scale data in parallel is not the original intended use case for serverless. It was unclear *whether* parallel processing applications should be implemented in serverless, and *how* these they should be implemented.

Sprocket is one answer to these questions. In Sprocket, I focus on parallel video processing applications. Video is a fast-growing data form that accounts for more than 70 percent of Internet traffic [17]. Video data can be a good match for parallel serverless computing since video data has intrinsic parallelism that can be exploited. However, many problems remain unclear, including handling input/output/intermediate data, the concurrency model, the programming interface, representing the computation dependencies, etc.

In Sprocket, video applications are modeled as video pipelines. Each pipeline consists of multiple stages, e.g., decoding, filtering, or encoding stages. The stages are connected using streams, which define the data dependencies between the stages. The stages and streams form a logical directed acyclic graph (DAG), similar to a dataflow system. The pipeline is specified using a JSON-based domain-specific language (DSL) which we call "pipespec".

When a pipeline starts, input data and the pipespec is provided to Sprocket. Then the logical DAG is expanded to a physical DAG, which defines inputs, outputs, and dependencies between serverless instances. The physical DAG is materialized by Sprocket by launching serverless instances and orchestrating the instances. A central controller orchestrates the execution of the instances according to the predefined logic for each stage. Data transfer between instances is implemented by reading and writing via an external object store (S3).

Using "pipespec", users can easily construct video pipelines, from simple ones like video filtering, to complex pipelines like facial recognition. Sprocket implements these pipelines in serverless computing and utilizes the high parallelism provided by the platform. The autoscaling feature of serverless allows Sprocket to launch serverless instances as much as exactly needed with low latency, which avoids resource idling and wait time.

Using the parallelism of serverless, Sprocket can process hours of video in seconds without needing to maintain a dedicated cluster. I propose several techniques to solve problems including handling data dependencies, task scheduling, and mitigating stragglers. The feasibility of running parallel video processing workloads on serverless also shed light on other GPSL applications that can potentially be implemented on serverless.

Observations from Sprocket

With my experience from Sprocket, I observed which features current serverless platforms have that support GPSL computing well, and which features the platforms lack. Filling the gap between them two can potentially enable more GPSL applications.

There are several features current serverless platforms offer that are attractive to GPSL computing, including:

1. **High-level "function" interface.** The function interface abstracts away many low-level details like servers, operating systems, virtual networks, and storage. This interface simplifies the development and operation for Sprocket and other GPSL applications.

2. **Pay-as-you-go billing model.** Serverless users are only billed for the time Sprocket instances spent on actual processing. Sprocket does not need to worry about resource utilization since it is handled by the serverless platform. This saves operational costs for Sprocket and other GPSL applications.

3. **Fine-grained auto-scaling.** Sprocket is able to "right-size" its resources to the exact amount needed because of fine-grained auto-scaling of serverless. This feature is valuable for any application whose workload fluctuates a lot including Sprocket and other GPSL applications.

I also observe some problems that are not solved in today's serverless offerings yet. Solving these problems can further accommodate future GPSL computing:

1. **Intercommunication between serverless instances.** Sprocket and other distributed applications have data dependencies between instances. Intercommunication between them is crucial. Current serverless platforms do not support intercommunication well. In Sprocket, intercommunication between instances is implemented by storing data temporarily on an external storage layer, which is neither efficient nor cost-effective.

2. **Efficient resource scheduling.** In Sprocket I noticed some large delays for launching serverless instances, ranging from several seconds to tens of seconds. These delays come from inefficient serverless resource scheduling, i.e, allocating hardware and software resources to serverless applications. The delays can lead to obvious performance degradation for Sprocket and other GPSL applications.

3. **Fast cold starts.** Cold start is an important topic in serverless. A cold start is needed when the environments of a serverless instance, including the operating system, language

runtime, etc., needs to be loaded or created from scratch. The platform needs to set up the environment when receiving a request, as opposed to a warm start, where the environment is ready to accept requests. Cold starts usually impose delays from less than a seconds to tens of seconds. Similar to scheduling inefficiencies, cold starts can cause large delay and performance degradation for Sprocket and GPSL computing.

Based on these observations, I propose three systems: Particle, Sophon, and FaaSnap, and each system focuses on one of the aforementioned problems. Compared to Sprocket, which focuses on the application level, these three systems support GPSL by improving the serverless platforms.

Particle

Existing function interfaces for serverless offerings do not provide full network access. For example, AWS Lambda provides network access for serverless instances via a NAT (network address translation) interface, allowing functions to initiate network connections. However, it does not support accepting incoming connections. This simple network capability works for applications like request handlers since their use of the network usually involves reading inputs from external locations and writing outputs to external locations.

However, for Sprocket and other GPSL computing, this network capability is restrictive: since there is no direct network connection, data transfers between serverless instances requires writing to and reading from remote storage or an external relay server, which is both inefficient and costly. In Sprocket I found that, for video processing workloads, about one thirds of total processing time was spent on reading and writing external data.

What intercommunication interface is best for GPSL is still an open question. The platform can provide higher level interfaces like a key-value store or pub-sub interfaces. It can also provide networked file system interfaces. In Particle, I focus on providing traditional IP network interfaces since other higher level interfaces can be implemented on top of it.

In a multi-tenant cloud environment, an overlay network is used to provide virtual networks. Containers, a popular isolation mechanism for multi-tenant clouds, need network namespaces and virtual network devices to support multiple overlay network endpoints on the same host. Creating multiple such network stacks, however, creates significant delays in the Linux kernel. When simultaneously starting 100 serverless instances and setting up an overlay network for them, the network setup delay accounts for more than half of the total execution time.

In Particle I reconsider the relationship between applications and the network abstraction. A lightweight network abstraction is used to reduce the overheads of creating network namespaces and virtual devices. Serverless instances of the same application share the same network namespace and virtual device. Experimental results show Particle greatly reduces network setup time. The total time for a burst-parallel serverless application improves by 2x to 17x.

The significance of Particle is two fold: it enables direct network connectivity between serverless instances, and it allows the instances to be connected efficiently. This is important to GPSL applications since direct network connectivity is needed, and lower setup time improves the responsiveness of those applications.

Sophon

The high-level function interface of serverless abstracts away the underlying low-level environment like operating systems and language runtimes. The management of the low level environments, however, is the responsibility of the serverless platform. Setting up the environment for an incoming serverless request is challenging since the setup needs to be conducted quickly to reduce request latency, an issue known as the cold start problem.

Since a cold start is costly and slow, after a serverless request is served, the platform does not terminate the environment immediately. Instead, the environment is kept warm for a short amount of time. If another request arrives during this time, the existing environment can be used, resulting in a warm start. A warm start has nearly no wait time besides the network delay of the request, which is usually less than 10ms. A cold start, on the other hand, takes up to tens of seconds.

Although warm starts have huge performance advantages, keeping environments warm imposes resource consumption, especially on memory. Serverless platforms usually only keep warm environments 5-30 min after a request is finished [32]. A key problem in serverless resource management is to make scheduling decisions to make use of warm environments as much as possible while keeping resource usage low.

I used real-world traces of serverless platforms to evaluate OpenWhisk, a popular open-source serverless system, and find performance inefficiencies in its current scheduling design. In OpenWhisk, the function environment is a container. The default scheduler in OpenWhisk does not keep track of container states. Instead, it schedules function invocations to machines that are likely to have warm containers using a sticky-random algorithm. I found that this algorithm works well when the workload is low and there is no bursty workload. However, when the workload is high or there is a burst of invocations, the sticky-random algorithm produces suboptimal scheduling decisions.

When the workload is high or bursty, resource contention occurs. Containers for different functions contend for memory resources while evicting each other's containers, which creates more cold starts and further increases resource contention. I use the term "container thrashing" to refer to this phenomenon for its resemblance to memory thrashing.

Based on these insights, I propose Sophon, container-grained scheduling, and implement Sophon in OpenWhisk. Sophon keeps track of container states for each function. With container states, the scheduler is able to make scheduling decisions based on more detailed information. Warm containers are always used when there is a warm container available, and when cold starts are needed, the scheduler makes sure the cold start has minimal impact on system performance by choosing the best machine for a cold start.

I develop several scheduling policies on the basis of Sophon. These policies use several metrics, including available memory and a cost metric, which is a metric of potential costs of evictions. Different policies are used for cold and warm starts. I test the policies using a real-world serverless trace and compare the results against the vanilla OpenWhisk scheduler and other non-container-grained schedulers. Results show that compared to OpenWhisk, Sophon improves system throughput by 76%, reduces the number of cold starts by 40%, and reduces average invocation latency by 200 ms.

Sophon reduces invocation latency, which improves the responsiveness of serverless applications. This improvement is important for many GPSL applications. Sophon improves system capacity by reducing unnecessary cold starts, which eventually reduces operational costs of GPSL applications.

FaaSnap

In a multi-tenant cloud environment, all user code is run in sandboxes and isolated from other users. One of the most common sandboxing mechanisms is virtual machines, or VMs. The Firecracker VM from Amazon, for example, is a lightweight VM that is tailored for fast booting of serverless workloads. However, besides booting, cold start involves other time consuming steps like starting the language runtime, installing libraries and application code, and warming up application states. Together these steps can take a long time, slowing the cold start process.

VM snapshots have been proposed to solve the environment initialization problem [26, 88]. Pre-initialized states can be saved to persistent storage and restored to memory when a request arrives. With snapshots, initialization steps can be skipped to reduce cold start time. Existing snapshot-based systems, however, can suffer from three problems: long initial prefetching blocking, inefficiency in handling changing working sets, and a semantic gap between the guest and host.

I propose FaaSnap, a fast snapshot-based VM for FaaS. FaaSnap uses several techniques to solve these problems. Concurrent paging allows the guest to start as soon as the basic VM state is restored and avoids the initial prefetching blocking. Host page recording and per-region mapping make FaaSnap able to tolerate changes in the working set. Per-region mapping also narrows the semantic gap between the guest and host, avoiding unnecessary slow disk reads.

Experimental results show FaaSnap solves these inefficiencies in existing snapshot-based VMs. It can achieve performance close to snapshots cached in memory: on average the function execution time is only 3.5% slower than that of cached snapshots. FaaSnap handles changing working sets much better than existing systems. High concurrency tests and remote disk tests demonstrate FaaSnap can be applied to a broad range of applications and scenarios.

GPSL computing can benefit from the fast cold start of FaaSnap. Moreover, FaaSnap supports burst-parallelism workloads, a pattern that is common in GPSL computing including Sprocket while not well supported in existing serverless offerings. FaaSnap brings more possibilities to GPSL computing.

Summary

Although initially proposed to host request handling workloads, features like massive parallelism and a pay-as-you-go model of serverless computing makes it possible to host general purpose workloads. General purpose serverless (GPSL) can further revolutionize the application landscape of cloud computing. Sprocket explores the potentials of GPSL. From my experience with Sprocket, I noticed several limitations of existing serverless offering that hinders the full realization of GPSL. I therefore proposed three new systems Particle, Sophon, and FaaSnap to mitigate limitations in cross-instance network connectivity, resource scheduling, and cold starts. These systems make GPSL more efficient from both the application framework level and platform level. In the following chapters, I will describe Sprocket, Particle, Sophon, and FaaSnap in detail.

Chapter 2

Sprocket: A Serverless Video Processing Framework

The rise of serverless computing has drawn interests in both academia and industry. General purpose serverless, or GPSL, is a promising model that extends simple request handler-style serverless to a broader range of applications. In this chapter, I propose Sprocket as an instance of GPSL for video processing.

Sprocket enables developers to program a series of pipelined operations over video content in a modular, extensible manner. By composing operations, developers are able to build custom processing pipelines whose elements can be reused and shared. Sprocket handles the underlying access, loading, encoding and decoding, and movement of video and image content across operations in a highly parallel manner on serverless platforms. The goal with Sprocket is to not only support traditional video processing operations and transformations, such as transcoding to lower resolutions or applying filters, but also to enable much more sophisticated video processing applications, such as answering queries of the form, "Show just the scenes in the movie in which Wonder Woman appears".

Massive parallel, low latency video processing of Sprocket is enabled by serverless platforms, which can allocate and use thousands of instances at millisecond timescales. Furthermore, providers offer these instances with a billing model that also charges at sub-second time scales. Running a Sprocket application that can process a video with 1,000-way concurrency using AWS

Lambda on a full-length HD movie costs about $3 per hour of processed video.

For the remainder of this chapter I start by providing background on video data properties and challenges. I then describe the design and implementation of Sprocket, in particular the pipeline programming model and support for data streaming and worker scheduling. Finally, I evaluate Sprocket's performance from various perspectives, including where it spends its time executing pipeline operations, how it exploits parallelism and intrinsic properties of the processing video, how it meets streaming deadlines when generating output, the performance characteristics of a complex application, and how it compares to other general processing frameworks.

2.1 Background

Video data is one of the predominant forms of data in the world: 70% of consumer Internet traffic is compressed video content [17]. Video streaming service providers like YouTube and Netflix, and social networks like Facebook and Instagram, receive, edit, encode, and stream multiple TBs of video data every day. Processing video data, however, presents unique challenges due to its unique properties.

As 4K and VR videos become more common, the size of video files increase rapidly: 4K videos typically have a bit-rate of over 30 Mbps, and a 2-hour-long movie can be 30 GB in size. Even applying a simple transformation on these videos can take hours using a single machine. In production environments, the video processing pipeline can become very complicated. Some large-scale applications require hundreds of tasks to be executed. The efficient scheduling and execution of video processing pipelines becomes a challenging task in these systems. Sprocket's use of a serverless framework allows for scheduling and processing to happen seamlessly and efficiently.

Working with the individual frames of a video is also no trivial matter. Since video encoding takes advantage of both spatial and temporal similarity for compression, individual

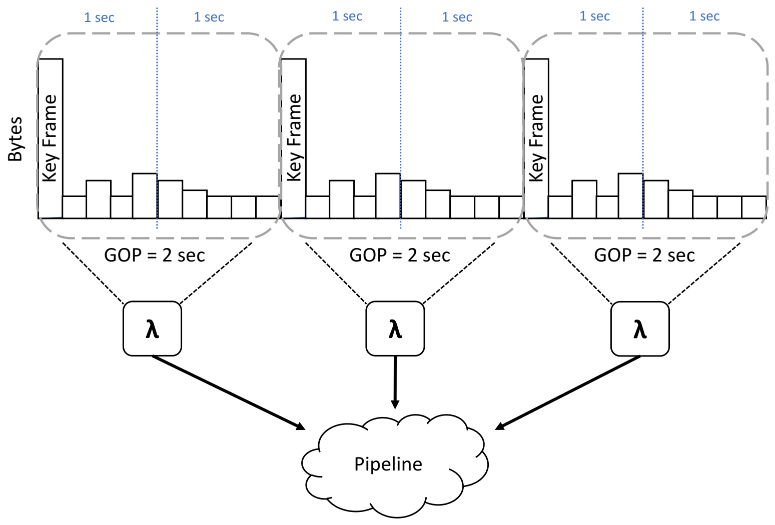


Figure 2.1. Frame sizes in a GOP.

frames become dependent on other frames. The encoder thus inserts "keyframes" to allow for groups of pictures (GOP) to be independent of each other. This structure, however, results in individual frames being non-uniform in size, as shown in Figure 2.1, and thus can greatly affect the behavior of sending, storing, and computing on the data. Accessing frames within a GOP requires at least partial decoding of that segment of the video, since all but the initial frame in the GOP are encoded as differences from that primary image. As we will show, Sprocket not only handles this decode operation on behalf of application pipelines, but also uses the encoded video to enable its approach to straggler mitigation, described in Section 2.2.6.

The content of the video itself can also greatly affect the behavior of any video-processing system. For example, a system that is running face detection will inherently take more time on a video that has many faces, as opposed to no faces. Furthermore, some Sprocket pipelines contain data dependencies, in addition to higher-level control dependencies, that impose unique requirements on its scheduling approach, as described in the next section. Sprocket takes the content of the video into account by dynamically dedicating more serverless computing resources when processing certain scenes, as described in Section 2.2.5.

Recent work has also specifically focused on video processing applications. ExCamera is a highly parallel video encoder that encodes small chunks of video in Lambda threads in parallel, ultimately stitching them together into a single final video file [31] (indeed, experience with Lambdas developing ExCamera convinced us of their viability for Sprocket). Other video analytic systems include VideoStorm, which focuses on querying characteristics of streamed live video content and the resource, quality, and delay tradeoffs of scheduling large numbers of queries on a cluster [94]. The Streaming Video Engine is Facebook's framework for processing all user-supplied video content at scale, with a particular focus on reducing the latency of ingesting and re-encoding video content so that it can be shared with other users [40].

2.2 Sprocket design

Before describing the design of Sprocket, we first highlight two example scenarios of its use. First, imagine that a user has a video on their laptop that they are editing, and they wish to see a version of the video with a remapped set of colors. They invoke, either via a command-line tool or through an interface in their editing program, a Sprocket-based filter tool with the new color mapping. The video is uploaded to the cloud, and Sprocket applies the filter to the video and they are able to stream the updated version in their browser within a few seconds.

In a second example, imagine that a user is watching a two-hour long video of a school play, and wants to only watch the portions starring their child. They visit a web page, backed by Sprocket, that lets them submit a URL of the video, along with a URL of their child's face, and within a few seconds they are able to stream an abbreviated version of the play in their browser featuring only those scenes starring their child.

2.2.1 Overview

Sprocket is a scalable video processing framework designed to transform a single video input according to a user-specified program. Users write these programs in a domain-specific language, described in Section 2.2.7, which is expressed as a dataflow graph. The dataflow graph consists of vertices and edges, similar to other dataflow systems such as Tez [85]. Edges (which we also refer to as *streams*) convey data between vertices, in particular individual frames of video, groups of frames, or compressed segments of consecutive frames we call *chunks*. Vertices execute individual functions, specified by the user, on data that arrives at their input(s), emitting any resulting frames or chunks to the next part of the DAG via one or more output edges. We refer to a single DAG program as a *pipeline*, and refer to vertices in that DAG as pipeline *stages*.

The modular design of the pipeline and pipeline specification allows developers to build complex video processing systems with little effort. The simplicity of the interactions with the Lambda infrastructure greatly mitigate the management overhead.

In this section, we describe a set of example applications built on the Sprocket framework, highlighting underlying stage implementations used to build those applications. Developers can use these stages to build custom pipelines, and can also implement new stages to extend this set of functionality.

2.2.2 Application 1: Video filter

Our first application example is a pipeline that applies an image processing filter, drawn from those supported by the `FFmpeg` tool, to an entire video (Figure 2.2). This pipeline consists of three stages: Decode, Filter, and Encode. An input, which can be in the format of a video link (YouTube, Vimeo, etc.), a cloud-local file hosted on S3, or uploaded by the user, will be sent to multiple Lambda workers in parallel. Each worker will decode a chunk of video into a set of sequential frames. The Lambda worker then invokes `FFmpeg` on each frame. Lastly, each transformed frame will be encoded into an output format, either a single compressed file (e.g., via the Ex.Camera distributed encoder) or to multiple fixed-length chunks suitable for streaming via the MPEG-DASH [23] format.

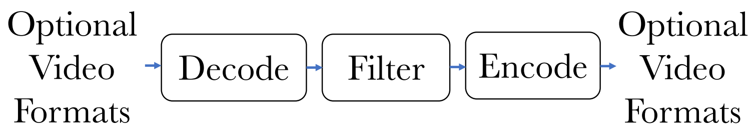


Figure 2.2. Logical overview of the Video Filter pipeline.

Stage design

Sprocket will initially evaluate the input video type and handle initiating the parallel nature of the pipeline accordingly. For example, for a video URL input, Sprocket will broadcast the link to each of many parallel download-and-decode workers, along with video metadata and the number of frames assigned to each worker, so that those workers can download and begin processing the input video in parallel.

This Sprocket pipeline consists of three stages:

Decode: This stage decodes a specified chunk of input video into individual frames (in PNG format). Decode receives video metadata as input, along with the timestamp of where in the video to begin decoding, and how many frames to output to the downstream worker. After processing, the decode stage emits the decoded frames to the S3 intermediate storage system.

Filter: The Filter stage applies the `FFmpeg` binary to a chunk of frames. Filter is spawned directly after the Decode stage and receives metadata along with references to the location of the frames stored on intermediate storage. Filter collects the frames from S3 and applies one of its internal filters as specified in the pipeline's pipespec configuration file, described later in Section 2.2.7.

Encode: This stage is responsible for encoding frames. Encode also uses `FFmpeg`, running in a different Lambda worker, which receives metadata along with references to the location of the frames in S3. Encode collects the frames from S3, encoding them using the specified encoder format, and finally writes them either in MPEG-DASH format or a single compressed output file generated by ExCamera. The final result is stored in S3.

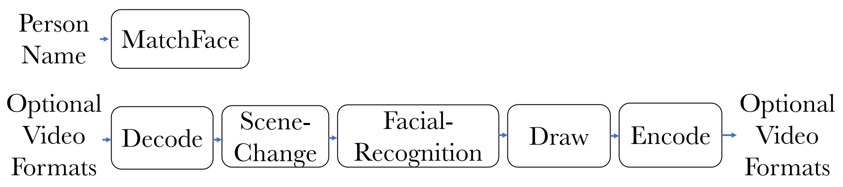


Figure 2.3. Logical overview of Facial Recognition pipeline stages.

2.2.3 Application 2: Facial Recognition

A second pipeline we describe implements facial recognition, which demonstrates a more sophisticated set of operations, including calling out to other cloud services to implement the facial recognition support (Figure 2.3). This pipeline takes an actor's name and a reference video URL as input, and draws a box around the given actor's face in all scenes of the video. This application could support the theatrical production motivating scenario presented earlier.

Stage design

The Facial Recognition pipeline consists of six stages: MatchFace, Decode, SceneChange, FacialRecognition, Draw, and Encode (Figure 2.3). At the beginning of the pipeline, an actor's name, provided by the user, will be used to locate candidate images from a Web search for later use in facial recognition. In parallel, the provided input video URL is fetched and decoded into fixed-length chunks of frames, currently one second each. Workers in parallel will then run a scene change algorithm on each chunk of frames to bin them into separate scenes. For each set of chunks grouped in scenes, a worker will then run a facial recognition algorithm to determine if the target face is present in that scene. If a face is identified in the scene, the chunk of frames will have a bounding box drawn on all the frames at the appropriate position returned by the vision algorithm API, which is then sent downstream to the Encode stage. If no face is detected, then the group of chunks will be sent directly to Encode.

Along with the stages described in the previous section, the following stages are used in the Facial Recognition pipeline:

MatchFace: The MatchFace stage searches for a target image for the face of a person whose name is specified as a parameter. Sprocket currently uses Amazon's Rekognition API [76], but could also use other service offerings: Microsoft offers a computer vision API as part of its Cognitive Services cloud offering [65], and Google offers a cloud-hosted vision system for labeling and understanding images through its Google Cloud Vision API [34].

MatchFace invokes one of these third-party image search services (in our case Amazon Rekognition) to find the top-*k* images returned given the provided name. The stage then iterates through the returned images and runs a face detection algorithm, via external API call, to determine whether the chosen target image contains a face. The first image to pass the facial detection algorithm becomes the selected target image. MatchFace then stores the selected image in S3 for eventual use by downstream stages. Unlike other stages, MatchFace itself does not emit any data directly to downstream stages.

SceneChange: The SceneChange stage detects scene changes in a set of decoded frames. It is invoked after the Decode stage, and is sent a reference to the decoded frames stored in S3. SceneChange collects the frames from S3 and, after detecting the scene change offsets (using an algorithm internal to `FFmpeg`), emits an event containing a list of these references to frames stored in S3, paired with a boolean value marking which frames serve as the boundaries of the scene change.

FacialRecognition: The FacialRecognition stage detects if a group of frames contains a target face (e.g., of the provided actor). The FacialRecognition stage is spawned once for every group of frames that make up one scene. We chose this design, rather than running on every frame in the scene, due to rate limits when invoking third-party recognition algorithms. FacialRecognition downloads the frames from S3 and calls the facial recognition algorithm once on every *n* frames in the scene. The facial recognition algorithm returns whether or not the target image was detected in the frame, and a bounding box of the identified face in the original frame. If at least one frame in the scene is found to contain the target face, all frames in the scene are marked as having the target face. The stage then emits an event containing a list of references to the frame in S3 paired with a bounding box of the identified face. If no target face is identified, FacialRecognition emits a list of frame references paired with empty bounding boxes.

Draw: The Draw stage draws a box at an arbitrary position in the frame, in this case provided as a bounding box around a recognized face. Draw is instantiated from the Facial-Recognition stage and only continues if a scene was labeled as containing a recognized face. Otherwise, draw automatically skips to its final state and sends references to the frames in S3 directly to the downstream stage. Draw only uses the dimensions from one bounded box to draw the same bounding box on all frames. Therefore, Draw assumes that there is little movement of faces in a single scene. We leave as future work interpolating the position of the box based on sampled points throughout the scene. Draw writes the new frames to S3 and emits an event containing a list of frame references.

An alternative version of a Facial Recognition pipeline can also choose to emit scenes that only contain the recognized face. In this case, the Draw stage would be replaced by a SceneKeep stage that only emits references to frames if a face is recognized. Otherwise, the stage will emit an empty list and those particular frames will never be encoded. SceneKeep would be employed to implement the version of the pipeline that edits out all scenes of a theatrical production that do not include a given actor (e.g., the user's child).

Calling external cloud services

Calling an external API to run a facial recognition algorithm creates different challenges for Sprocket. The first is the additonal latency of calling the external service that is beyond Sprocket's control. The second is that Sprocket may encounter an API call throughput limit that slows down the execution time of stages. This execution latency increases as the parallelism of the pipeline increases, since more concurrent calls create a faster overload of the external API. To address this, Sprocket does two things. First, the facial recognition pipeline includes the scene detection stage, reducing the number of calls to the facial recognition API within one scene. Second, the pipeline has the option to use the streaming scheduler, described in more detail in Section 2.3.3, which adaptively calculates the minimum amount of Lambdas needed

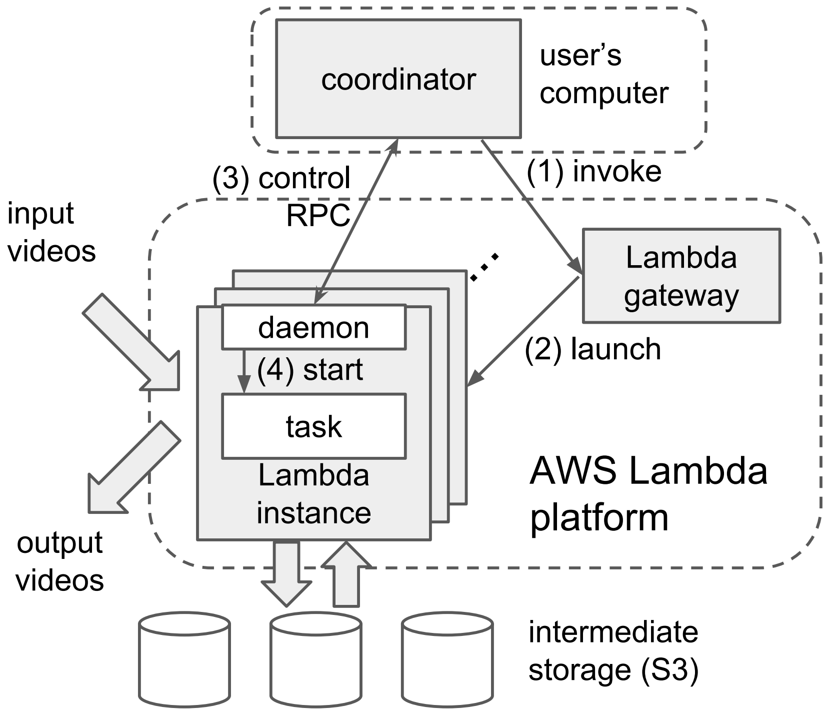


Figure 2.4. Sprocket overview

to meet a streaming deadline throughout execution. In this way, Sprocket limits the amount of concurrent API calls needed, thus avoiding API call throughput limits.

2.2.4 Invoking and managing Lambdas

The Sprocket Coordinator manages the system-wide "control plane," managing the life-cycles of Lambda workers including spawning and tearing them down, and orchestrating the flow of data between workers via the intermediate data storage system. We implemented the Sprocket coordinator in Python, which runs either on a VM in the cloud or on the user's computer. Amazon has since deployed *Step Functions* [83], which is a Lambda workflow management system that serves a similar purpose as our coordinator. Currently AWS Step Functions cannot implement Sprocket pipelines, since support for dynamic levels of parallelism in Step Functions is currently limited. It can scale the number of Lambdas based on an overall workload, but it is not evident how to scale individual stages based on dynamic load or resource optimizations at that stage. Figure 2.4 shows a detailed overview of the main components of our Python-based coordinator design.

The coordinator maps the execution of one task within the Sprocket dataflow graph to a Lambda worker. An individual Lambda worker is invoked by receiving data on its inputs via the delivery function, which comes either from external input sources or from upstream workers via the intermediate storage system. In the AWS cloud, Lambda instances cannot establish direct network connections between each other, and so we convey all data from upstream workers to downstream workers through the intermediate store, in particular through S3. This approach has the added advantage of enabling the scheduler to decouple the execution lifetimes of upstream and downstream stages, enabling them to exist at disparate times. Although the rather heavy-weight S3 storage interface may seem like a potential performance bottleneck, Jonas et al. [44] have shown that it can scale to support thousands of clients without imposing significant performance penalties.

We indirectly control Lambda workers via a modified version of Mu [31], a framework for managing parallel execution of Lambda instances. The Coordinator sends invocation requests to the AWS API gateway to manage Lambda functions. Once the Lambda instance is started, it spawns a local daemon within the Lambda that establishes a connection back to the Coordinator so that the Coordinator can communicate with and manage them. Each Lambda worker is implemented as a state machine, and the particular state machine employed is sent through this interface. In this way, an individual Lambda does not yet know what role it will play in the overall pipeline until it has connected back to the coordinator to request further instructions. This approach has the potential benefit of reducing variance and stragglers.

The Coordinator and the Lambda workers interact with each other using asynchronous RPC. The Coordinator gives commands to the Lambda workers while the workers reply with status reports. The Coordinator treats the status reports as inputs to its internal DAG dataflow, enabling it to generate new states to send to Lambda workers to further process the overall pipeline. If the status report from a worker indicates an error in executing the most recent command, the coordinator can re-send the command to the worker or potentially spawn a new worker to continue the execution of that task.

We extend Mu in several ways to support Sprocket's design. Instead of batch-style invocation of Lambdas in Mu, we support dynamic creation of tasks during processing, and asynchronous messaging and data transfer among the Lambda workers. Moreover, Sprocket abstracts away from any particular serverless platform and can easily support other platforms such as Google Cloud Functions [33].

2.2.5 Pipeline scheduling

Managing on-demand Lambdas

Sprocket's dataflow scheduler relies on the stateless nature of AWS Lambda functions. Generally, a cluster or job scheduler will allocate a large set of resources in advance, and then manage that pool of resources over time across a large number of jobs. For example, Mesos [39] manages resources among frameworks, with each having their own scheduler to further manage resources among applications and tasks. Because Sprocket is targeting a serverless cloud environment, the Sprocket scheduler can allocate and deallocate Lambda resources on fine-grained timescales, rather than all at once at system start time. The cloud provider is then responsible for managing that pool of resources.

The primary resource limit that Sprocket faces is a concurrency limit imposed by the provider, which bounds per account the number of Lambda workers that can run concurrently, as well as the rate at which new Lambda workers can be spawned. In this work, we focus on applying Sprocket to a single instance of a single pipeline, and so assume that it can freely use resources up to the account-wide concurrency limit.

When the demand for concurrent workers is within the concurrency limit set by the provider, scheduling is straightforward: any requested task will immediately trigger an invocation of a worker and begin its processing after the worker is launched. We have observed that the vast majority of Lambda workers launch with subsecond latency. In terms of responsiveness, we therefore treat the workers as a homogeneous resource.

When the demand for Lambda workers exceeds the concurrency limit, such as when processing a long video or processing a very deep pipeline, some tasks have to be deferred and scheduled later. Clients can specify which scheduling policy to use, either optimizing for the lowest overall execution time, or optimizing to prioritize earlier frames for streaming results. The "overall time" strategy minimizes the overall job time by increasing the parallelism of each stage and making sure the task is utilizing the most available Lambda workers. The "time to first frame" strategy identifies the tasks that are on the critical path of the eventual output's frame presentation time (the time at which an output frame will be displayed), and assigns these tasks higher scheduling priorities.

Optimizing for streaming

Sprocket's ability to seamlessly allocate and deallocate resources also allows it to change its scheduling behavior in real time, based on the pattern of the current job. Concretely, Sprocket's streaming scheduler continuously keeps track of the amount of time it is taking to process the current frames to determine if it will meet the streaming deadline. If Sprocket is currently ahead of schedule, the streaming scheduler will speculatively put the current pipeline to sleep for the number of seconds nearly equivalent to the difference between the streaming deadline and the current execution time. Putting the executing pipeline to sleep not only optimizes for use of minimum Lambda resources, as the number of new Lambda invocations drops down to zero for the current pipeline, but also lends itself well to achieving load balancing so a simultaneous different pipeline can be run on sprocket. Section 2.3.3 evaluates Sprocket's streaming performance.

Supporting complex dependencies

Depending on the complexity of the implementation of a vertex in a given dataflow DAG, it is not always straightforward for the scheduler to know when to execute that stage. Indeed, there may exist dependencies across a stage's input that depend on other parts of the

DAG. To capture these dependencies, Sprocket defines a *delivery function* for every stage which specifies the dependencies on its inputs. It is the responsibility of the Sprocket framework, not the implementer of the stage, to manage and satisfy these dependencies. Managing the dependencies and scheduling of inputs to continuous query and dataflow systems is a well-studied problem in a number of domains, and Sprocket's delivery functions might benefit from further development (e.g., "moments of symmetry" and "synchronization barriers" as described by Avnur and Hellerstein [10]).

In a video pipeline, the most basic dependency is one-to-one: the output from an upstream worker is only processed by a single downstream worker, and the downstream worker only needs the output from a single upstream worker. An example of this dependency is a pipeline that takes a chunk of video as an input, decodes that chunk into a sequence of individual frames, applies a filter to that frame set (e.g., color modification), and then encodes the frames back into a compressed chunk. In this simple example, a downstream worker only needs one input event: once an input is ready, the task can be run immediately.

Customizing delivery functions

An example of a pipeline with complex dependencies is the blend application which superimposes the content of two videos.This blend application might serve as one component of a multi-step "green screen" application which replaces regions of an image of a certain color with a separate live video stream. This application includes two parallel decode stages, each of which converts the input video into individual frames. The next stage is a "blend" node, which needs to overlay each frame of one video with the corresponding frame from the other. An example of this pipeline can be seen in Figure 2.5.

For the blend application to work, it needs simultaneous access to corresponding frames. We define a *pair delivery function* that is used to capture this dependency. The "pair" delivery function ensures that frame number *i* from one video is delivered to the blend stage simultaneously with frame *i* from the other video. In other words, the delivery function has to wait for both

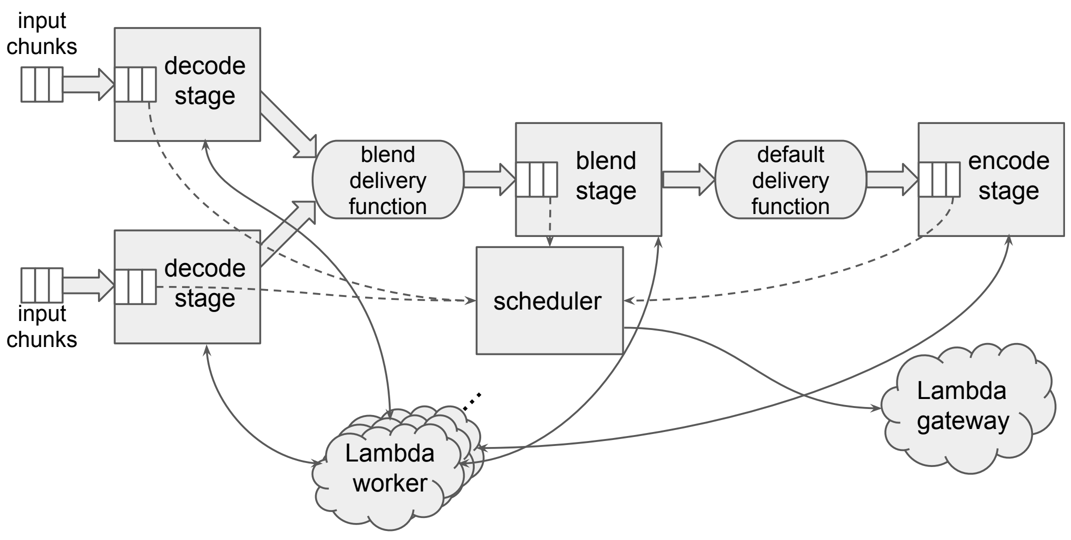


Figure 2.5. A detailed view of the Sprocket Coordinator for the blend pipeline

upstream stages to have generated a set of continuous frames bearing the same presentation timestamps (“pts”), which are used to match up the frames from the two input sources. Once such a pairing occurs, the delivery function will deliver the merged input to the downstream worker as an atomic unit to the encode stage. The encode stage then gathers a set of frames before encoding them into a single compressed output video. A delivery function specific to the encoder ensures that it receives the correct number of frames and ensures that they are consecutive and in order.

Delivery functions can also be used to mitigate challenges of complex pipelines. For example, the Facial Recognition pipeline can incur new scheduling challenges based on the properties of the input data. Stragglers might arise, not just based on performance variation within the cloud platform (e.g., a slow Lambda worker), but rather as a data-dependent result of whether or not a given frame contains a face, as has been reported in other contexts [54]. Scenes with recognized faces must eventually go through the Draw stage, thereby unavoidably taking a longer time to complete. Customizing different delivery functions can help address the recognized face straggler problem.

One way to mitigate the recognized face stragglers is to create a delivery function that has the capability to split up the delivery of downstream events based on cut-off markers provided by the upstream stage. With this design, the FacialRecognition stage is able to request the next downstream stage, Draw, to receive only one frame per worker. By dedicating one Lambda worker per frame, the Draw stage completes almost instantly and takes minimal overhead compared to the rest of the pipeline. Other techniques for straggler [6, 93] and skew mitigation [4, 53, 55], drawn from the database and distributed systems literature, could be employed as well.

Partition function comparison

The delivery function serves a similar role as partition functions in Hadoop and Spark, which determine how the system re-distributes data from upstream to downstream elements, ensuring that the partitioning satisfies data dependency requirements, including avoiding skew in the workload distribution. However, the main difference between Sprocket delivery functions and these other partition functions is that, in Hadoop and Spark, they are serialization points in the pipeline: the respective worker will have to wait for all the input data to be ready to apply a partition function to further send the resulting data downstream. The delivery functions in Sprocket, however, deliver whatever work is available, which is particularly useful when processing video. As long as there exists *some* inputs to a stage that satisfy the data dependency requirements, the delivery function will ensure that the downstream worker is invoked with work to do.

2.2.6 Straggler mitigation

In a pipeline that consists of a large number of workers, stragglers are likely to happen. For example, I/O, CPU scheduling, hardware malfunction and non-deterministic programs can significantly slow down one or more workers. For video applications, it is common for a single worker to have a "wide dependency" of subsequent frames, i.e., many downstream tasks depending on that worker's output, thus stalling the entire pipeline. Such events can directly impact user experiences, such as forcing the user to wait for the streaming of a particular chunk.

Speculative execution [24, 93] is widely used to tackle stragglers. When the framework detects which workers are slower, it duplicates the worker's tasks to other nodes. However, by the time a straggler is detected and the speculative task is launched, it may already be too late to mitigate the straggler. To deal with this problem, proactively cloning tasks [5] can be used. However, this approach requires sending extra copies of input data to the workers and may potentially cause resource contention.

In Sprocket, we combine the two approaches, while avoiding the disadvantages of both, by exploiting the characteristics of input video chunks. Concretely, with straggler mitigation, we choose to use input video chunks that decode into a group of pictures (GOP) that is twice as long than the usual chosen length (e.g., a two-second chunk instead of the usual one-second chunk). Sprocket then sends both seconds of the chunk to the worker dedicated to processing the first second of the GOP, and the worker dedicated to processing the second second of the GOP. We call both workers a "pair". A worker first processes its assigned part of the GOP. After it finishes, it checks if the other worker in the pair is finished. If not, it continues to work on the other worker's part of the GOP, which in effect becomes speculative execution. Because the other worker's data is already there, speculative execution can be conducted efficiently.

Although workers in a pair have identical data, we are not doubling the total amount of data transferred. As seen in Figure 2.1, the second half of a GOP is smaller in size due to the absence of the key frame. We calculate that, in total, the extra data sent is less than a quarter of the original data.

2.2.7 Programming Sprocket applications

Sprocket programmers construct pipelines using a domain-specific pipeline specification language. In a pipeline specification ("pipespec"), a user first specifies the set of stages making up the pipeline, and then they specify the directed edges that connect those stages. Each edge must connect an upstream endpoint to a downstream endpoint, except for input/output endpoints that represent external sources and sinks of video to the pipeline.

The pipespec allows for individual stages to be parameterized with stage-specific parameters (e.g., target bit-rate parameter of an encoding stage), system-wide parameters (e.g., which Lambda function should be used to implement a stage), and additional data dependency information (Section 2.2.5). Similarly, edges can be parameterized to provide context and directives, e.g., how to convey data through the intermediate storage system to downstream stages or optimizations for interacting with stable storage such as S3.

Below is an example pipespec for a sample "blend" application, described in Section 2.2.5 which superimposes the content of two videos.

```
{ "nodes:
  [ {
      "name": "decode_0",
      "stage": "decode"
    }, {
      "name": "decode_1",
      "stage": "decode"
    }, {
      "name": "blend",
      "stage": "blend",
      "delivery_function":
        "pair_delivery_func"
    }, {
      "name": "encode",
      "stage": "encode"
    }
  ],
  "streams":
  [ {
      "src": "input_0:chunks",
      "dst": "decode_0:chunks"
    }, {
      "src": "input_1:chunks",
      "dst": "decode_1:chunks"
    }, {
      "src": "decode_0:frames",
      "dst": "blend:frames_0"
    }, {
      "src": "decode_1:frames",
      "dst": "blend:frames_1"
    }, {
      "src": "blend:frames",
      "dst": "encode:frames"
    }, {
      "src": "encode:chunks",
      "dst": "output_0:chunks"
    }
  ]
}
```

Listing 2.1. The Blend pipespec example. Note that input_0 and output_0 are bound to runtime parameters.

Edges include a source and a destination which specify upstream and downstream endpoints, respectively. An endpoint is defined by a stage name and an edge identifier within the stage, separated by a colon. In this way a stage can specify multiple streams that connect to it, providing broadcast/multicast semantics. The prefix of the edge identifier also implicitly indicates the data type of the stream: only matched data types can be sent through it, and the source and destination types have to match.

2.3 Evaluation

In this section we evaluate Sprocket's behavior from various perspectives, including where it spends its time executing pipeline operations, how it exploits parallelism, how it meets streaming deadlines when generating output, the performance characteristics of a complex application, and how it compares to other general processing frameworks. We execute Sprocket

Table 2.1. Details of input videos used in the experiments.

Name	Length	FPS	Content	Resolution
EARTH [28]	10 hours	25	Nature	1080p
SYNTHETIC EARTH	Variable	25	Uniform 1 second chunks from Earth	1080p
SINTEL [80]	14 min 48 sec	24	Animation/Action	1080p
TEARS OF STEEL [84]	12 min 14 sec	24	Action/Science Fiction	1080p
NATURE [67]	60 sec	25	Trailer/Contains no faces	720p
AVENGERS TRAILER [9]	35 sec	24	Trailer/Action/Contains faces	720p
INTERVIEW [18]	45 sec	24	Interview/Contains faces	720p

using the Lambda service in AWS's North Virginia region (`us-east-1`). The Lambda instances use 3GB of memory, and the Coordinator runs on an AWS EC2 `c5.xlarge` instance. Table 2.1 shows the properties of the test videos we use.

In general, the results we report in this section use "warm" Lambda instances, which do not include the delays for creating a new container, initializing the runtime, etc. The benefits of warm Lambdas are well known in the community (e.g., [90]) because they reduce Lambda creation delays considerably: in our experience, the time to invoke 1,000 cold Lambdas is on the order of 6 seconds, while the time to invoke 1,000 warm Lambdas is an order of magnitude less at 600 ms. We report results with warm Lambdas because we expect them to be a reasonably common case; our experience is that warm Lambdas are cached for at least 15 mins, and often much longer (if a VM has at least one active instance, [90] reports 27 mins, and 1–3 hours).

2.3.1 Time breakdown

First we show where Sprocket workers spend their time when executing pipeline operations. Figure 2.6 shows the execution times for 734 workers performing a simple operation on the TEARS OF STEEL video (Table 2.1). Each worker operates on a one-second chunk of video, reading and decoding it in one stage, converting it from color to grayscale using `FFmpeg` in a second stage, and then encoding and writing it back to S3 as the final stage. Each bar in the graph shows the total execution time for the worker, and the bar breaks down the time into processing (encode, grayscale, decode), accessing video chunks from storage (each stage reads

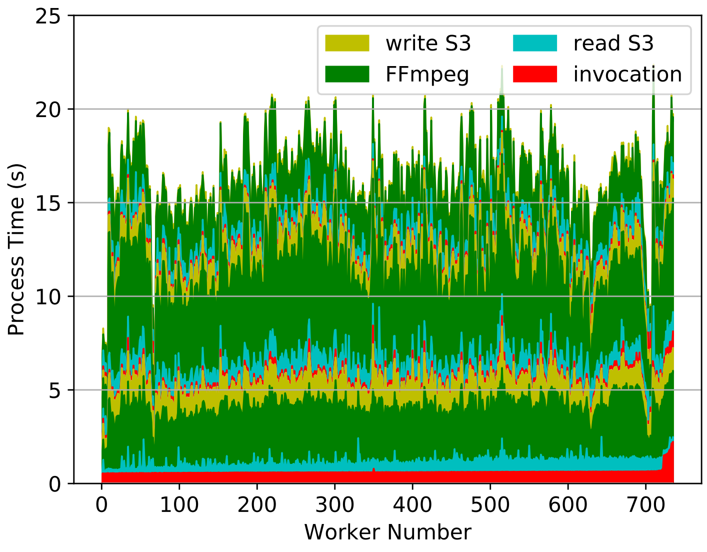


Figure 2.6. Execution time breakdown running the grayscale pipeline on the TEARS OF STEEL video.

and writes to S3), and any waiting time.

For this pipeline, overall execution time is split roughly evenly as follows: across all of the workers, on average 34.7% of time is spent encoding and decoding, 30.4% is spent grayscaling, and 27.6% of time is spent reading and writing to S3, while the remaining 7.3% is spent waiting for the creation and initialization of the workers. There is much variation among workers, but the variation is primarily due to the nature of the video input: chunks of video that contain less data (are more compressible) take less time to process and read/write to S3.

2.3.2 Burst parallelism

A key benefit of using serverless cloud infrastructure is the opportunity for inexpensive burst parallelism. For short periods of time, users can inexpensively launch computationally intensive jobs that exploit large-scale parallelism. We show how Sprocket takes advantage of this opportunity with an experiment that processes jobs in constant time for inputs that do not exceed the parallelism limits of the underlying infrastructure. Figure 2.7 shows the completion times of Sprocket executing the grayscale pipeline as a function of video length for four videos. In this experiment, we scale the number of workers with the length of input video. Each worker

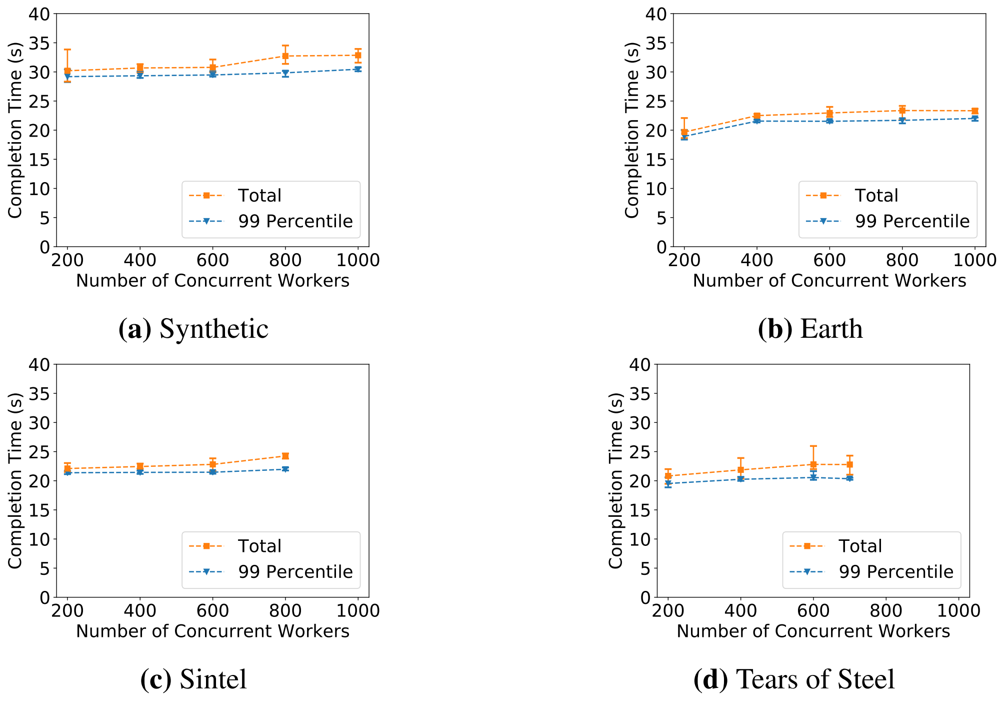


Figure 2.7. Matching Lambda parallelism to video length.

processes one second of video, and the *x*-axis shows the length of input video and hence number of workers used (e.g., 600 workers process the first 600 seconds of video). We show curves up to 1,000 workers (the limit of parallelism that we reliably and consistently extract from AWS), unless the video length is less than 1000 seconds. We measure both the job completion time (total) and the time 99% tasks finish (99th percentile). Each point corresponds to the average of five runs, and the error bars show the minimum and maximum values.

For the SYNTHETIC video, in which every chunk has the same content, Sprocket achieves constant execution time until 800 workers, where times increase slightly due to slight resource contention on S3 accesses. For the EARTH video, the time increase from 200 to 400 workers is caused by data size increases in the video content, since the first few minutes of the EARTH video are highly compressible. In SINTEL, the increased processing time near the end of the video is caused by more data variance near the final stage of the movie. TEARS OF STEEL has more data variance among chunks than other videos throughout the movie, so the distance between total time and 99th percentile is larger.

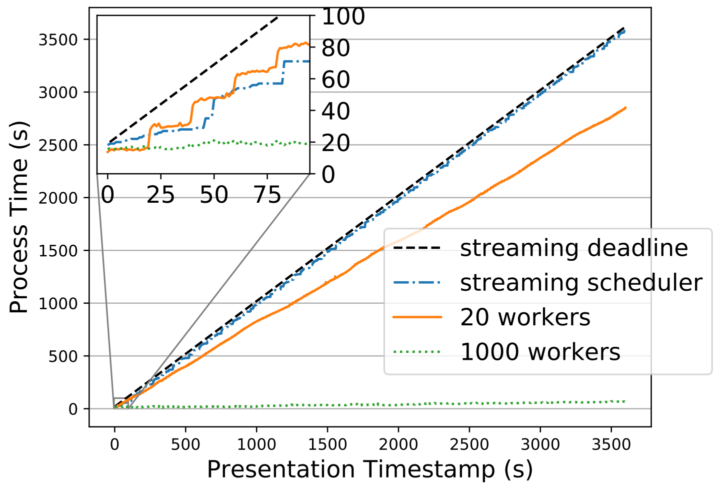


Figure 2.8. Streaming the EARTH video through the grayscale pipeline.

2.3.3 Streaming

In previous experiments we used Sprocket as a batch system focusing on completion time for the entire video. But we designed Sprocket to operate equally well as a streaming system. In this mode, once Sprocket finishes processing the first result frame, users can start to stream the video. Recall from Section 5.3 that workers encode video chunks into standalone DASH segments. As a result, even if a video is hours in length it does not need to be serialized into a single final video file before viewing. Sprocket is also able to handle streaming input, thus being able to process input video that is most recent. Hence, streaming is seamless as long as Sprocket processes and delivers subsequent video chunks in time.

To demonstrate this behavior, we execute the grayscale pipeline on the first hour of the EARTH video. Figure 2.8 shows the completion time of each worker and its chunk of video for the variable amount of workers determined by the streaming scheduler (Section 2.2.5) for 1,000 workers and for 20 workers. The dashed line corresponds to the deadline that Sprocket needs to meet for seamless streaming. The line starts at the completion time of the first chunk since users have to wait for Sprocket to process it. But once started, Sprocket can easily meet the deadline for the remainder of the video.

The inset graph zooms in to the first 100 seconds of video, clearly showing stairstep

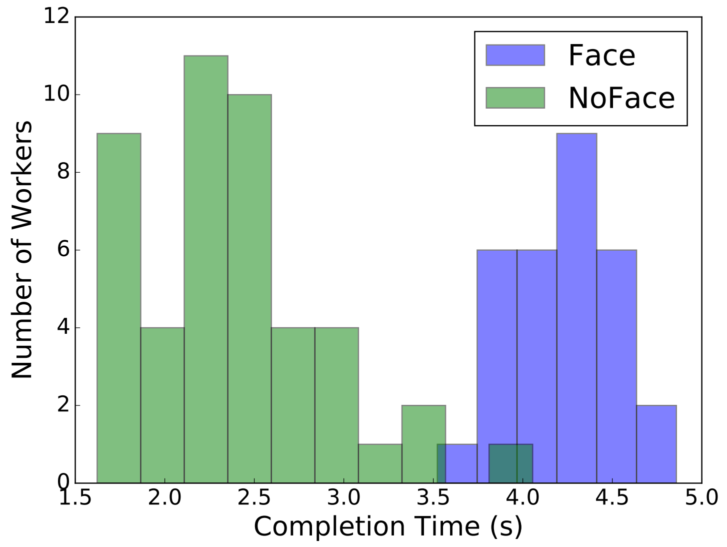


Figure 2.9. FacialRecognition stage behavior with and without faces.

behavior for both the 20-worker and streaming scheduler configuration. Each step corresponds to one wave of *n* workers executing in parallel, and zooming into the graph shows the steady and stable performance over time as waves of workers process the video.

2.3.4 A complex pipeline

Sprocket's behavior is highly dependent on the properties of the input video. Straightforward filters or transformations of video chunks perform the same work on each frame. The behavior of a more complex pipeline that recognizes and draws a box around a given actor's face, however, greatly depends on whether the input video contains a face, and whether that face is the one being queried. To demonstrate, we run the FacialRecognition pipeline on the NATURE and INTERVIEW videos (we use these short videos due to rate limits of Amazon's Rekognition API, as discussed in Section 2.2.3).

Figure 2.9 illustrates the bimodal execution times of the pipeline's Lambda workers in the FacialRecognition stage, depending upon the presence of a face in a given frame. Lambda workers processing the NATURE video, which has no faces, take between 1.5 and 4 seconds to complete the FacialRecognition stage. This execution time consists of the time it takes to invoke Amazon Rekognition to detect any faces present. Since no input frames contain faces, the stage passes along the frames immediately. For the INTERVIEW video, all the frames have faces, so the Lambda workers will experience increased execution times. When Rekognition does detect a face in a frame, it makes a second API call to compare the detected face with the target face. This second call adds another 1.5 to 3.5 seconds of execution time, resulting in the bimodal execution times. For this experiment, the total execution time for the 60-second NATURE video with no faces was 32 seconds, and the execution time for the 45-second INTERVIEW video with faces was 45 seconds.

2.3.5 Straggler mitigation

We evaluate the straggler mitigation strategy in the decode stage of the FacialRecognition pipeline. In the FacialRecognition pipeline, there is a serialization point between two stages, causing the delay of a single chunk to delay all later chunks.

We measure the pipeline performance without straggler mitigation, with a simple cloning strategy, and with Sprocket's straggler mitigation strategy (Section 2.2.6) for comparison. The cloning strategy duplicates each task, and when a task finishes the decoding process, it sends a message to terminate the other task. Since it is difficult to create reproducable straggler situations when using AWS organically, we instead emulate stragglers by forcing a particular task to sleep for 30 seconds before decoding. As shown in Figure 2.10, if there is no straggler mitigation, a straggler in this pipeline can delay the delivery of many chunks. When using cloning or Sprocket's straggler mitigation, though, the delay is nearly reduced to the expected time to process a chunk.

Both task cloning and Sprocket's straggler mitigation removed the effects of the straggler. But they have different costs: task cloning performs much more redundant work than Sprocket's approach. Table 2.2 presents the total Lambda time in the decode stage under the different strategies, averaged across five runs.

As explained in Section 2.2.6, after a worker finishes processing its own part of the GOP, it continues to work on the other worker's part of the GOP if the pair's work is not finished. In the

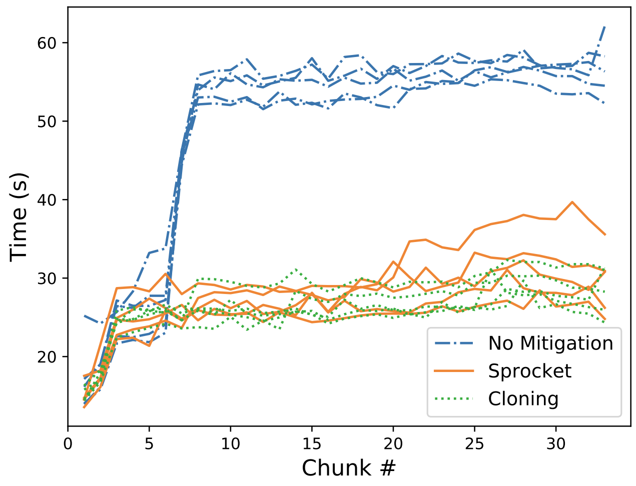


Figure 2.10. Comparison of chunk delivery time with various straggler mitigation strategies.

Table 2.2. Total Lambda running time and standard deviation in the Decode stage under different straggler mitigation strategies. The higher Lambda run times for cloning results in higher costs compared with Sprocket's strategy.

	No Straggler (sec)	Straggler (sec)
No Mitigation	181.05±6.17	218.58±15.44
Cloning	259.54±8.23	252.17±3.58
Sprocket	184.25±5.75	191.77±6.02

case that there is no straggler, this strategy only creates little extra Lambda running time, because the finish time for two workers in the same GOP is often very close. Once speculative execution finishes, the worker sends a notification to the straggler worker to terminate it immediately. In terms of cost, the slight difference in extra Lambda running time causes a negligible difference in extra costs. Although the cloning strategy duplicates each task, it does not double the total Lambda running time when there is no straggler — it costs only 41% more Lambda running time compared to no mitigation or Sprocket's approach.

The reason is twofold. First, once the faster task finishes the decoding task, it sends a stop message to the slower duplicated task, and the slower task can immediately stop what it has been doing and skip writing to S3. As a result, writing output data to S3 is never duplicated.

Second, there is an inherit advantage of cloning — the faster decode time of the two tasks decides the average decode time. The cloning strategy also has a faster average task time, but it requires more concurrent Lambda workers, and thus more total Lambda running time. Sprocket's straggler mitigation is almost as effective as cloning the tasks, yet cloning costs 41% more Lambda running time than Sprocket when there is no straggler, and around 31% when there is one.

2.3.6 Alternatives

The rise in data processing requirements has led to the development of a number of parallel data processing frameworks. Tools such as MapReduce [24], Hadoop [37], and Spark [82] have become essential components for many organizations processing large volumes of semi-structured, primarily textual data. A number of distributed-computing frameworks, such as Dryad [42], Apache Tez [85], Apache Kafka [47], and Hyracks [12] further implement pipeline-oriented computation and support arbitrary DAG-structured computation, with data passed along edges in a computation graph. These environments are particularly beneficial when they can amortize their resource footprint over a large number of user requests, enabling increased resource efficiency and job throughput via optimized resource allocation, assignment, and scheduling decisions (e.g., [39, 92]).

Sprocket, however, focuses instead on individual users who want to submit a single job to the cloud, and thus cannot amortize the resource footprint of any acquired resources over anything else. We would like to enable these users to program their video processing application independently, without requiring major providers to implement their desired functionality. As a result, we focus on serverless cloud environments which are well-suited to this deployment scenario. The ability to have near-instant, extremely bursty parallelism on demand is a compelling alternative to requiring a traditional, dedicated cluster-based approach. The on-demand nature of cloud serverless infrastructure allows any user to run jobs consisting of thousands of parallel executions for short periods of time at low cost, even for a single job.

Cloud providers do provide elastic offerings of more established parallel data processing systems such as Hadoop and Spark, but they are not a good match for Sprocket's goals. In terms of responsiveness, allocating and provisioning clusters potentially takes minutes before the new cluster can begin accepting jobs. Processing video with even simple transformations often does not involve any reduction in the data (input and output sizes are similar); efficiencies afforded by reductions in MapReduce-style computations do not apply to a wide range of video processing tasks.

To make this argument more concrete, we perform a simple experiment to illustrate the performance of a simple video processing application on Amazon's EMR Spark, an EC2 instance, and Sprocket. We use the EARTH video as input, segmented into two-second video chunks, and performed a simple grayscale operation using the `FFmpeg` tool in all frameworks. The Spark implementation used an 18-node cluster, with each node processing a partitioned set of video chunks into resulting mp4 output files. The EC2 implementation executed a batch script running 64 `FFmpeg` processes in parallel on an m4.16xlarge instance, which has 64 virtual cores and 256 GiB of memory. Sprocket used a filter pipeline (Section 2.2.2) executing a variable number of Amazon Lambdas, one per chunk, using up to 1,000 instances. Intermediate data was stored locally and the final output written to S3.

Figure 2.11 shows the execution time of the application on each platform as a function of video length. For Spark we show two lines, one including the time to provision and bootstrap the resources on AWS, and the other with just the application time after the cluster is ready. The Sprocket curve represents using either "cold" or "warm" Lambdas; the difference between the two is so small at these time scales that having separate curves would just overlap each other. Our goal is not to present this experiment as a "bakeoff" among the most optimized versions possible, but to illustrate the benefits of using serverless infrastructure for a single job. In particular, the startup time of provisioning cluster resources is significant in existing commercial offerings, which Sprocket avoids using the on-demand nature of Lambdas. (There are monetary advantages as well: the computation costs for a 30-minute video using the Local Script, Spark Warm, and

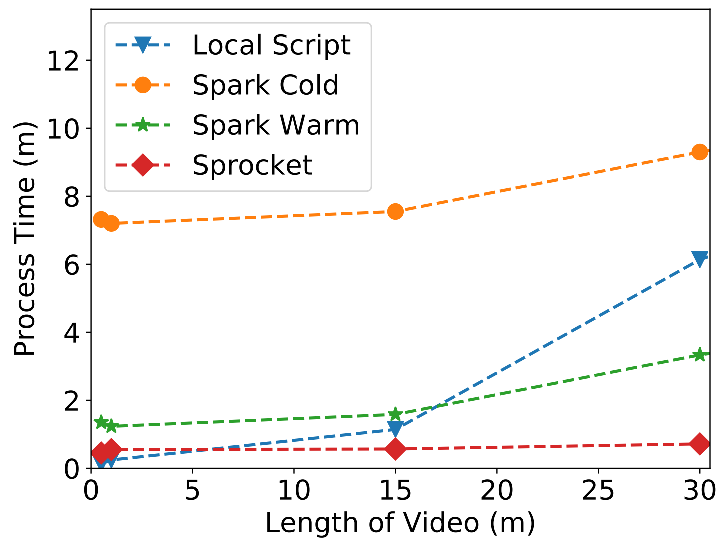


Figure 2.11. Comparing different video pipeline implementations.

Sprocket were $2.38, $1.42, $0.63, respectively.)

2.4 Conclusion

Sprocket is a parallel data processing framework for video content that uses serverless cloud infrastructure to achieve low latency and high parallelism. Sprocket applications can range from traditional video processing tasks, such as filters and other transformations, to more advanced operations such as facial recognition. Applications consist of a familiar DAG in which vertices execute modular user-defined functions on video frames, and frame data flows along the edges connecting the outputs and inputs of vertices in the graph. Sprocket targets serverless cloud infrastructure such as Amazon Lambda as the execution environment. As a result, Sprocket does not require dedicated infrastructure, can take advantage of the massive parallelism supported by underlying container-based virtualization, and can launch applications on-demand with minimal startup delay. Sprocket is an instance of GPSL model, which uses large amounts of serverless instances for parallel processing. The feasibility of GPSL model is proved by Sprocket as well as related systems like ExCamera, gg, and PyWren. The limitations of current serverless offerings including lack of direct networking support and cold start latency lead to the systems that will be described in later chapters.

2.5 Acknowledgement

Chapter 2, in full, is a reprint of the material as it appears in the Proceedings of the 9th ACM Symposium on Cloud Computing. “Sprocket: A Serverless Video Processing Framework.” Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Particle: Ephemeral Endpoints for Serverless Networking

In the previous chapter, I described Sprocket, a application framework for video processing pipelines on serverless. As a system in the GPSL model, Sprocket not only demonstrates the feasibility of the GPSL model, but also sheds light on limitations of existing serverless computing. Solving these problems can further improve the efficiency, practicability, and ease of implementation of GPSL applications. In this chapter, I focus on solving one of the problems: the lack of direct networking support in existing serverless infrastructure. In particular, I study the networking layer underpinning serverless platforms, which is particularly inefficient for applications that use hundreds of concurrent serverless functions to complete complex tasks, a model we called burst parallel. In lieu of native peer-to-peer networking capabilities on serverless platforms, communication among serverless instances must coordinate through external intermediate storage [7, 30, 31, 50]. This workaround has been widely used thus far, but is ad-hoc, application-specific, requires additional infrastructure services, and complicates user code. These drawbacks are an impediment to efficiently supporting a wider range of GPSL applications, which requires inter-instance communications. Examples of such communication are the "shuffle" phase in MapReduce-like applications [24, 74], message passing in typical scientific computing applications [35], and vertex traversals in communication graphs of dataflow systems [42, 66, 82].

While none of the major cloud providers today have burst-parallel optimized networking capabilities, enabling this kind of peer-to-peer networking has been of increasing interest in industry [87] and academia [45]. Two workarounds have been proposed, namely NAT hole punching and overlay networking, yet both have versatility and performance drawbacks.

In this chapter I characterize existing network approaches for serverless and propose an optimized network stack, Particle, to reduce network overhead when setting up networks for burst-parallel serverless jobs. Particle's key insight is that the network underpinning a burst-parallel job need not provide isolation between a user's containers, only between containers of different users. This trade-off is similar to why threads are more efficient to create as compared to processes, due to the difference in inter-thread isolation guarantees. I show that Particle can support a number of different GPSL applications, making GPSL more practical and efficient.

3.1 Background and Motivation

Much effort on serverless systems has focused on container startup time separate from the role of the network for interfunction communication. As serverless evolves from independent single functions to coordinated burst-parallel applications, fast, versatile, and scalable network creation becomes increasingly critical to satisfy the bursty nature of this application class.

VXLAN-based overlay networks such as Weave, Linux Overlay, and Docker Swarm were designed to accommodate the versatility and scalability requirements of modern datacenter networks, but their implementation is tailored to support tens of strictly isolated long-running connections rather than thousands of short-running ones.

We describe the underlying mechanism for how overlay networks are architected today and benchmark each piece of overlay network creation at both the application level and kernel level. Our primary finding is that the overlay data plane interacts with containers in a way that introduces severe latency issues for many VXLAN-based overlays — an issue that is exacerbated when interconnecting hundreds of serverless functions. Fortunately, such a bottleneck provides the opportunity for addressing the problem in a portable and general manner.

Containerization is the most common isolation mechanism in serverless platforms and is used by Google Function, IBM OpenWhisk, and Azure Functions. Therefore we focus the rest of the chapter on containers as an execution platform.

3.1.1 Overlay Data and Control Planes

The underlying technology that enables overlays is the VXLAN protocol. VXLAN is an encapsulation protocol that wraps packets from a container group with unique identifiers (VNIs) that allow communication without compromising isolation. Devices connected in this way then form an overlay network. An overlay network consists of two distinct parts, the control plane and the data plane. The control plane is an in-network service that exists to manage overlay networks across multiple tenants. These connections are initiated by the data plane within each host. The data plane, unlike the control plane, exists only as long as the serverless application. It is responsible for forwarding data to the correct containers based on VNI, IP, and MAC address.

The VXLAN data plane requires each host to have a VXLAN Tunnel Endpoint (VTEP) that is responsible for VXLAN termination and encapsulation. When a packet is sent from one container to another using VXLAN, the VTEP on the host encapsulates the original Ethernet frame from the container with a VXLAN header. The encapsulated packet in turn is sent out of the host with an outer IP and MAC header. When another container receives the packet, the VTEP on the receiver side looks at the VNI and inner MAC addresses and delivers the payload to the appropriate container.

Overlay networks also require a control plane to manage VTEP routing information. The control plane keeps a mapping of host VTEPs, VNIs, and container MAC addresses. When a container on one host sends data to a container on a different host, the VTEP encapsulates the packet with the VNI and checks locally if the routing information exists. If it does not, the control plane is probed and then the packet is routed with the new route. Control plane implementations are diverse, with some using virtual routers [91], gossip protocols [25], BGP [14], and KVstores [21, 25].

The glue that holds both of these network planes together is the network namespace. The Linux network namespace mechanism creates new logical network stacks in the kernel that each have their own network devices, neighbor and routing tables, `/proc/net` directories, and other network stack state. A network namespace is created by calling `unshare` or `clone` with a `CLONE_NEWNET` flag depending on the implementation. In the context of the overlay, the Virtual Ethernet devices (VETH) are used as endpoints that then connect namespaces together and can be configured to have MAC and IP addresses.

Overheads with respect to the data plane are a function of setting up namespaces, while overheads in the control plane are a function of setting up routes. To understand which of these place a larger burden on the application we perform a scalability analysis for both.

3.1.2 Performance Bottlenecks

We aim to understand overlay performance bottlenecks without being tied to any specific overlay approach. To this end, we avoid using proprietary software for these microbenchmarks when possible and build our overlay network using native Linux commands to manage the data and control planes. To ground our understanding of system overheads, we carried out a small microbenchmark. This experiment ran on Amazon AWS, using c5.4xlarge machines with Ubuntu 18.04 on Linux kernel 5.0.0-1004. We created a fully functional overlay network using the BGP-based Quagga EVPN [8] as the virtual router and the Linux native `iproute2` v5.2.0 [41] to manage Docker containers and namespaces.

Scaling Node Counts:

We first determine whether adding more nodes to an existing overlay network slows down the control plane. To answer this question, we first launched a 100-container cluster and created an overlay network interconnecting the containers on each node. We varied the number of nodes and recorded when new routes were added. Table 3.1 summarizes our observations,

Table 3.1. Scaling Up Nodes: End-to-end startup time remains relatively constant when increasing numbers of nodes while keeping the number of connections per node constant.

Total Connections / Nodes	Connection Time (s)
101 / 1	15.74
404 / 4	15.66
1616 / 16	15.99

Table 3.2. Scaling Up Connections Per Node: In contrast to Table 3.1, increasing the number of connections/namespaces on a single node scales poorly.

Total Connections	Namespace Setup Time (s)
100	10.02
400	38.90
1600	119.79

showing the time required to start a Docker container, initiate the data plane, connect to the control plane, and send data to a given receiver node. To increase the load on the BGP-based control plane, we scale up the number of nodes (and thus endpoints that connect to the control plane). The number of containers per host remains the same but the number that needs to be connected increases linearly until 1600 containers are networked. The extra connections are ones between VTEPs.

Takeaway: On 16 nodes the performance impact from the control plane is negligible and within the margin of error at this observed scale. We will show that this outcome is not the case for the data plane.

Increasing Connections On A Single Node:

Next we characterize the impact on data plane performance and scalability of adding containers to an overlay network. We create a single overlay network on a node and connect it to the control plane, varying the number of network namespaces added to this overlay. We focus our analysis on the overhead of the network namespaces themselves, rather than on container creation time.

Table 3.2 shows that the overall time increases linearly with the number of namespaces attached to the overlay (unlike what we observed with the control plane). Note that we show only the time to instantiate the network data plane. We measure scalability by varying the number of namespaces attached to this overlay network. We observe that the majority of the time is spent in the kernel.

Takeaway: Table 3.1 shows that end-to-end startup takes 15.74 seconds with 100 containers on one node. Table 3.2 shows that most of this time goes to networking the namespace together, more than 60% of the end-to-end startup time. As the number of network namespaces increases, so does the setup penalty. This lack of scalability is a major bottleneck for burst-parallel deployments on serverless.

3.1.3 The Role of Network Namespaces

Figure 3.1 illustrates the steps involved in adding a new network namespace to an overlay network. This process is similar for all overlay network software using a VXLAN-based overlay.

Initially the host instantiates a control plane namespace for the VTEP and the host's standard network namespace. First, we create a new guest network namespace. Next, we create a pair of VETH devices in the host namespace. From the host namespace, we place these VETH devices into the network namespace for the control plane and guest namespace. We then tether the local VETH with the VTEP's VXLAN and bridge interface, and turn up the local and guest interfaces. Finally, we establish a connection to the VTEP by setting an IP and MAC for the guest network namespace. At this point the guest is connected to the overlay and all data will transfer through the appropriate VNI. These steps are repeated for each new guest in the overlay network.

We further breakdown data plane creation by instrumenting the steps from Figure 3.1 with eBPF [61], and report relative and absolute times for connecting 100 network namespaces to the overlay. Table 3.3 breaks down execution time across the steps shown in Figure 3.1. Most of the time is spent in two steps, S3 and S4, and a negligible amount in others. While most of

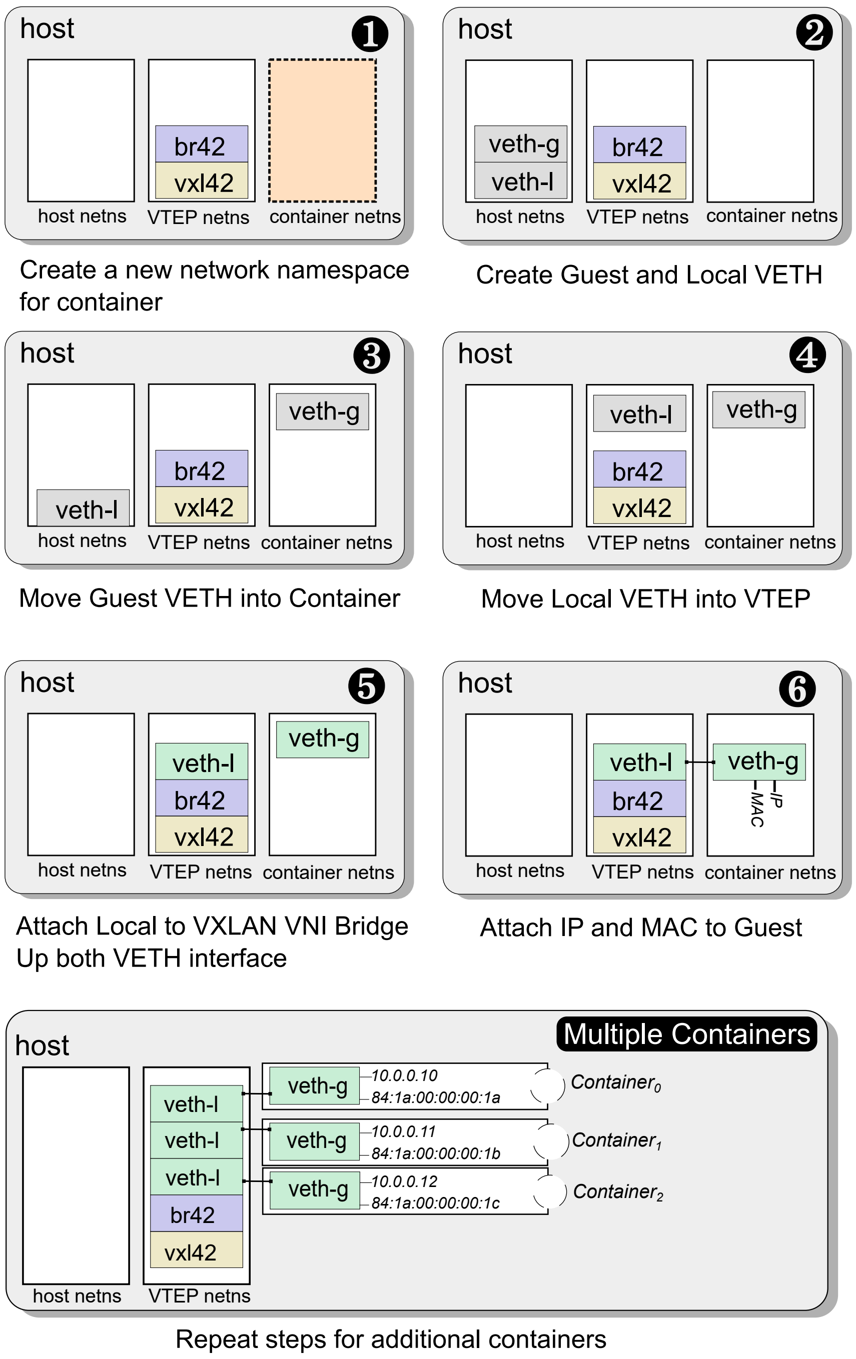


Figure 3.1. Creating a new network interface for the overlay dataplane involves a sequence of operations that are repeated for each new container. We refer to the network namespace as "netns". Initially only the VTEP, which communicates with the control plane, exists. BR is the bridge interface and VXL is the VXLAN interface attached to the bridge.

Table 3.3. Breakdown of the steps in Figure 3.1 for the overlay data plane for 100 network namespaces. Most time is spent moving VETH devices between namespaces (steps S3 and S4).

Step	Time (s)	Percent of Total
S1	0.10	0.92%
S2	0.10	0.92%
S3	5.18	47.71%
S4	4.77	43.95%
S5	0.49	4.45%
S6	0.22	2.03%

the other steps from Figure 3.1 either configure a VETH device or create a new one, S3 and S4 are the only ones that perform a *namespace crossing* and move a network interface, the VETH. The local VETH device is moved from the host network namespace into the control plane network namespace and the guest VETH is moved from the host network namespace into the guest network namespace.

Moving VETH devices is inherently expensive because the `dev_change_net_namespace` kernel routine performs a long-running task to ensure that the VETH device is safely moved while holding the `rtnetlink` semaphore. When a move is initiated, the kernel first informs all devices on the notifier chain that the VETH is being unregistered. Next, it removes the VETH device handle from the host namespace and flushes old configurations. Finally, it updates the VETH data structure to point to the new namespace, and informs the namespace and notifier chain that the device is live.

In terms of scalability, when more guests are added to the overlay each of these six steps are repeated, resulting in three different `unshare` calls and two namespace moves per container. This overhead accounts for the linear increase in time in Table 3.2. A design for a burst-parallel overlay network needs to address both the scalability and performance challenges.

3.1.4 Challenges of Existing Approaches

Table 3.4 compares the capabilities and challenges for different serverless networking options. Any serverless networking approach must be suitable for a bursty low latency multi-

Table 3.4. Comparison of capabilities and challenges for different serverless networking options: A serverless network solution must be suitable for a bursty low latency multi-tenant environment. Serverless systems must also be able to work with Layer-3 connectivity and provide a per function IP for direct interfunction communication [90]. Today's overlay networks have the appropriate control plane mechanisms for serverless environments but have high network startup latency.

Serverless Networks	Isolated	Low Latency	IP Addressable	L3 Solution	Connection Type
NAT Hole Punching	✓	✗	✗	✓✗	Point-to-point
Kubernetes Pods	✓	-	✗	✗	Port multiplexing and volumes
Docker Host Networking	✗	✓	✓	✓	Direct IP
Overlay Network	✓	✗	✓	✓	Direct IP
Particle	✓	✓	✓	✓	Direct IP

tenant environment and make minimum assumptions about the network and application layer. Additionally, based on previous work [71, 87, 90], serverless systems must also be able to work with Layer-3 connectivity, as nodes hosting lambda functions are not always Layer-2 adjacent.

Overlay networks are attractive because they fulfill most of the requirements, but current implementations have significant performance overheads. Container orchestrators such as Kubernetes use pods to consolidate containers under a single namespace with one routable IP per pod because of the "one-container-per-pod" commonly-used design pattern [69]. This approach potentially has higher startup latency as each pod starts a container, a network namespace, and a pause container. If we use hundreds of containers per pod to avoid this overhead, each container will need to communicate through application-managed port multiplexing or by creating a volume in the pod for containers to share. From a developer standpoint, changing applications to have port multiplexing logic and manage per-pod databases with related application logic incurs significant engineering costs.

Other alternatives also have significant limitations. Using the Docker host network fundamentally is not a multi-tenant solution, and NAT hole punching requires creating multiple point-to-point connection pairs, none of which are IP addressable. Based on our evaluation of existing approaches, we have designed Particle to satisfy existing serverless requirements, and focus on the ability to quickly generate and interconnect thousands of ephemeral network endpoints.

3.2 Particle Design

We present Particle, a networking architecture that optimizes network startup in burst-parallel serverless environments. Particle provides an ephemeral dynamically generated pool of IPs at an almost constant startup time. Rather than using memory-intensive caching techniques, Particle creates groups of network endpoints by first separating network creation from other user namespaces, and then optimizes the creation of network endpoints by eliminating serialisation points, batching calls, and consolidating VETH devices while maintaining per-function IPs. In this way, Particle can accelerate network namespace creation without any adverse effect on capability or generality for applications. Particle addresses the challenges from §3.1.4 through three design principles:

Match Infrastructure to Application:

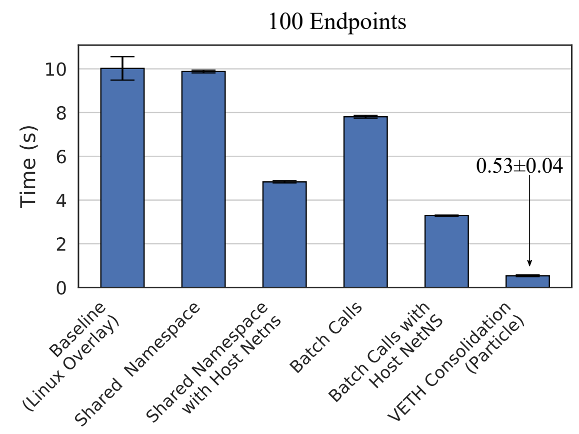
Burst-parallel applications invoke hundreds to thousands of serverless instances to complete a single complex job. Today's underlying infrastructure is not optimized for the bursty nature of this application class since each serverless task is treated as a stateless independent function. Particle employs techniques to consolidate network infrastructure without compromising generality, programmability, or network versatility.

Generic Socket Interface:

Containers that have their IPs allocated by Particle must be able to communicate with each other without the need for any specialized IPC protocol, system, or storage. Accessing third party or network-hosted services must also be possible. Containers must be able to use POSIX socket calls to communicate with each other (§3.2.1).

Portability:

Particle makes minimum assumptions about the system where it is deployed. Porting Particle to additional overlay systems is straightforward as most overlays rely on the default



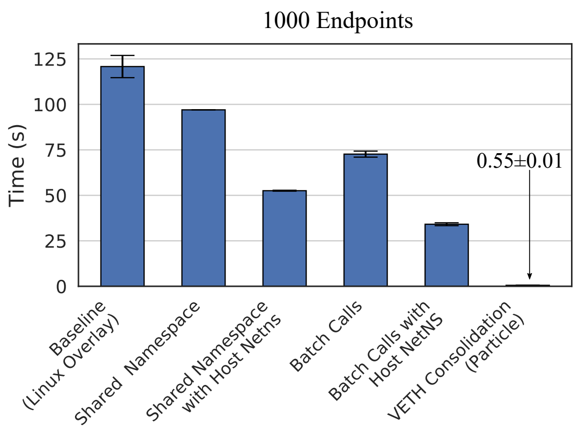


Figure 3.2. Time to connect 100 and 1000 concurrent network namespaces to an overlay. While the baseline and other optimizations increase linearly with more endpoints, VETH consolidation allows Particle's startup to remain close to constant when scaled up.

Docker runtime for network provisioning. When integrated, Particle has no adverse effect on throughput.

3.2.1 Design Space Exploration

In this section we explore three different approaches for optimizing network startup: (1) namespace consolidation, (2) batching, and (3) virtual interface consolidation. We seek to understand the trade-offs in each optimization to inform our final Particle design. In Figure 3.2 we use microbenchmarks that focus on network creation time to compare the designs, and use the Linux Overlay data plane as the baseline (system configuration details in §3.4).

Design 1: Namespace Consolidation

Based on our findings in §3.1.3, the network namespace itself is a contributor to high startup latency. One way to address this problem is to adopt what many container orchestrators such as Kubernetes [69] and Amazon Elastic Containers [3] do when co-locating related services under one network namespace and IP. These services perform namespace consolidation to simplify the management plane, but we can extend the traditional 'one-container-per-pod' model to 'many-containers-per-pod' as a performance optimization. This optimization is a natural fit for a burst-parallel environment where many serverless instances work together as part of a single task.

We explore this design by creating a new *root network namespace* for groups of containers, but each container maintains separate kernel namespaces (mnt, pid, ipc, user, cgroup) for other types of isolation. In this way namespaces can also be created for each tenant, while individual containers operate without needing to change assumptions about the environment. Each container is attached to the Particle root namespace and inherits all of its iptables and routing configuration without creating a network namespace itself. Since we want each container to have an addressable and routable IP address (§3.1.4), we create a VETH interface for each container inside the namespace with an IP and MAC address.

Microbenchmark results in Figure 3.2 show that this "shared namespace" design has a modest performance benefit when starting 1000 endpoints, but almost no performance impact for 100 endpoints: shared namespaces alone do not address the root issue shown in Table 3.3. When a new container is created the overlay controller must create a VETH pair for each new container and perform VETH moving. With a shared namespace, the only difference is that, rather than moving the VETH into a separate network namespace per container, the VETH moves into the shared network namespace.

This optimization can be taken a step further if the host namespace can be used rather than an additional shared namespace. Doing so reduces the number of namespaces and it reduces one VETH move: the host namespace creates a VETH pair and only moves the local end into the VTEP. The trade-off is that this optimization does not match a multi-tenant setting as there is no isolation of the host interfaces.

Design 2: Batching and IP Pooling

A major disadvantage with shared namespace consolidation is that, although it takes advantage of the fact that burst-parallel tasks can be consolidated within a single root network namespace, setting up the network namespace itself was still performed iteratively. The advantage to performing namespace consolidation is that it reduces the complexity of managing many namespaces in a burst-parallel environment. In the next design, we push the ideas further by performing a batching optimization to VETH creation inside the network namespace.

With batching, when a burst-parallel request is received, the system creates an IP pool based on a specified IP range and number of containers. Rather than wait until the container is created, all the necessary virtual interfaces for the data plane are created immediately and attached to the root network namespace. Once complete, the system then enters the control plane namespace of the overlay and sets up the corresponding VETH devices and attaches them in batch to the VXLAN port. One key benefit to batching is that it reduces context switching and the number of `unshare` calls. However, implementing batching alone still results in $O(N)$ namespace crossings and VETH movings.

For the same benchmark, Figure 3.2 shows that batching provides a 22% improvement over a standard Linux Overlay with 100 containers, and improves to 42% with 1000 containers. Although batching improves performance over Linux Overlay and a simple shared namespace, the system still performs many $O(N)$ operations within the namespace, albeit batched. For example, N different VETH devices, MAC addresses, and IP addresses are still being created.

Final Design: Virtual Interface Consolidation

The first two designs develop a management plane, the root namespace, and the insight that creating the network, VETH and IPs in batches for a burst-parallel group improves performance. Unfortunately, neither of these designs significantly reduces the total number VETH devices that the system must create. Table 3.3 shows that regardless of batching and namespace consolidation, each VETH device created incurs an overhead. Additionally, for each container created there are still $O(N)$ VETH devices created and $O(N)$ VETH interfaces moved across namespaces. The first two designs improve performance, but do not address this last issue. As a final design element, we focus on making VETH device creation a constant time operation rather than a linear one. To do so, we create a single VETH device inside the root namespace and attach multiple IPs to this root VETH interface.

In traditional overlay networks there is a one-to-one mapping between VETH device pairs and containers. This mapping is only necessary because each container resides in its own isolated environment. We dispense with per-container network isolation, attaching one VETH pair to the control and data planes. Multiple IPs and MACs are then attached to the root namespace's single root VETH interface. From the perspective of the control plane, all IP addresses attached to a VXLAN interface are routed.

Figure 3.2 shows that this new one-to-many mapping between VETH interfaces and IP addresses improves performance by an order of magnitude since only one namespace crossing is required for one burst-parallel job. VETH consolidation improves performance by a factor of 17$\times$ when creating 100 containers, and 213$\times$ with 1000 containers.

The absolute time to start 100 network namespaces is 534ms, and for 1000 network namespaces 553ms. The 534ms comes from two parts, creating a new root network namespace and attaching the overlay. The root namespace starts as a Docker container with only the loopback interface (`---net=none`) which takes on average 300ms. The remaining 234ms is the time to create the root VETH interface, attach it to the control plane network namespace, and add IP addresses.

At a high level Particle systematically replaces expensive *O(N)* “per-container” calls to be *O(1)* “per-job” calls. Based on our findings in §3.1.3 the creation of a container network involves several kernel locks that effectively serialize network creation. This overhead is exacerbated when trying to create hundreds of serverless instances and corresponding network endpoints to coordinate a single job.

Particle is designed to be integrated into existing overlay networks with minimum changes. As described in §3.1.2, overlay networks consist of a control plane and data plane. While the control plane varies among designs, all of them rely on a similar data plane implementation as shown in Figure 3.1.

Burst-parallel applications have the property that the logical compute unit is a batch of serverless instances working towards a single goal. As a result, while each container benefits from the standard isolation guarantees (process, file system, etc.), the network interface does not require strict isolation among instances. Particle does, however, still enforce strict network isolation from the host and other tenants.

Figure 3.3 illustrates Particle’s architecture. A single namespace and VETH device are created per tenant per node. Multiple secondary IPs are then attached to the VETH device, creating an ephemeral per-job IP pool. The overlay enables these IPs to be routable through its own policies and mechanisms. Containers can then attach to available IPs and transmit data between containers both intra- and inter-node. When the job completes, Particle removes its namespace and IPs.

Figure 3.4 shows an example of VETH consolidation in a multi-tenant setting. Each guest has its own Particle namespace with a MAC that is shared with the VTEP. A single VETH interface can host thousands of secondary IP addresses for any container sharing the Particle (root) namespace. Applications have several different options for how to interface with this system.

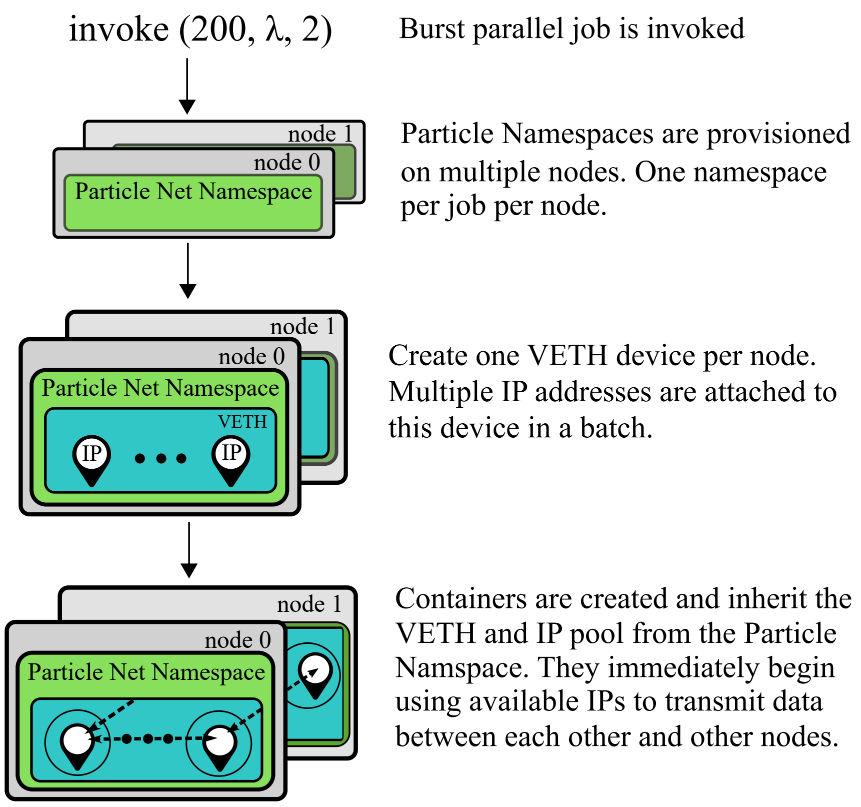

invoke (200, λ, 2)
Burst parallel job is invoked
node 1
node 0
Particle Net Namespace
Particle Namespaces are provisioned on multiple nodes. One namespace per job per node.
VETH
IP
Create one VETH device per node. Multiple IP addresses are attached to this device in a batch.
Containers are created and inherit the VETH and IP pool from the Particle Namspace. They immediately begin using available IPs to transmit data between each other and other nodes.

Figure 3.3. A Particle namespace with containers attached. Only the network namespace is shared among containers of a single application. Each Particle namespace has its own MAC address which is given to the VTEP for routing. Multiple tenants have different Particle namespaces.

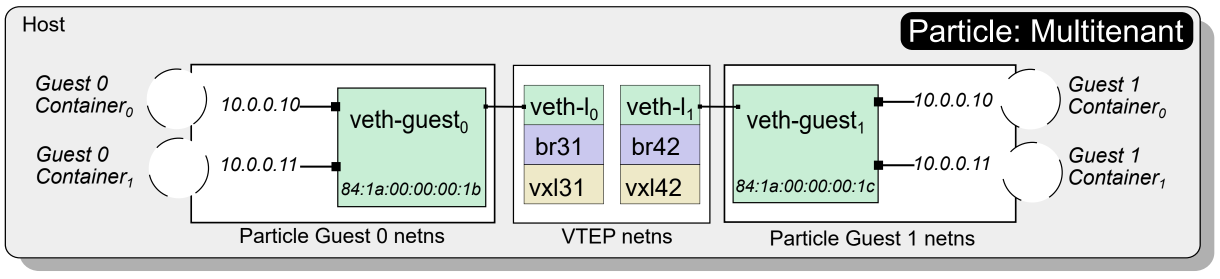


Figure 3.4. Multitenancy with Particle: Each application has its own Particle namespace to maintain network isolation from other tenants and the host. The control plane is responsible for provisioning extra VNIs in the form of additional VXLAN interfaces for additional tenants. Designing an overlay this way also allows each guest to use any IP address they want without restriction.

3.2.2 Isolation and Application Interface

Consolidating VETH devices and namespaces of multiple containers into one virtual device in one namespace can have side effects for containers within an application. Separate namespaces isolate network resources and provide security isolation. If a container is compromised, other containers in different namespaces are unaffected. Because Particle consolidates network namespaces, it cannot provide the same granularity of security isolation. However, since Particle only consolidates namespaces of the same tenant/application, and different tenants are always isolated by separate namespaces, we consider this tradeoff acceptable for application patterns consisting of multiple serverless instances working together as part of the same application.

Conceptually, applications request different IP addresses for different containers, and Particle assigns those IPs to avoid conflicts. However, if the application in the containers do not respect the assignment, different containers for the same application can interfere with each other by trying to `bind` to the same IP address/port pair. One scenario where this can happen inadvertently is when an application runs multiple containers on the same host and shares a VETH device via Particle. If they try to bind to the same port with `INADDR_ANY`, a port conflict can occur. As a result, applications need to use the IP addresses assigned to them to avoid such conflicts. Rather than relying on the application to use the correct IP address, Particle can interpose by overriding libc's `bind` call (via an LD_PRELOAD mechanism) to ensure that the IP address arguments match the IP addresses assigned to the container. If the application does not use the dynamically-linked libc, or directly calls the `bind` syscall, Linux Seccomp [43] provides a mechanism to enforce the assignment of the IP addresses. Seccomp's filter mode allows specifying what arguments are acceptable for certain syscalls, in this case, assigned IPs as arguments for `bind`.

3.3 Implementation

Particle is implemented in C and is integrated into the `iproute2` tool included natively in Linux. Particle is not designed to be used directly, rather it is a core module that exists within an overlay system. As a result, Particle does not make any assumptions about what kind of control plane is being used.

Porting Particle to an existing overlay network requires an adapter on the control and data plane side. For the data plane, now overlay systems do not need to create a network namespace when provisioning single containers. Additionally, they must create one additional container that is passed into Particle for the group namespace. For overlay systems that use a key-value store, a control plane adapter is required to pass the IPs into the database as a one-time operation.

In our evaluation we integrate Particle's module into the Linux Overlay and Weave Overlay systems. In both cases, these overlays pass in the namespace of the VTEP and must create a new container with no initial network. A pointer to this network namespace is also passed into Particle. At this point Particle has a handle to both a control plane network namespace (VTEP) and data plane network namespace. Based on how many containers are requested, Particle initializes the shared namespace with the same number of IPs. Context is switched back to the existing overlay system which advertises the route to the other members of the control plane based on the Particle MAC address.

3.4 Evaluation

We evaluate Particle's startup performance for a real-world burst-parallel application, Sprocket (Chapter 2). In our experiments we use AWS EC2 `C5.4xlarge` instances, each with 24 vCPUs, 32 GB of memory, and 10 Gb/s network bandwidth. All instances are in the same virtual private cloud (VPC) and placement group for stable network performance. The instances run Ubuntu 18.04.2 LTS using a Linux 5.0.0 kernel with a default configuration. We use iproute2 version 5.2.0, Quagga router version 1.0.0, and Docker version 19.03.1-ce.

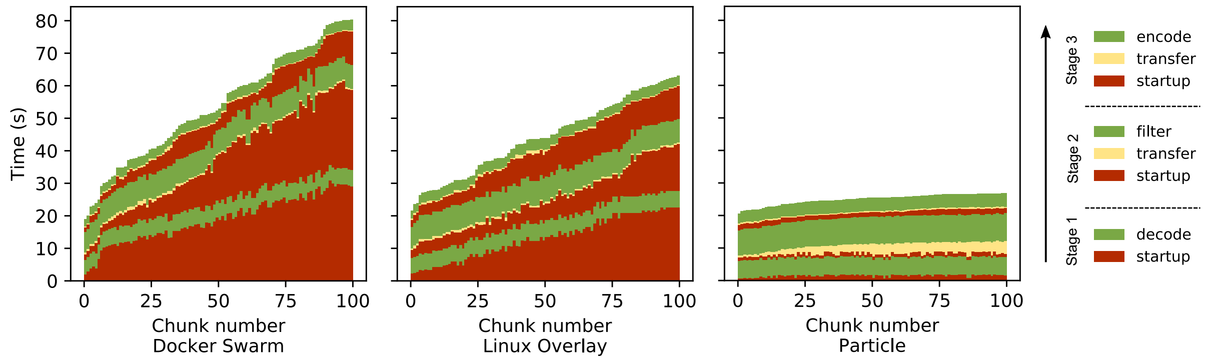


Figure 3.5. Timeline of Video Processing Pipeline: Comparing Docker Swarm, Linux Overlay, and Particle performance running a Sprocket video processing pipeline. Points on the x-axis represent the processing steps of a single video chunk (decode, filter, and encode), ordered by completion time. Particle eliminates the bottleneck in container startup, and runs 3× and 2.4× faster than Docker Swarm and Linux Overlay, respectively.

We let the application itself determine the number of threads and run the application on 96 core `C5.24xlarge` machines with 192 GB of memory and 25Gb/s bandwidth.

Sprocket is a serverless system that takes a video as input and first decodes it into frames. These frames are then subject to various transforms such as object detection, facial recognition, or grayscale. After the transforms complete, the frames are finally re-encoded. We ported Sprocket's runtime to run locally and changed its communication module to work with Docker Swarm Overlay, Linux Overlay, and Particle+Linux Overlay.

Each stage (decode, transform, encode) is processed on a different nodes using 100 containers in each stage. The input video consists of 100 one-second video chunks, which are given as input to a first wave of 100 containers started at the same time. Once each video chunk has finished a stage, it signals the downstream service to start a new container and pull data via the overlay network. The process is repeated until the video chunk has passed through three stages.

Figure 3.5 shows the per-chunk processing timeline of a Sprocket pipeline. Before each pipeline stage, the containers must start and connect to the overlay network so data can be sent to the downstream machines. With Docker Swarm and Linux Overlay, the startup time dominates the overall processing time. Even though each chunk is started simultaneously during the first stage, the network causes a serialization effect that prevents the system from being truly burst-parallel. On subsequent stages, the containers are started on-demand,i.e., as soon as a chunk has finished processing it starts the next stage without a barrier. This freedom causes subsequent stages to take relatively less time as there is reduced contention on the machine.

Particle eliminates the bottleneck in startup so that all containers are started within 2 seconds. As a result, the Sprocket pipeline using Particle is $3\times$ faster than using Docker Swarm and $2.4\times$ faster than Linux Overlay.

The increased data transfer time between the Particle pipeline's decode stage and filter stage is because the elimination of serialization in container startup increases the number of concurrent data transfers. This change shifts the bottleneck to the network, temporarily congesting the network and slowing down the transfer step; in other words, Particle accelerates network startup to the point where network throughput becomes the bottleneck. This effect does not manifest between the filter and encode stages because processing of the decode-filter stage effectively spreads out the data transfer. When using Docker Swarm and Linux Overlay, container startup is much slower, spreading out data transfers between stages and preventing the system from fully utilizing the network.

Table 3.5 summarizes the percentage of time spent on each stage over three runs. Particle spends most of the time doing data processing, while other overlay networks spend substantial time in the startup stage. Particle's higher proportion of time in data transfer is a result of both reduced overall execution time and the network saturation discussed above.

For a user paying for a serverless burst-parallel service, Particle enables the cost of a job to reflect meaningful work being done rather than infrastructure and setup time.

Table 3.5. Video Pipeline Breakdown: Percent of total run time spent in different operations in three networks. We take the average of three runs. Particle spends the most time in actual video data processing.

	Startup	Data Transfer	Data Processing
Docker Swarm	69.86%	1.89%	28.25%
Linux Overlay	62.12%	2.50%	35.38%
Particle	17.77%	10.92%	71.31%

3.5 Discussion and Limitations

Multi-Node Scalability. Particle's common use case is to enable serverless networking for burst-parallel functions (containers) that are distributed across multiple hosts. Microbenchmarking showed that the overlay control plane connecting multiple hosts was not a bottleneck. This finding led us to focus on optimizing bottlenecks on each node, and evaluating the effect in a multi-node setting through multiple experiments (§3.1.2).

As the number of namespaces on a single node increases, namespace setup time increases proportionally. Particle solves both of these problems by reducing namespace setup time regardless of the number of namespaces. For jobs spanning multiple hosts, Particle reduces setup time on each host on which the job runs. Particle enables serverless providers to increase the number of containers per machine without compromising application latency. If we need to interconnect 100 containers for a burst parallel job, the spectrum is 100 containers on 1 machine or 1 container on 100 machines. The choice represents a trade-off between monetary cost and performance. Particle closes the gap between these options and enables a trade-off that improves performance without sacrificing cost.

Application to General Serverless Workloads. Containers are often started on different hosts to reduce load and improve the availability of the serverless functions. Particle is an optimization using overlay networks to address this multi-node case. Figure 3.1 shows the six steps necessary to setup an overlay network. A management container is not necessary to set up a network between containers on a single host, a bridge will suffice. The advantage of Particle is that it enables users to write programs as if they are still using a bridge, but the containers are available across multiple nodes. Particle is primarily optimized for this multi-node use case.

This chapter shows that container overlays are one way that a serverless cloud provider can implement serverless networking. Unfortunately, overlay networks today are not optimized for this use case. With Particle, overlay networks can be created with a negligible amount of overhead on multiple nodes with thousands of serverless functions.

While Particle was motivated by burst-parallel applications, the lessons learned are not limited to it. The experiments show that the network namespace itself is a source of inefficiency in serverless, and a design like Particle can address this issue, achieving the greatest benefits if the VETH and/or namespace can be consolidated. If they cannot, the network namespace may be reused across multiple calls (also reducing cold start at the cost of higher memory usage).

Jobs and Network Namespace Sharing. In this chapter we define a job as a single invocation of a computation run by one user. As a Particle namespace is cheap to create, the isolation level can be modified without loss of performance. On one extreme, a Particle namespace can be created for each tenant. In this case jobs that a tenant runs would not be isolated (Figure 3.3). At the other, every job can have its own Particle namespace that exists just for the job. The design enables providers to choose what isolation model in this spectrum is most appropriate for their use case.

Particle chooses to only relax the isolation of the network namespace to ensure that, if a single function fails, it does not cause a domino effect that corrupts other parts of the system (e.g., the file system) which in turn could cause further function failure.

3.6 Related Work

Container Orchestrators. Kubernetes and Amazon Elastic Containers use shared namespaces to simplify service management between shared jobs in a pod or task group. However, employing this method alone does not appreciably change startup latency, as discussed in §3.1.4 and §3.2.1.

Communication Alternatives. Pocket [50] is an intermediate storage layer for burst parallelism that employs multiple storage media (e.g., a Redis key-value store, AWS S3, etc.) to accommodate different workloads in a cost-efficient way. Locus [73] focuses on shuffle performance in burst-parallel applications. It uses a performance model to select the appropriate storage medium in the cloud. SAND [2] proposes a message queue approach for inter-container communication. While these systems improve on existing communication mechanisms, they incur extra infrastructure costs and lack the generality of a direct communication mechanism. Shredder [97] takes a completely different approach by performing compute directly inside storage nodes.

Alternative Virtualization Layers. Particle focuses on optimizing network startup for container-based serverless systems since containers are a dominant virtualization platform. However, Kata [48] and Firecracker [1] have proposed an alternate serverless virtualization architecture using microVMs. These microVMs employ TUN-TAP devices to build an overlay network rather than network namespaces, and therefore represent an entirely different approach to networking. As a result, evaluating and optimizing network startup and configuration in these architectures is an interesting open question.

Container Network Setup. Mohan et al. [63] identify that network creation and initialization account for the majority of latency in bursty container creation. They extend the idea of Pause containers [57] to pre-create network namespaces that can later be attached to new containers. This technique is effective but it introduces security issues in a multi-tenant setting as new containers are reusing cached network namespaces. Additionally, the caching overhead is linear with the number of namespaces, i.e., memory usage increases with more containers attached to the network. Particle only needs to create a single network namespace for a group of containers, making it faster and more memory efficient than the caching technique.

3.7 Conclusion

As serverless evolves to accommodate GPSL applications such as burst-parallel, we need to reconsider the original notions about serverless design patterns. We take for granted that a long-running application will amortize costs for certain one-time operations, such as setting up infrastructure. In serverless burst-parallel, however, these one-time operations are repeated hundreds of times and the cost is paid on each invocation. In this chapter I focused on a key bottleneck for burst-parallel applications, network startup time. I found that provisioning the network can be a significant portion of execution time. I closely examined the overheads in establishing connectivity among containers in overlay networks and designed a system, Particle, to address these issues. Particle maintains serverless application requirements of generality, versatility, and multitenancy while providing short network startup time on both single and multi-node deployments. I show that in these scenarios Particle improves total application runtime by at least a factor of two over existing solutions. Particle enables fast serverless networking setup in multi-node deployments, which would benefit Sprocket and other GPSL applications that require intercommunication between serverless instances.

3.8 Acknowledgement

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the 11th ACM Symposium on Cloud Computing. “Particle: Ephemeral Endpoints for Serverless Networking.” Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. The dissertation author was the co-primary investigator and author of this paper.

Chapter 4

Sophon: Efficient Container-grained Serverless Scheduling

In Chapter 3, I described Particle, which tackles the challenge of providing direct networking among serverless instances. Another challenge I observed in serverless is resource scheduling in serverless platforms. Although serverless users do not need to worry about resource allocation and scheduling, such tasks are now offloaded to the cloud vendor. From my experience with Sprocket, cloud platforms cannot always efficiently satisfy the resource requirements of GPSL applications.

GPSL applications like Sprocket can request computing resources aggressively in a short amount of time due to the high parallel nature of GPSL applications. This behavior imposes challenges for the underlying platforms since they were originally designed for less resource intensive workloads like request handlers.

For each serverless request, the platform needs to choose the host node to handle the request. When the chosen host has a warm function environment ready, a warm start ensues, otherwise, a cold start occurs. Warm starts have huge performance advantages over cold starts for they skip all the initialization steps including starting VMs, containers, language runtime, etc. Keeping warm environments, however, has high resource costs, especially memory resources. Cloud providers often make tradeoffs between keeping warm environments to improve performance for future requests, and shutting down idle warm environments to reduce resource consumption.

The ratio of these warm- vs cold-starts is critical to both the user-perceived performance of the deployed functions as well as the efficiency of the platform from the service provider's perspective. In the former case, if functions are scheduled in such a way that they mostly run in warm containers, the overhead to issue and start execution can be greatly reduced.

For users, faster function start times results in higher overall job performance. For providers, maximizing the scheduling of functions to warm containers avoids the overhead of state transfer and reduces the setup time of the serverless function execution context, which is time that is typically not billed to the customer. In this chapter, my goal is to design serverless scheduling mechanisms and policies which maximize warm start opportunities, benefiting both users and providers.

Based on my measurements of OpenWhisk [71], an open-source serverless platform, together with workloads from the Azure Function Traces [78], I observe that its node-grained serverless scheduler can make inefficient scheduling decisions when handling real-world serverless workloads, especially for when workload is high or bursty, which is typical in GPSL-style applications. In OpenWhisk, the coarse-grained resource information used by the scheduler is not sufficient to track the underlying warm container resource states, which ultimately dictate overall system performance.

To address these scheduling inefficiencies I propose Sophon, a container-grained serverless scheduler. Unlike existing node-grained schedulers that do not track the states of containers, Sophon maintains centralized information tracking container lifecycles using piggy-backed invocation and container state transition messages. I explore several scheduling policies on Sophon that strikes a balance between spacial locality and avoiding container resource contention.

I implement Sophon in the OpenWhisk platform and evaluate the different policies on both Sophon and OpenWhisk. Experimental results show that scheduling policies utilizing Sophon significantly reduce contention on the worker nodes, and greatly reduces the number of cold starts and container thrashing. When servicing workloads from the production Azure traces, Sophon provides 78% higher maximum throughput and significantly lower invocation latency than that the node-grained scheduler of OpenWhisk.

4.1 Background

4.1.1 Serverless platforms

As serverless computing has increased in popularity, many serverless platforms have emerged. Examples include AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions, as well as locally-hosted open-source projects such as OpenWhisk [71], OpenFaas [70], Knative [51], and OpenLambda [38]. Most commercial serverless platforms do not reveal their underlying implementation details. As a result, limited information is available to the public, although academic studies have probed into these commercial platforms using various measurement techniques to better understand the underlying environment [90]. Because of its openness and use in a large commercial platform—IBM's cloud—in this chapter I focus our attention on the OpenWhisk platform.

For a given serverless workload, across multiple tenants and applications, the mixture of cold and warm starts has significant performance implications for users and overhead implications for providers. In concrete terms, cold starts can take several seconds to begin execution, whereas warm starts might begin in just milliseconds [31, 64]. According to published serverless trace analyses from Microsoft Azure [78], most functions run for a very short time: the median function duration is only 150 ms. Such short function invocations underscore the importance of increasing the opportunity to exploit warm starts, since the overhead of a single cold start could swamp potentially dozens of warm invocations. Execution contexts primed with the code and data necessary to begin execution quickly are due to a previous invocation remaining active, rather than being reclaimed by the serverless platform. A common heuristic is to cache used containers for a fixed amount of time before decommissioning them in case additional function invocations arrive [90].

4.1.2 Two scheduling architectures

I now describe the scheduling architectures of open-source serverless platforms using publicly-available information. Serverless cluster schedulers must establish one or more sandboxes on nodes in the cluster, and then assign individual function invocations to one of those sandboxes. The first decision a serverless scheduler must make is the location of those isolation sandboxes. The second is the assignment of individual invocation requests to a sandbox. I separate serverless schedulers into two categories based on whether these two scheduling decisions are made in the same process or separate processes. When both are decided by one scheduling process, I call it "coupled scheduling", otherwise I call it "decoupled scheduling". OpenWhisk employs coupled scheduling because the creation of a sandbox is directly decided by the scheduling of an invocation. OpenFaas and Knative exemplify decoupled scheduling. Both of them rely on an underlying sandbox orchestration service (e.g., Kubernetes) for sandbox auto-scaling and scheduling, while individual invocations are scheduled to existing sandboxes in a separate process.

Coupled schedulers have more control of the scheduling of each invocation. In decoupled scheduling, the central resource manager of the orchestration service does not have control over individual invocations. Thus, the invocations are handled by many per-application schedulers in a decentralized manner. Centralized serverless schedulers have many advantages over decentralized approaches in terms of scheduling quality [46]. OpenWhisk uses the coupled approach. A single scheduler is in control of the placement of every individual invocation.

4.1.3 OpenWhisk scheduling

OpenWhisk uses containers as its sandboxing mechanism, and so for the remainder of this chapter I use "container" to describe sandboxes. Figure 4.1 shows the system architecture of OpenWhisk. Within OpenWhisk there is a Controller component which manages scheduling. Individual worker nodes are called Invokers. When the Controller receives an invocation request,

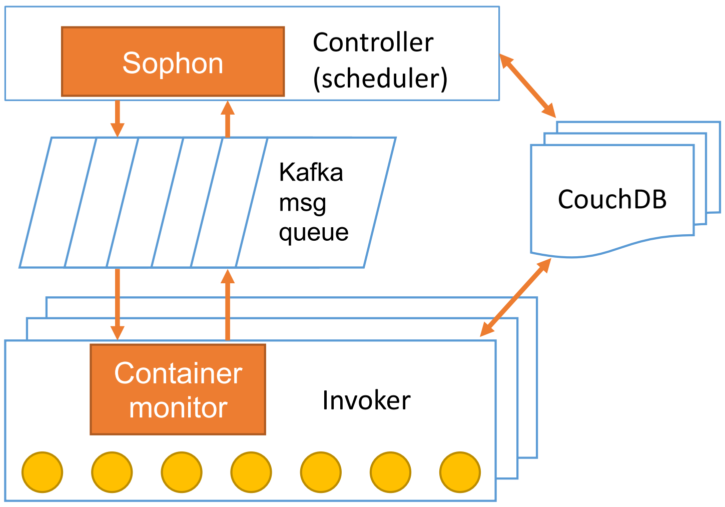


Figure 4.1. Architecture of OpenWhisk. The orange parts are the additions for Sophon.

it chooses an Invoker for the request via a default scheduling policy. The Controller and Invokers communicate via Kafka message queues. Function code and invocation results are stored in a separate CouchDB database.

OpenWhisk implements a "Sticky-Random" scheduling policy. It uses hashing to randomly schedule invocations to Invokers. By using the same hash values, it achieves the sticky property which tends to schedule the same functions to the same Invokers, improving the probability of warm starts. For each function, there is a Home Invoker and a step size parameter that is co-prime to the number of Invokers. Both are determined by the hash value of the function identifier. The scheduler checks each Invoker for available memory, and this available memory information is stored on the scheduler. The scheduler starts from the Home Invoker, and checks other Invokers by the step size until it finds an Invoker that has enough available memory for the function to execute there. Therefore each function is evaluated along an ordered sequence of possible Invokers, determined by a fixed "step size" interval. The closer the Invoker is to the front of the sequence, the more likely an invocation of that function has been recently scheduled to that Invoker, and the more likely scheduling the invocation in that location will result in a warm start.

Since the scheduler tracks node-level resource information, I call this style of scheduling

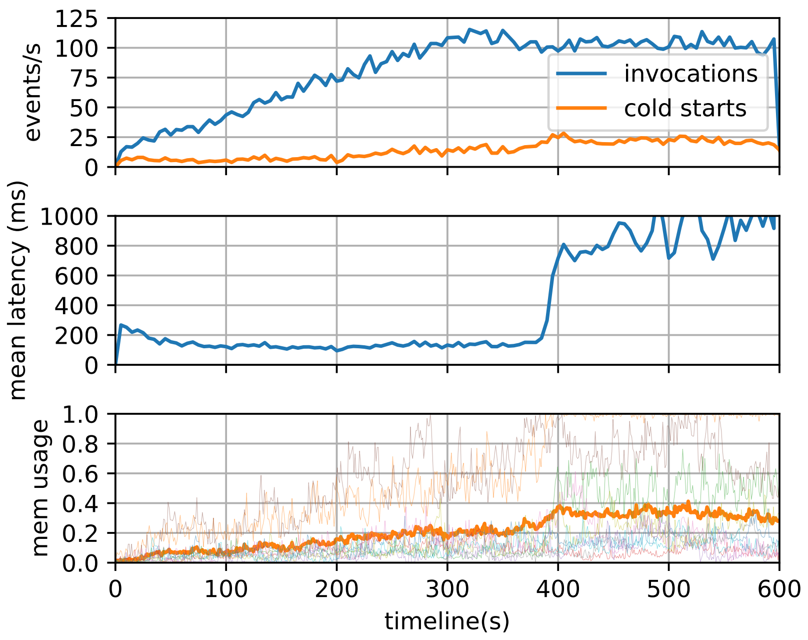


Figure 4.2. OpenWhisk tested using an excerpt workload from the Azure Function Traces. The top, middle, and bottom figures show function throughput/cold starts, average latency, and memory usage, respectively. For memory usage, the thick line represents average memory use while the other thin lines track the memory used on each Invoker.

"node-grained scheduling". The scheduler makes its best decision with the information it has, which might lead to more unexpected cold starts since the scheduler and the Invokers are not fully coordinated. After the request is scheduled and sent to the Invoker, the Invoker will use a warm container if one exists, otherwise it launches a cold container.

4.2 Scheduling Analysis

To better understand how OpenWhisk's Sticky-Random policy performs, I conduct tests that use real-world serverless traces recently made available from the Azure Function service at Microsoft [78]. Our testbed consists of a Controller node, which is an AWS EC2 c5n.9xlarge instance with 36 cores and 96 GB of memory, and 10 Invoker nodes which also use c5n.9xlarge instances. I configure 20 GB of memory on each node for the containers to use.

The Azure Function Traces [78] consist of two weeks of function invocation data from approximately 70,000 serverless functions deployed in the production Azure Function environment. The traces include workload distribution among functions (how many invocations belong to a function), per-function invocation patterns, execution duration, and associated memory consumption.

I randomly choose 10 continuous minutes of workload data from the traces and downsample it to match the much smaller scale of our OpenWhisk testbed. I downsample the traces by randomly choosing the invocations instead of choosing functions to better match the workload distributions of real systems. I sample approximately 50,000 invocations from 10 minutes of the trace and generate a representative workload that I deploy on OpenWhisk.

To prevent a synchronized burst of function invocations from overloading the system when it starts cold, I shape the workload into a slow-start curve that grows in the first half of the test. This slow start builds up warm containers. The second half of the test then has a steady invocation rate to reflect steady-state long-running systems that already have existing warm containers, as would be found in production.

Figure 4.2 shows the number of invocations, number of cold starts, mean latency, and memory usage (including memory usage of every Invoker in the thin lines) over time. The latency is defined as the time interval from when an invocation is posted to the scheduler to when the function starts running on an Invoker, which includes cold start latency and messaging delays. Memory usage is defined as the memory used by active functions divided by the total amount of memory that can be used for functions.

Initially OpenWhisk handles the increasing workload reasonably well. The sticky property of the Sticky-Random policy provides spacial locality for invocations. Most the of invocations are warm starts, and the average latency is low. However, when the workload has a high invocation rate, the average latency and cold start rate increase dramatically.

I note that the workload in the realistic Azure Function Trace is highly skewed: just five functions account for more than half of all the invocations. The function durations can vary from a few milliseconds to several minutes. Because of the sticky property of the Sticky-Random policy, the invocations are concentrated on Invokers with popular functions. The memory usage is highly imbalanced among the Invokers (as shown by the per-node load curves) even when the overall load is low (as shown by the average curve). When the workload has a higher invocation rate, some of the busy Invokers become overloaded and evict a lot of containers due to memory constraints. Container evictions cause future invocations to become cold starts, increasing memory usage, which in turn leads to more evictions. The functions start to compete for container resources and ultimately evict each other's warm containers.

This behavior is not unlike memory thrashing on a single node. Thus, I call this phenomenon *container thrashing*. The increase in cold starts not only directly increases invocation latency, it also burdens the underlying OS with more container creations, which incurs high creation delay for each container. The skewness in real-world serverless workloads leads to a pronounced load imbalance among Invokers, making the system more prone to container thrashing.

Takeaway. The default Sticky-Random policy performs reasonably well under a low invocation workload. Under higher load, however, the imbalance in memory use leads to container thrashing, significantly degrading performance.

4.3 Exploring Serverless Scheduling Policies

As shown in the last section, OpenWhisk's Sticky-Random policy performs sub-optimally under real-world serverless workloads. In this section, I explore several alternative policies tailored to warm-start locality, avoiding container thrashing, or a balance between the two. By exploring with these policies I obtain a better understanding of the requirements of serverless scheduling under real-world workloads, ultimately inspiring the design of Sophon.

4.3.1 Node-grained Policies

Random. Since the sticky property directly leads to memory imbalance in OpenWhisk's Sticky-Random policy, I remove the sticky property and implement a simple Random policy. In this policy each invocation is randomly scheduled to an Invoker that has capacity. Figure 4.3

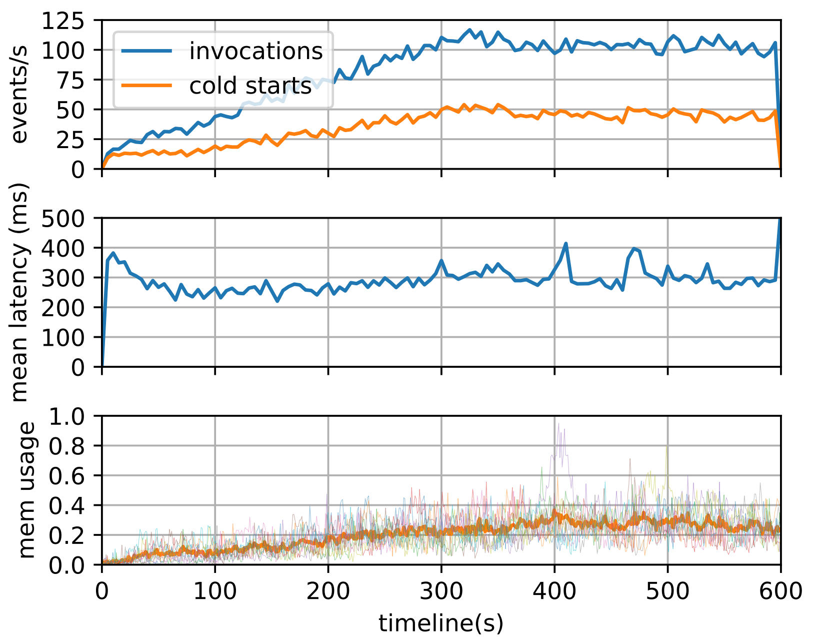


Figure 4.3. Number of invocations, number of cold starts, mean latency, and memory usage when scheduling using the Random policy.

shows the results of this policy using the same workload as in Section 4.2.

Because of the random nature of this policy, invocations of the same function are unlikely to be placed onto the same Invoker, leading to more cold starts (in fact the percentage of invocations that are cold starts is roughly constant throughout the experiment). Memory usage is more balanced across the Invokers than the Sticky-Random policy, which results in less container thrashing. The Sticky-Random policy outperforms Random in both latency and the average load when there is no container thrashing, but it performs worse once container thrashing happens.

Max Available. The Max Available policy chooses the Invoker with the most available memory, i.e., total memory minus memory used by active containers. Since OpenWhisk is designed in a way that the scheduler maintains the current available memory of each Invoker, the system does not need any additional probing messages to know the up-to-date available memory.

Figure 4.4 shows the results of scheduling using Max Available. Similar to the Random policy, the Max Available policy has more cold starts than the Sticky-Random policy because it does not provide the sticky property. Since the least-loaded Invoker is chosen for every invocation,

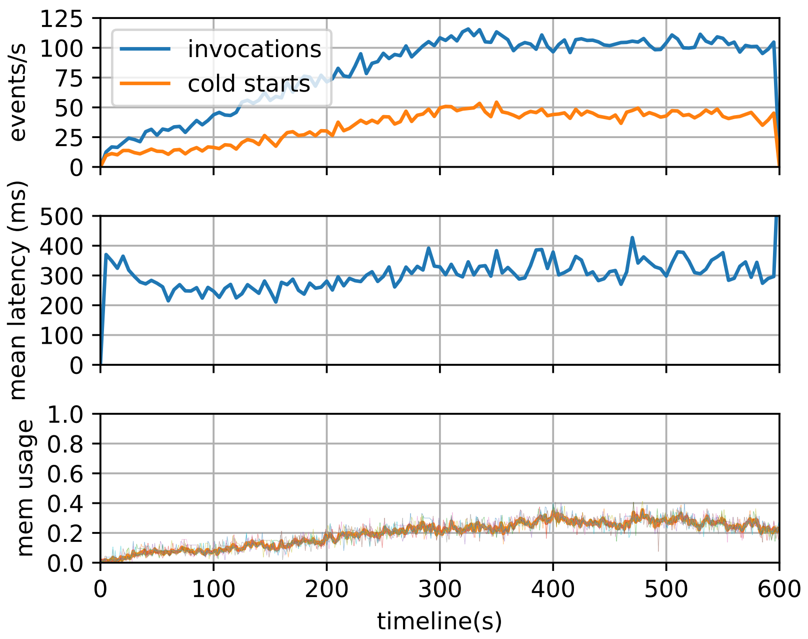


Figure 4.4. Number of invocations, number of cold starts, mean latency, and memory usage when scheduling using the Max Available policy.

memory usage is very balanced among the Invokers: the difference between the highest and the lowest memory usage is less than 10% most of the time. The total system memory use is higher than the Sticky-Random policy and lower than the Random policy. In traditional task scheduling, choosing a node with the minimal load often leads to a good policy. However, due to the extra overheads of cold start and container thrashing, it is not necessarily the case with serverless.

Sticky-Max Available. The Sticky-Max Available policy combines Sticky-Random with Max Available. Sticky-Random provides good spacial locality, reducing cold starts, but it has the disadvantage of being prone to container thrashing. Max Available can effectively reduce container thrashing, but causes many cold starts. The Sticky-Max Available policy balances these two factors. To encourage spacial locality, I use a distance factor d, which is defined as the distance from the current Invoker to the Home Invoker. I normalize d by dividing by the number of Invokers in the system. A smaller d indicates better spacial locality. To encourage better load balance, I factor in the load of the Invoker denoted by l. The policy chooses the Invoker that minimizes $\alpha \times d + (1-\alpha) \times l$ where α is a weight parameter. I find $\alpha = 0.3$ performs the best.

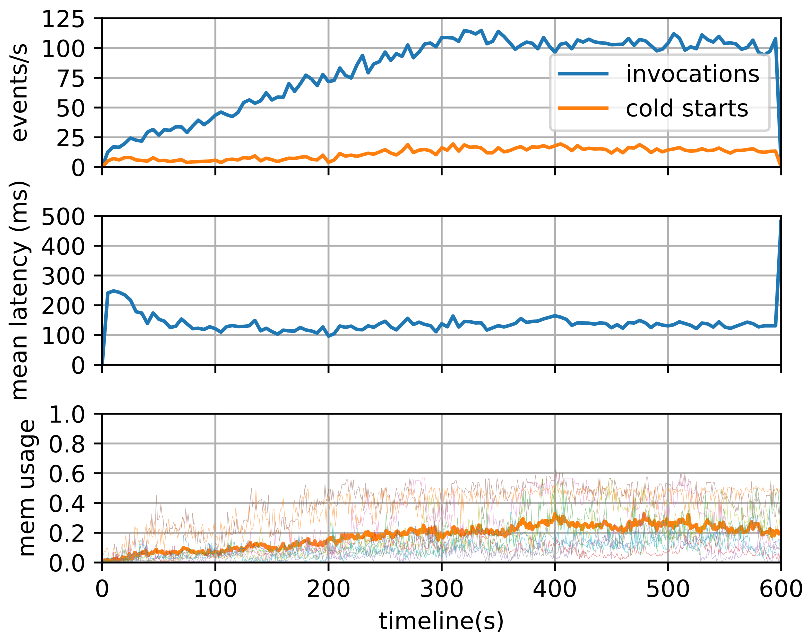


Figure 4.5. Number of invocations, number of cold starts, mean latency, and memory usage when scheduling using the Sticky-Max Available policy.

Note that this policy reduces to the Max-Available policy when $\alpha = 0$, and Sticky-Random when $\alpha = 1$.

Figure 4.5 shows the results. The Sticky-Max Available policy provides more balanced memory usage than Sticky-Random, reducing both Invoker hotspots and container thrashing. Under low load, it performs similar to the Sticky-Random policy since both policies preserve spacial locality. Under higher load, it outperforms Sticky-Random in both latency and memory usage by preventing container thrashing.

4.3.2 Discussion

A major goal of serverless scheduling is to maximize the use of warm containers while avoiding eviction of recently-used warm containers to prevent container thrashing. I find that although some of the policies can address parts of the scheduling problem like imbalance of memory usage, they cannot prevent container thrashing from happening.

The key observation is that the scheduler only has node-level resource information in making scheduling decisions. However, container-level resource information is required for the scheduler to make better scheduling decisions because container states play a crucial role in serverless system performance. An Invoker with low memory usage does not imply that scheduling a function to the Invoker is risk free. It can have warm containers that are recently used and would be evicted by a seemingly safe scheduling decision.

Since "available memory" (total memory−active container memory) is not sufficient for the scheduler to make improved scheduling decisions, one tempting possibility is to track warm containers by considering the "free memory" (available memory−warm container memory). Since eviction and container thrashing only happen when an Invoker is out of free memory, it could be tempting to conclude that avoiding hosts with low free memory can prevent container thrashing. However, keeping warm containers available for a long time can cause substantial waste in real-world systems [78]. Our experiments match previous findings, where containers of different functions quickly fill up the free memory, thereby making a free memory-based policy impractical. Moreover, different warm containers can have very different benefits. For example, a recently-used warm container can be much more useful than one idle for minutes. To schedule more effectively, I need to track the warm containers at a fine granularity instead of treating them indiscriminately.

4.4 Sophon: Container-grained Scheduling

I conclude that a serverless scheduler should take into consideration not only currently available resources, but also the states of containers when making scheduling decisions. To achieve this goal, I propose Sophon, a container-grained scheduling approach for serverless platforms. Unlike OpenWhisk, which tracks resources at the granularity of worker nodes, Sophon collects resource information at the granularity of containers. The Sophon scheduler maintains container information including its associated function ID, memory size, and last-used timestamp. Keeping track of container lifecycles provides better observability of system resources, and helps reduce resource competition among functions.

By tracking container states in Sophon, I can implement scheduling and container lifecycle policies that were not possible with existing node-grained scheduling. These new policies can be used to improve system performance in several ways. For example, with improved container visibility I can design policies to avoid container thrashing as discussed in Section 4.2. Moreover, the Azure Function Traces show that production serverless workloads have some unique invocation patterns. With container-grained scheduling, I can also design policies with these patterns in mind to further improve scheduling for these function workloads.

4.4.1 Sophon requirements and goals

This section discusses the requirements and goals of a scheduling policy for Sophon. As with the initial OpenWhisk scheduling policy, a Sophon policy starts with an available memory constraint. An Invoker needs to have at least the memory required by the invocation to be considered.

Since the Sophon scheduler now tracks the states of containers, the scheduler knows if a warm container exists on a certain Invoker. An intuitive decision is to choose Invokers that have a warm container available for a function. When there are multiple such Invokers, the scheduler then needs to choose one among them.

When there is no warm container on any Invoker for a function, a cold start is necessary. Then an Invoker needs to be chosen to create a container for the invocation. Since creating new containers can evict existing warm containers, the scheduler should also use its container visibility to carefully choose Invokers to avoid container thrashing by preventing recently-used containers from being evicted.

Depending on whether there exists a warm container, I divide a Sophon scheduling policy into two cases: 1) with warm containers, choose an Invoker that has one; 2) without a warm container, choose an Invoker to create a container. Invokers with warm containers always have priority over those without one.

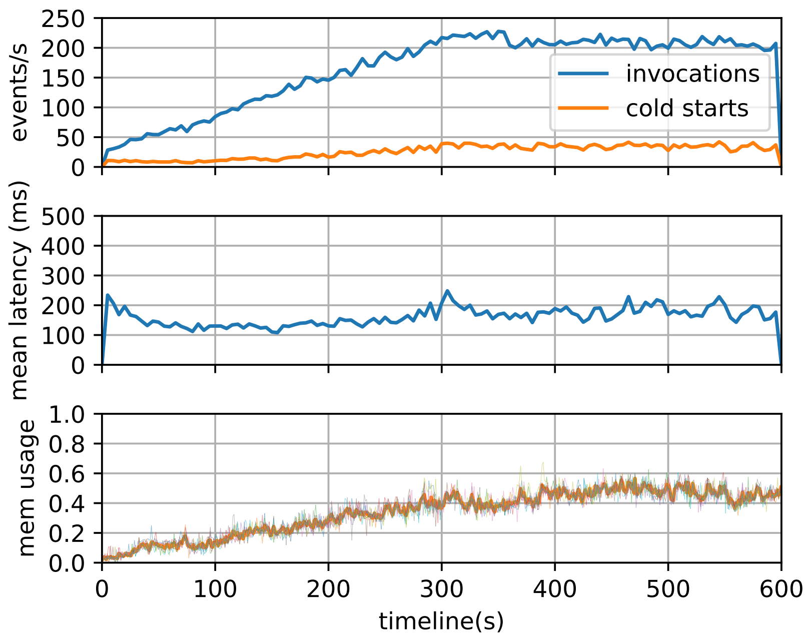


Figure 4.6. Number of invocations, number of cold starts, mean latency, and memory usage when scheduling using the Max-Available/Max-Available policy.

4.4.2 Sophon policies

Based on the discussions above, I explore several policies for Sophon. I first tested the Sophon policies using the same workload as Section 4.3 and found that they can support a much higher invocation workload. As a result, for these experiments I use the same methodology for sampling from the Azure Function trace to create a workload to schedule, but with double the invocation rate.

Max-Available/Max-Available (MA/MA). This policy handles warm starts by choosing the Invoker with the maximum available memory. Choosing the Invoker with max available memory can balance the load among Invokers more evenly. For cold starts, this policy also chooses the Invoker with the maximum available memory. This policy is different from the Max-Available policy discussed in Section 4.3.1 since it always chooses a warm container whenever possible.

Figure 4.6 shows the results of the MA/MA policy. The results show that the cold start activity is low, the memory usage is evenly balanced across the Invokers, and that mean latency is generally stable with a few small fluctuations.

Max-Available/Min-Cost Policy (MA/MC). This policy is the same as the MA/MA policy for warm starts but differs for cold starts. When a cold start is required, instead of choosing the Invoker with the most available memory, this policy makes a choice based on a cost metric. I define the cost of a cold start as the potential loss of a future warm start due to the eviction caused by the cold start. I model the arrivals of invocations as a memoryless Poisson point process. With this model, the probability of potential future invocations is then subject to exponential decay as the container stays idle. Therefore, a container's eviction cost is defined as $cost = \exp(-\lambda t)$ where t is the container idle time (in seconds), and λ is the decay rate; based upon our experimentation, I have found an empirical value of 0.3 to work well across a range of workload settings. If the scheduler decides no container will need to be evicted, then the cost reduces to 0. In this policy the Invoker with minimal cost is chosen when a cold start is necessary.

Figure 4.7 shows the results. Compared to the MA/MA policy, MA/MC exhibits more stable mean latency, but more imbalanced memory use among the Invokers. For cold starts, it chooses the Invoker with minimal cost, which can be different from the Invoker with minimal memory usage.

Sticky-Random/Max-Available (SR/MA). This policy is the same as MA/MA Policy for cold starts but differs for warm starts. It uses a simple sticky-random approach to choose among Invokers with warm containers. The rationale behind this choice is that, by choosing warm containers in a sticky-random manner, some warm containers are reused more often. As a result, the invocations are concentrated to a few warm containers to increase their utilization.

Figure 4.8 shows the results. SR/MA shows more balanced memory usage than MA/MC, but less balanced than MA/MA. The scheduling decisions of cold starts have more memory usage impact than warm starts, which is not surprising considering that cold starts are more costly. SR/MA has high variance of mean latency since the memory usage is less balanced than

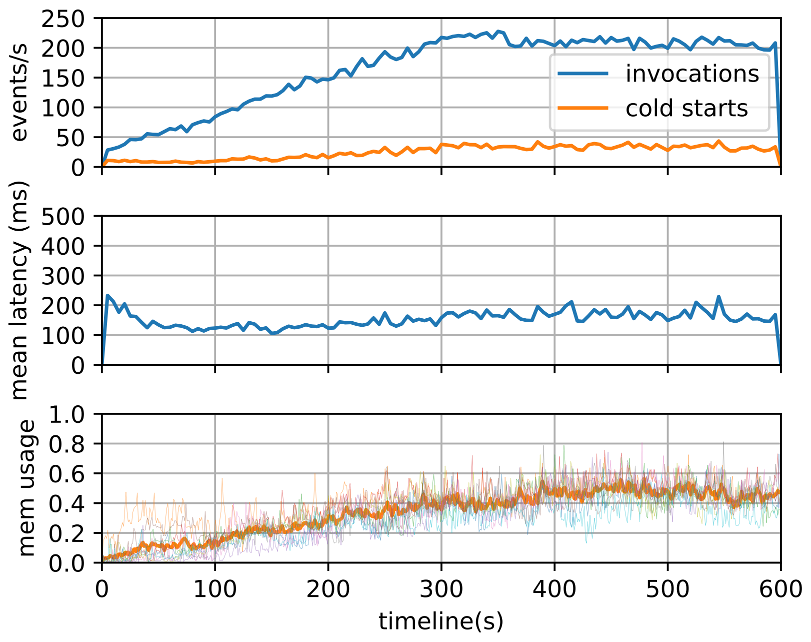


Figure 4.7. Number of invocations, number of cold starts, mean latency, and memory usage when scheduling using the Max-Available/Min-Cost policy.

MA/MA and it does not avoid container thrashing when choosing Invokers for cold starts.

Sticky-Random/Min-Cost (SR/MC). This policy uses the same sticky-random approach for scheduling warm starts as the SR/MA policy, while using the minimal cost metric of MA/MC to choose an Invoker for cold starts. Figure 4.9 shows that this policy has the most imbalanced memory usage, since memory use is not considered in either warm starts or cold starts. It has lower latency than SR/MA because the min cost policy for cold start prevents container thrashing.

4.4.3 Sampling

When scheduling for a large system, evaluating all candidate Invokers can be inefficient and unnecessary. Therefore, for large-scale deployments I sample Invokers, a common practice in many high performance scheduling systems.

In stateless schedulers like Sparrow [72], sampling requires probing host states. Such schedulers are usually conservative in the number of samples to reduce overhead. OpenWhisk does not need probing since the resource states are always updated to the scheduler. It only

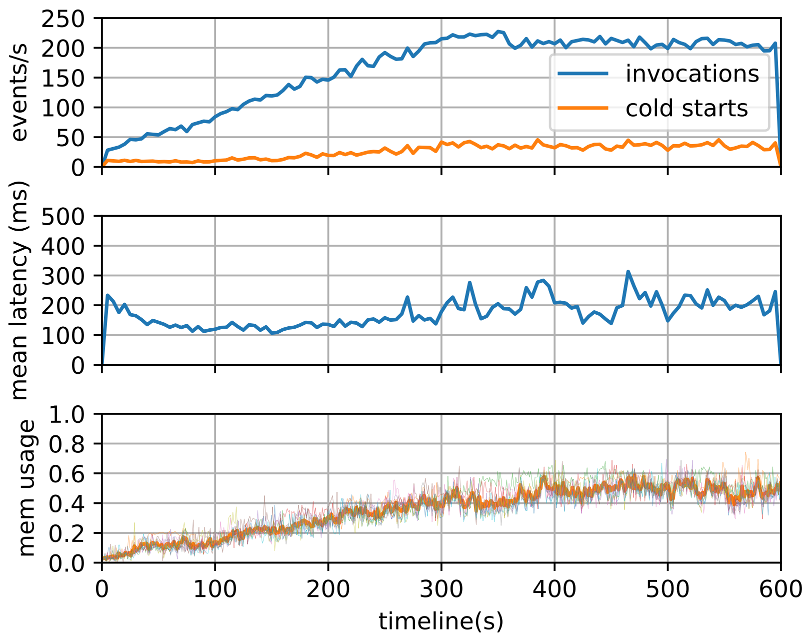


Figure 4.8. Number of invocations, number of cold starts, mean latency, and memory usage when scheduling using the Sticky-Random/Max-Available policy.

needs to evaluate the Invokers using the states already in the scheduler. Therefore, I can sample more candidates without generating extra network traffic. In our implementation, I sample 10 Invokers as I found it is sufficient for the scheduler to make high quality scheduling decisions while keeping the scheduling overheads low.

While Max-Available and Min-Cost can use random sampling for large deployments, Sticky-Random policy does not need to sample since it always chooses the first available Invoker from the Invoker sequence. I evaluate the scheduling quality and performance of sampling in Section 4.6.2.

4.5 Implementation

4.5.1 Integration in OpenWhisk

To incorporate container-grained scheduling into OpenWhisk, I extend its scheduler, messaging, and Invoker components. The Sophon scheduler maintains container information

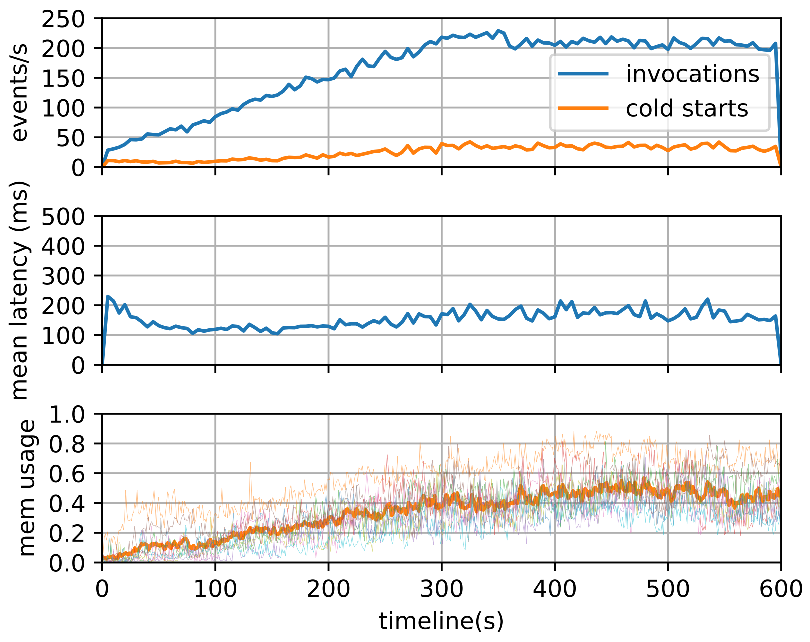


Figure 4.9. Number of invocations, number of cold starts, mean latency, and memory usage when scheduling using the Sticky-Random/Min-Cost policy.

including location, current running states, associated function ID, container memory size, and the timestamp of most recent use. I extend the messaging interface between the Invokers and scheduler so that they can communicate container information with each other. I also extend Invokers to monitor containers and report up-to-date container states to the scheduler. Sophon is implemented in about 1,200 lines of Scala code.

The scheduler receives container states reported by the Invokers whenever container states change. Since there is a delay from the state changes on the Invokers to when the scheduler receives the update, the scheduler can make scheduling decisions using stale states. Sophon does not rely on perfect container states to make scheduling decisions, and stale states can be tolerated. Once the Invoker detects a scheduling decision is made on stale states, it immediately sends the scheduler a state report so that the scheduler can updated states for future invocations. In practice, the state propagation latency is usually less than a millisecond. In our experiments I found only about 0.11% of the scheduling decisions use stale states, thus stale states are not a major concern.

4.5.2 Performance considerations

By tracking container states, Sophon may incur more overhead, including messaging overheads for the container state updates and time overhead in the scheduling process. However, I expect these extra overheads to be acceptable. OpenWhisk's scheduler already maintains invocation states and generates several messages for each invocation. Sophon's container state transitions use the same communication channels as OpenWhisk's messages. It can often piggy-back on the existing *invocation* and *acknowledgment* messages to propagate the container state transitions.

The Sophon scheduler maintains two mappings: a global mapping that maps a function to containers belonging to the function, and a per-Invoker mapping that tracks containers of each Invoker. The first mapping is used for the fast locating of warm containers when scheduling an invocation. The second mapping is for evaluating candidate Invokers for creating a cold container. For large scale systems with many Invokers, sampling can further reduce the scheduling overheads.

I evaluate the scheduling overhead in Section 4.6.2. I find Sophon's scheduling latency is on average 0.5 ms, which is similar to that of vanilla OpenWhisk. The scheduling latency accounts for a tiny portion of invocation latency (which is at least tens of ms).

4.5.3 Network Namespace Contention

OpenWhisk uses Docker containers as the default function runtime. The containers are created using the *bridge* network mode, in which each container has a network namespace that connects to the host network via a VETH device. Network namespaces are required in multi-tenant cloud environment to provide network devices and address isolation between containers. In OpenWhisk, each container has a proxy server that handles initialization and invocation requests sent from the Invoker that runs on the host. Different containers are identified by different IP addresses.

In our system experiments I found that, when there are concurrent creations and deletions of network namespaces, the creation performance degrades significantly: the creation rate reduces from 10/second to around 3/second on a single machine. This decrease is due to network namespace contention in the kernel, as the Linux network namespace implementation is not optimized for frequent creation and deletion [86, 68].

Since our work focuses on the serverless scheduling problem, I do not want to include the contention overhead introduced by the kernel network namespace implementation since it is an artifact orthogonal to a particular scheduling policy. Longer term, such contention can likely be removed by using alternative sandboxing methods like microVMs [1] that do not require network namespaces. Shorter term, for our scheduling policy evaluation I avoid the contention problem by using the *host* network mode, which does not create namespaces for containers. Instead of using different IP addresses for the containers, I assign different ports of the same host IP to the containers so that the Invoker can talk to each of them.

4.6 Evaluation

In this section I evaluate the performance of node-grained and container-grained scheduling by comparing different node-grained policies and Sophon policies. Our experiments show that container-grained scheduling achieves significantly better function execution throughput and invocation latency compared to node-grained scheduling. I show that Sophon achieves these performance benefits by greatly reducing the number of cold starts and avoiding container thrashing.

In the scheduler performance evaluation (Section 4.6.1), I use an EC2 c5n.9xlarge instance with 36 cores and 96 GB memory for the scheduler, and 10 c5n.9xlarge instances for the 10 Invokers. I configure 20 GB of memory on each Invoker for containers, providing a container pool of 200 GB in total. In the scheduling overhead tests (Section 4.6.2), I use two c5d.24xlarge instances, one for the scheduler and one for all the Invokers.

I use the OpenWhisk version with commit number `18bbac5` in the master branch of its repository committed on May 10, 2020. No major change has been made to the scheduler since then. I use the recommended Kubernetes-based deployment for managing the system components, including the Controller, Invokers, Kafka message queues, and the bookkeeping database. The containers are directly created by a local Docker daemon instead of via Kubernetes to obtain best performance. The scheduler is configured not to use "forced" scheduling, meaning that when the system is overloaded, requests queue on the clients until the system can accept them.

I use the Azure Function Traces for the workloads. I use the same subsampling method as in Sections 4.2 and 4.3. In Section 4.6.1, I primarily use two workloads, a low workload with about 100 requests/s and a high workload with about 200 requests/s. I use higher workloads to stress test the scheduler in Section 4.6.2. I use three randomly selected 10-minute intervals from the trace. I calculate the presented results by averaging (for numeric values like throughput) or aggregating (for statistical values like latency).

4.6.1 Scheduler Performance

System throughput. I first evaluate the maximum throughput of the node-grained scheduling policies and Sophon policies. I start the test with a low invocation rate and slowly increase the workload to reach the maximum throughput the system can support. Since the workload has inherent fluctuations, I smooth the result by averaging the throughput of 10-second time windows and report the highest value.

Figure 4.10 shows the results. All the Sophon policies significantly outperform node-grained scheduling policies, providing 74–76% improvements over the default OpenWhisk Sticky-Random policy. Among the node-grained policies, Sticky-Max Available performs the best, providing a 17% improvement over the Sticky-Random policy. Although in Section 4.3 the Random and Max-Available policies show more balanced memory load than the Sticky-Random

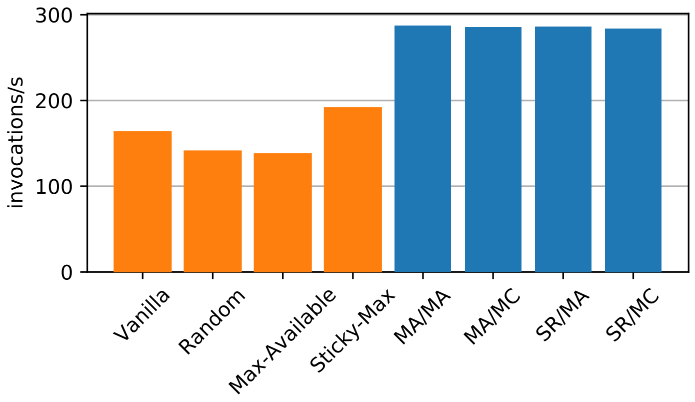


Figure 4.10. System throughput of node-grained policies (orange bars) and Sophon policies (blue bars). Among node-grained policies, Sticky-Max Available outperforms OpenWhisk's Sticky-Random policy. All the Sophon policies have at least 74% higher throughput than Sticky-Random.

policy, with higher workload their higher cold start rate quickly leads to container thrashing that degrades throughput. In constrast, the Sophon policies all have similar throughput.

Invocation latency. Invocation latency is the time between when the Controller receives a function invocation request to the time when the function starts executing on an Invoker. Low latency is important since the majority of serverless invocations last for a very short time. The latency comprises scheduling delays, messaging delays, and most importantly, cold start delays.

Figure 4.11 shows the CDF of all of the policies tested with the low workload (100 requests/s). Node-grained policies are shown with dashed lines while Sophon policies use solid lines. All Sophon policies have lower latency than the node-grained policies. The distributions have a discontinuity between 500–700 ms, which corresponds to invocations that require cold starts. The Random policy and Max-Available policy have the most cold starts, as indicated by the large discontinuity of their lines. The OpenWhisk Sticky-Random policy has a long tail, which is the result of container thrashing caused by unbalanced Invoker hotspots.

Figure 4.12 shows the CDF of Sophon policies tested under the high workload (200 requests/s). I did not include the node-grained policies in this test because they are unable to support the high workload. Different Sophon policies perform similarly for most of the requests.

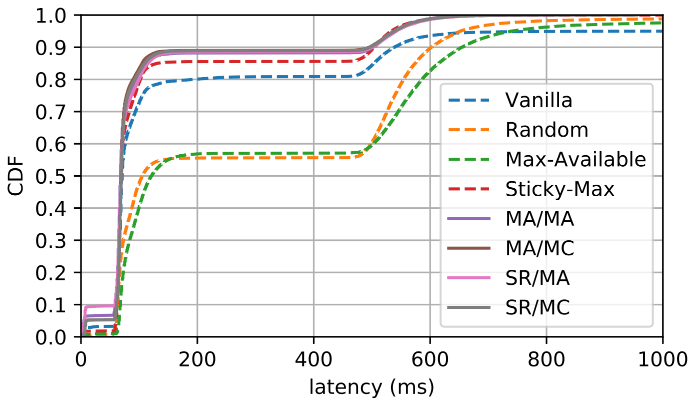


Figure 4.11. Function invocation latency of different policies under a 100/s request rate. Node-grained policies use dashed lines while Sophon policies use solid lines. All Sophon policies have lower latency than the node-grained policies.

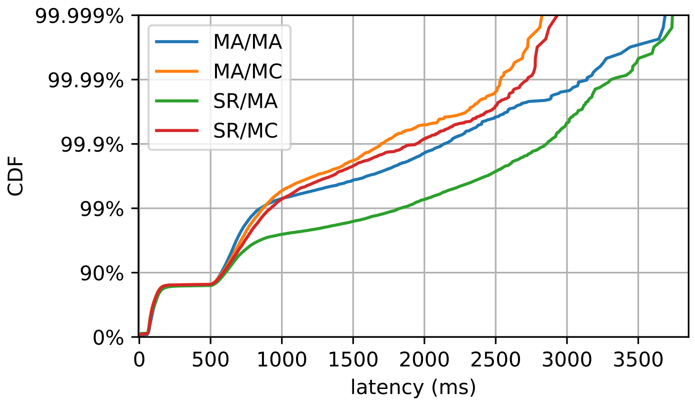


Figure 4.12. Function invocation latency of Sophon policies under a 200/s request rate.

Around 85% of the requests are warm starts, and have a latency of less than 100 ms.

There are a few percent of “straggler” invocations with much higher latencies than other invocations. For these requests, the latency of Sophon policies differ noticeably. Overall the MA/MC policy performs the best. Its 99% latency is 868 ms, less than half of that of the SR/MA policy. The mean latency is 164 ms for MA/MC, and 195 ms for SR/MA. The */MC policies have better latency than the */MA policies, while the MA/* policies have better latency than the SR/* policies. The */MC policies are better at preventing container thrashing by using the cost metric when scheduling cold starts. The MA/* policies tend to avoid Invokers with high load

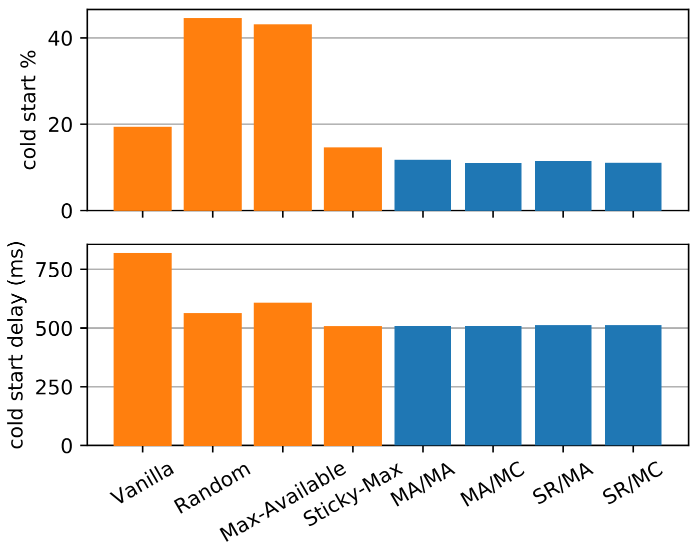


Figure 4.13. Percentage of cold starts and average delay of cold starts under 100 request/s workloads for the node-grained policies (orange) and the Sophon policies (blue).

when scheduling warm starts, which leads to more balanced load (fewer hotspots).

Cold starts. To understand how cold starts and container thrashing impact performance, I tracked the number of cold starts and the average container start delay for the various scheduling policies. I compute both the percentage of cold start relative to all invocations, and the average delay of the cold starts.

Figure 4.13 shows the invocation latency of all the policies under low load (100 requests/s). The Random and Max-Available policies have a high percentage of cold starts because they do not have the sticky property. The OpenWhisk Sticky-Random policy has a moderate percentage of cold starts but the highest cold start delay, which is due to container thrashing. All the Sophon policies have a low percentage of cold starts because they schedule to warm containers whenever possible, and low cold start delay because they significantly reduce container thrashing.

Figure 4.14 shows the invocation latency of the Sophon policies under high load (200 requests/s). The percentage of cold starts remains similar among the policies. The average cold start delays, however, are different. The SR/MA policy has the highest delay while MA/MC has the lowest. This difference is consistent with the latency results where MA/* policies outperform

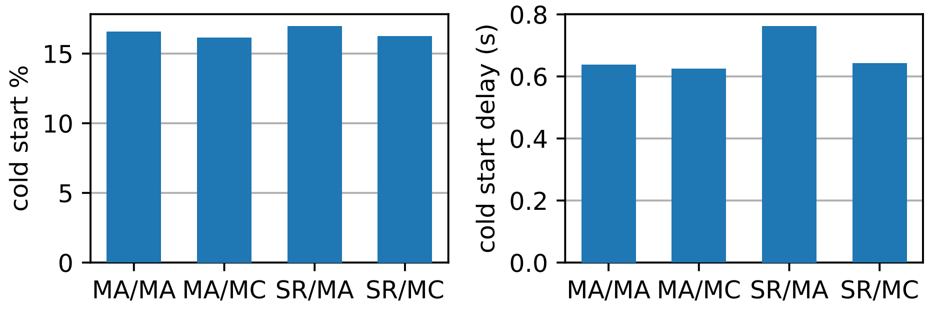


Figure 4.14. Percentage of cold starts and average delay of cold starts under 200 requests/s workloads.

SR/* policies, and */MC policies outperforms */MA policies. The differences in cold start delay caused by different degrees of container thrashing is the determining factor of invocation latency.

Memory usage. I compare the scheduling policies from the perspective of memory consumption. As before, I evaluate all of the policies under low workload (100 requests/s) and the Sophon policies under high workload (200 requests/s). I measure the average memory usage for the whole system (average memory consumption across all of the nodes), as well as the standard deviation across all of the nodes. For the standard deviations, I record the standard deviation of memory usage on the Invokers at every second and average them for the duration of the workload. The standard deviations reflect how well memory usage is balanced across the nodes.

Figure 4.15 shows the results. The bars represent the average memory usage and error bars represent the average standard deviation. Among the node-grained policies, the OpenWhisk Sticky-Random policy has both the highest memory usage and highest memory imbalance. This imbalance in memory consumption leads to significant container thrashing. As designed, the Max-Available policy has the most balanced memory use. The Sticky-Max Available and Random policies are in the middle. For the Sophon policies, the MA/* and */MA policies have more balanced memory use than the others. MA/MA provides the most balanced use, almost to the same degree as the Max-Available policy. SR/MC is the most imbalanced among the Sophon

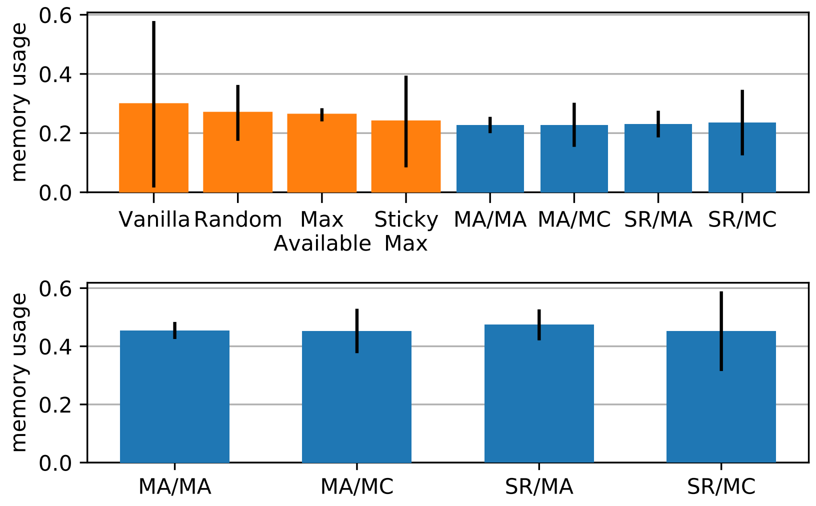


Figure 4.15. Average memory usage and average of memory usage deviation. The top figure shows results under 100 requests/s, while the bottom under 200 requests/s.

policies because, for both warm starts and cold starts, current memory use is not a direct decision factor. Although SR/MC has higher memory imbalance, it has less container thrashing than SR/MA under high load (Figure 4.14), which shows balancing memory usage should not be the only goal in scheduling cold containers, and avoiding container thrashing should be a priority.

Recommendation. At a high level, all of the Sophon policies significantly improve upon the various node-grained policies including OpenWhisk Sticky-Random policies. Compared to each other, the various Sophon policies exhibit similar throughput. Overall, though, the MA/MC policy provides the best latency, and thus is the recommended scheduling policy for Sophon.

4.6.2 Overheads

Sampling. Sampling reduces the computation overhead of the scheduling process. To show sampling does not negatively impact scheduling quality, I compare the scheduling quality of Sophon with and without sampling. I use a cluster consisting of 50 Invokers, each with 20 GB memory capacity, and sampling just considers 10 Invokers when scheduling. I schedule the trace workload with and without sampling and compare the maximum throughput and cold start percentage.

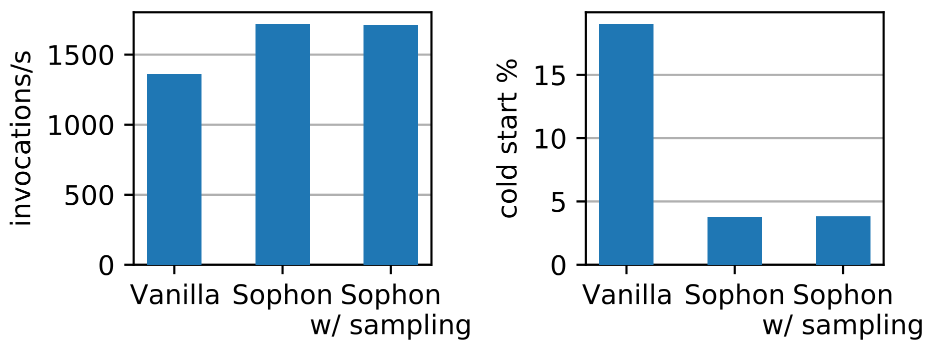


Figure 4.16. Throughput and cold start percentage comparison of vanilla OpenWhisk, Sophon without sampling, and Sophon with sampling, indicating the scheduling quality. The cold start percentage is measured under 900 invocations/s workloads.

Since 50 Invokers exceeds the capacity of our experimental testbed, I use a *container stub* implementation instead of the full Docker container runtime for the test. The container stub implements OpenWhisk's `Container` interface and emulates real container execution behaviors like cold star delays, initialization delays, and function call runtime. The container stub allows us to scale the number of Invokers to a meaningful regime for comparing sampling to complete information. Since the container stub does not have real workloads, I run all the 50 Invokers on the same c5d.24xlarge instance.

Figure 4.16 shows function throughput and cold start percentage of vanilla and Sophon with and without sampling. The maximum invocation throughputs with and without sampling are similar, both at around 1700/s. The cold start percentages are also similar. These results show that Sophon with sampling provides similar scheduling results as Sophon without sampling, and both perform better than vanilla OpenWhisk.

Scheduling throughput. More sophisticated scheduling policies come with computation overheads. I measure the scheduling overhead of Sophon's container-grained scheduling, with and without sampling, and compare it with vanilla OpenWhisk. I use container stubs to scale the system to 100 Invokers, each with 40 GB of memory. The capacity of the Invokers is higher so that I can stress test the scheduler performance instead of the Invokers. The workload is

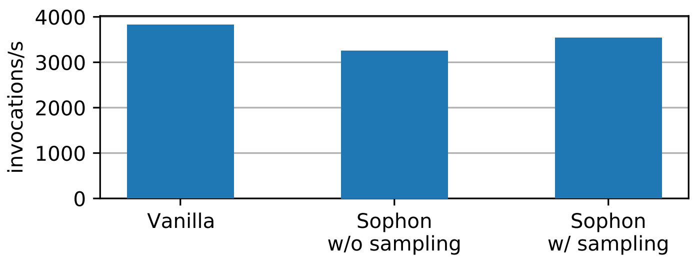


Figure 4.17. Scheduling throughput of vanilla OpenWhisk, Sophon without sampling, and Sophon with sampling.

generated from the Azure trace as before.

Figure 4.17 shows the results. The vanilla OpenWhisk scheduler has a maximum throughput of 3,828 invocations/s. Sophon's throughput is 3,255 invocations/s without sampling, and 3,542 invocations/s with sampling. Sampling reduces the scheduler's load by evaluating 10 Invokers instead of 100 Invokers, making Sophon's scheduler throughput closer to the simple Sticky-Random scheduling of vanilla OpenWhisk.

4.7 Related Work

Serverless platforms and scheduling. As discussed in Section 4.1.2, scheduling polices for serverless platforms fall into two categories based on whether sandbox scheduling and placement of function invocations are coupled or decoupled. Serverless systems that rely on container orchestration services like Knative [51], Kubeless [52], and OpenFaas [70] are examples of decoupled approaches.

Decoupled systems separate placement of the container from the scheduling of individual function invocations. The scheduling of invocations is decentralized, which can lead to under-utilization and resource contention for different functions since there is no global fine control over the load of each worker node.

Kaffes et al. [46] proposed the idea of core-grained serverless scheduling. Their proposal is to design a central scheduler to directly assign function invocations to worker CPU cores to avoid imbalances and contention. They assume the function code to be lightweight lean user code instead of container images and ignore the cold start latencies. However, cold starts remain a major issue in real-world serverless platforms [90]. Sophon is container-centric and takes into consideration container lifecycles to avoid cold starts.

Job scheduling and task scheduling. Traditional distributed job scheduling and task scheduling have been extensively studied in the research literature. Sparrow [72] is a distributed task scheduler that supports up to a million scheduling requests per second. However, since it targets a task scheduling scenario where the runtime environment is already set up and ready to accept requests, it cannot be directly applied to serverless scheduling where cold starts and setting up environments are a major source of overheads. Traditional cluster schedulers like Hydra [22], Mesos [39], and YARN [92] are designed for workloads with longer durations and lower request rates, and as such are not directly translatable to the serverless scheduling problem.

Cold start mitigation. Cold starts play an important factor in serverless performance, and as such significant attention has been paid to reducing serverless cold start overheads. Shahrad et al. [78] propose a hybrid policy based on histograms that use the history of serverless invocations to predict the arrival of future invocations. Based on these predictions, the system can evict containers to free up resources or pre-warm containers to reduce cold starts. This work is largely complimentary to Sophon since the invocation predictions can be used by Sophon to proactively create or evict containers to improve resource utilization.

FaasCache [32] tackles the single-server container keep-alive problem. They model container keep-alive policy as a cache replacement problem. The choice of which container to evict is based on a keep-alive priority calculated from recency of use, invocation frequency, cold start cost, and memory size. They use a hit-ratio curve to determine the optimal size of resources allocated to a function. FaasCache deals with local container provisioning and is complimentary to Sophon. Their keep-alive priority can be integrated into Sophon's Min-Cost policy to enable cluster-level container keep-alive policy.

OpenWhisk can be configured to use a pre-warming technique called "stem cell". It starts a container with a given memory size and language runtime. If an invocation arrives and finds a stem cell that has matching memory size and language runtime, it can use the pre-warmed container to skip the container startup process and start to install user code. The limitations of this pre-warming approach is that the platform needs to assume a quantized memory allocation coupled with the language runtimes of future invocations. Unfortunately, prewarmed containers may take memory resources that can be used for other containers, and do not support user-provisioned images. PCPM and Particle [64] tackle network overheads in container setup using pooling of network abstractions and lightweight network abstraction. SOCK [68] reduces container cold start time by using a combination of lightweight abstractions and pooling methods. SAND [2] improves function initialization latency by using per-application sandboxing instead of per-function sandboxing. Firecracker [1] and LightVM [60] are lightweight VMs that are tailored for serverless scenarios that require the rapid creation of sandboxes. The above approaches can reduce cold start overhead to various degrees, but cannot solve the general container cold start problem especially when the container image is large, and thus container cold start mitigation remains an open question.

4.8 Conclusion

Existing node-grained serverless schedulers ignore the states of warm containers, which causes functions to compete for container resources, increases cold starts and unnecessary evictions, and degrades performance. In this chapter I propose Sophon, a container-grained serverless scheduler implemented in OpenWhisk, that tracks container states to more carefully schedule requests, minimize container contention, and reduce the number of cold starts and evictions. Evaluations show Sophon can greatly improve maximum throughput of OpenWhisk while providing much lower latency for function invocations.

Furthermore, in the original OpenWhisk, GPSL applications are prone to container thrashing problem due to their higher resource demands. Sophon mitigates the container thrashing problem, which reduces the number of cold starts and reduces invocation latency. This is significant for GPSL applications like Sprocket since GPSL performance is sensitive to few slower stragglers that become performance bottlenecks.

Sophon reduces the number of cold starts, which benefits GPSL performance while improving system throughput, which can eventually reduce operational cost for users. However, in some cases a cold start is inevitable. In the next chapter I will describe how cold start latency can be reduced using snapshot-based virtual machines.

4.9 Acknowledgement

Chapter 4, in full, is currently being prepared for submission for publication of the material "Sophon: Efficient Container-grained Serverless Scheduling." Lixiang Ao, George Porter, and Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this material.

Chapter 5

FaaSnap: FaaS Made Fast Using Snapshot-based VMs

In Chapter 4, I introduced Sophon, which makes high-quality scheduling decisions to improve serverless system efficiency by avoiding cold starts as much as possible. However, in the case that cold starts are unavoidable, their latency leads to long delays for serverless execution. The cold start delay especially impacts GPSL performance since bursty GPSL workloads are likely to cause more cold starts, and the cold-start delays hurt overall performance since such stragglers often become the performance bottlenecks.

Both academia and industry have explored cold start mitigation using techniques like lightweight sandboxing [1, 60, 79], sharing of resources among instances [2, 27], or cloning pre-initialized environments in memory [13, 68, 89].

Virtual machine snapshots are a recent method developed to mitigate cold starts by restoring the guest VM to an existing warm initialized state to avoid the time-consuming steps of cold start like initializing runtimes and loading libraries [29]. Existing VM snapshot methods apply lazy loading of guest memory to avoid loading the whole guest memory when starting guest VMs. The memory pages are loaded from disk on-demand when accessed by the guest. However, the guest page accesses exhibit low spacial locality, leading to many major page faults and scattered disk reads that add significant overhead and slow down function execution.

Focusing on the active memory working set at the time when a snapshot is taken is a promising direction for improving snapshot and restore performance. By prefetching the working set when restoring a snapshot for a function invocation, slow major page faults and disk reads can be avoided during execution. Zhang et al. [96] explore this approach in the context of traditional virtual machine environments, scanning the access bits of page table entries to determine the recent working set of a guest VM at checkpoint time to reduce page faults after restoring. REAP [88] is the state-of-the-art for using working sets to accelerate snapshot restoring for FaaS. REAP assumes memory access is stable across function invocations and prefetches a compact representation of the working set pages of previous invocations when serving new ones. However, as I show, this assumption does not hold when the function input data differs significantly from previous invocations, leading to page accesses outside of the working set that slow down the invocation. In addition, the guest VM is paused until the entire working set has been restored, a problem especially troublesome for functions with large working sets.

Analyzing the snapshot behavior of FaaS functions, I present several notable observations on guest VM snapshots for FaaS. First, there is substantial differences in performance between major page faults and minor page faults, and the host OS page cache can be used to reduce the cost of major page faults. Second, due to the variance of function inputs and execution flows, the working set can change dramatically. Pages used by previous invocations should be treated as a reference for future invocations, and restoring a snapshot should efficiently handle invocations whose working set substantially differs. Third, a semantic gap between host pages and guest pages leads to unnecessary disk reads that slow down functions.

Based on these observations I propose FaaSnap, an efficient VM snapshot loading method that integrates several new techniques to reduce the cost of restoring guest VM snapshots and thereby improve overall FaaS invocation performance. Concurrent paging and working set groups avoid blocking function invocations while loading the working set, and opportunistically prevent slow disk reads by taking advantage of the host OS page cache. Host page recording relaxes the working set criteria to better tolerate variance in pages used by future invocations. Per-region mapping bridges the semantic gap between the host and the guest by handling memory pages differently based on their content. Finally, FaaSnap uses loading sets, a more compact working set definition, and an efficient layout of the loading set file that improves working set prefetching by removing unnecessary pages and consolidating disk reads.

I implement FaaSnap based on Firecracker, a lightweight VM tailored for serverless workloads. FaaSnap improves function execution by up to 3.5x compared to REAP snapshots. It is on average only 3.5% slower than snapshots cached in memory across a wide range of FaaS functions. FaaSnap is resilient to changes of working set across invocations, and remains efficient under GPSL-like workloads and when snapshots are located in remote storage.

5.1 Background and Related Work

5.1.1 Cold start problem

If the environment for a FaaS application does not exist when a function is invoked, the FaaS platform needs to initialize the environment for invoking the function, a process called cold start. The initialization steps include creating the virtual network, booting the isolation sandbox (usually a VM or a container), installing the function code, initializing the runtime, etc. The initialization steps take from several seconds up to minutes. A cold start is especially costly for FaaS since more than 50% of all invocations are less than 1 second, and 75% are less than 3 seconds [78]. The long cold start process can negatively impact both the performance of FaaS applications and system capacity. Therefore, many platforms keep the environment alive after an invocation finishes using an optimization called warm start. The warm state of the environment, including the runtime, loaded libraries, and accessed files, are kept in memory or in the page cache. In subsequent invocations, the environment and warm state are reused to accelerate function invocation.

However, there is a cost for keeping warm resources alive. Too many idle environments consume memory, reducing overall system capacity and throughput for invoking functions. Moreover, experience from large providers, such as the real-world traces from Azure [78], reveals that only a small portion of functions are invoked frequently. Less than half of the functions are invoked every hour, and less than 10% are invoked every minute. Cloud providers only keep the environment alive for a short period of time after an invocation is finished to minimize wasting resources; AWS Lambda keeps functions warm for 15-60 minutes [32]. As a result, invocations of hot functions are likely to have warm starts, while less frequent functions are prone to cold starts.

The cold start overheads can be generally divided into two parts, booting and initializing state. Booting a VM takes at least several seconds. Initializing memory state includes starting runtime, installing function code, loading libraries, etc., which can take seconds to minutes.

5.1.2 Booting time mitigation

Containers [58, 19, 81] are a more lightweight isolation mechanism than virtual machines. Containers use kernel abstractions like cgroups and namespaces to provide isolation among instances. The overhead of starting a container can be the same order of magnitude as starting a process.

However, in multi-tenant clouds like AWS and Azure, VMs are used as the sandboxing mechanism. VMs have a simpler, more stable guest-host interface than container's syscall interface and is considered more secure [60]. The downside is that VMs are traditionally designed for general-purpose guests. Full-fledged heavyweight VMs lead to long booting times for cold starts.

Unikernels have been explored to accelerate VM booting time [59]. Unikernels trim the guest kernel by customizing it for the application and eliminate the kernel-application boundary. LightVM [60] reduces the booting time to less than 10 ms by using a unikernel and a Xen VMM specialized for serverless computing. Faasm [79] uses lightweight language-based isolation to avoid the cost of virtualization.

Firecracker [1] is an open-source virtual machine monitor from Amazon and is used by AWS Lambda and other cloud services as a security sandbox. It uses a lightweight design that is tailored to serverless computing. The device emulation is minimized and BIOS is excluded to improve performance and reduce resource consumption. Firecracker can boot an unmodified Linux kernel in 125 ms.

5.1.3 Preparing memory state

In FaaS, not only does the isolation sandbox need to be created, the initialization of the OS, runtime, and libraries also takes significant time. Du et al. [26] report that the majority of cold start time in Google's gVisor is spent on the initialization of runtimes.

Copying existing memory state that has already been initialized is a method to skip the initialization step. Potemkin [89] proposed flash VM cloning and copy-on-write delta virtualization to quickly make copies of existing VMs. Snowflock [56] enables the cloning of VMs to remote hosts, lazily transferring guest pages to remote hosts when handling guest page faults. SOCK [68] is a container-based isolation system that creates copies of existing processes that have already initialized runtimes and libraries to skip the initialization step. SEUSS [13] takes in-memory snapshots of a unikernel guest and removes redundant execution paths, including initializing runtime and importing libraries, by serving invocations from the memory snapshots.

These systems require an existing VM or container for the function to exist in memory. This requirement is not always feasible in FaaS due to its memory resource consumption, especially for less frequently invoked functions.

5.1.4 Snapshots

Snapshot and restore is another method for avoiding booting and initialization overheads. The VM or container in-memory state can be saved to a file, and later the state can be restored from the file, skipping the initialization steps.

gVisor [36], a sandboxed container runtime by Google, supports checkpoint and restore. The latency of restoring a gVisor container can be a few hundreds of milliseconds due to the loading of guest memory and recovering guest kernel state. Catalyzer [26] uses optimizations including lazy memory loading and restoring some kernel state out of the critical path. The memory pages are read on-demand, reducing the initial wait time.

Firecracker recently introduced a snapshot and restore feature [29]. A Firecracker snapshot can be restored in a few milliseconds, a drastic reduction from typical cold start latencies. A Firecracker snapshot includes a snapshot file that stores the state of the VM like virtual devices and CPU registers as well as a memory file, which is the copy of the entire guest physical memory.

When Firecracker restores a snapshot, it loads and restores the VM state and maps the guest memory file to the VMM memory region that is provided to KVM (the virtual machine monitor in the host system) as the guest memory. Similar to Catalyzer, the guest memory pages are then loaded by the host on-demand when guest page faults happen.

While this approach to snapshot and restore improves performance, the cold start problem is still not entirely solved. Although the VM state can be restored and guest memory initialized in just milliseconds, to do any useful work, the guest needs to access at least a few thousand memory pages. Lazy restore only loads pages when the guest VM accesses them and creates page faults. As motivated by REAP [88], and our experiments also show in Section 5.2, a simple `hello-world` function takes more than 200 ms to execute using Firecracker snapshots, compared to a warm VM which finishes within 4 ms. Guest page faults are slow because the pages need to be read from disk, and the reads are small and scattered.

5.1.5 Working sets

Zhang et al. [96] proposed accelerating the lazy restore of VMs by eagerly prefetching working set pages. The working set is estimated by detecting which pages have been accessed recently before a snapshot. Their approach continuously scans the access bits in the page table to obtain the working set. When creating a snapshot, the working set pages are stored in a separate file. In the restore step, the working set pages are loaded sequentially and copied to the guest memory before the guest VM starts. This optimization decreases the number of future guest VM page faults. Halite [95] merges pages that are accessed together into locality groups. During restoring of the VM, when any page in the group is accessed, the whole group is prefetched.

REAP [88] is the state-of-the-art in optimizing serverless performance using working sets. Their observation is that serverless functions tend to access a stable set of pages across invocations. The idea is to prefetch pages accessed from previous invocations when an invocation starts. REAP uses `userfaultfd`, a kernel feature that allows user-level programs to handle page faults. It records the pages accessed in the first invocation into a working set file. In subsequent invocations, the working set pages are prefetched and installed in the guest memory, reducing the number of later page faults and disk reads. The working set pages are saved to a compact working set file and can be fetched in a single batch read, avoiding the cost of scattered page reads.

REAP works well for some workloads. However, as we show in Section 5.2, its performance is sensitive to changes in the working set. Invocation performance decreases when the working set differs significantly from the previous invocations because of change of input data, or when large amounts of anonymous pages are allocated in the guest. Both cases are common in real-world serverless functions.

5.2 Snapshot Analysis

To understand the overheads and challenges in snapshot-based function invocations, we measure several aspects of snapshot restoring of Firecracker VMs and REAP. We conduct our experiments using our FaaSnap platform, which we describe in more detail in Section 5.3.1.

5.2.1 Measurements

We measure the invocation of the following functions: a trivial `hello-world` function that replies with a "hello" string; a `read-list` function that reads every page of a large (512 MB) existing Python list; an `mmap` function that memory maps a large (512 MB) anonymous memory region and writes to every page of the region; an `image` function from FunctionBench [49], a comprehensive FaaS benchmark, that processes a JPEG image. `image-diff` is the same as `image` except it uses different inputs across invocations. In real-world deployments, inputs are most likely different across invocations.

We measure each function under four settings. Warm executes a function using a warm VM cached in memory that served a previous invocation. Firecracker executes a function by restoring a VM from a standard Firecracker snapshot memory file that was recorded and saved after a VM served a previous invocation. Cached executes a function similarly to Firecracker, but the snapshot memory file is loaded into the page cache so that there is no disk read overhead. While not practical in real-world deployments, it is a useful reference point for comparing snapshotting systems. We integrated REAP into our platform as an optional setting. REAP executes a function using a snapshot memory file together with a working set file created from a previous invocation, and it loads the entire working set file into memory immediately before executing the function.

For the time breakdown tests, we measure the time of the setup and execution steps for all functions. For additional insight, we also measure the time the guest VM spent handling page faults (`kvm_mmu_page_fault` kernel function) specifically for the `image-diff` function. We use bpftrace [77], a tracing tool based on eBPF, for the page fault measurements.

The host is an AWS EC2 `c5d.metal` instance with an Ubuntu Linux kernel version of 5.4.0. The disk is an NVMe SSD with measured maximum throughput of 1589 MB/s and 285,000 IOPS. The guest VM is configured with 2 GB of memory and 1 VCPU. Guest VMs use Debian Linux with a 4.14 kernel.

5.2.2 Time breakdowns

The time breakdown results are shown in Figure 5.1. The gray bars show the time taken to set up the VM, including starting the VMM, restoring virtual devices and CPU states, and for REAP, loading working sets. The primary color bars show the time to invoke the functions. As

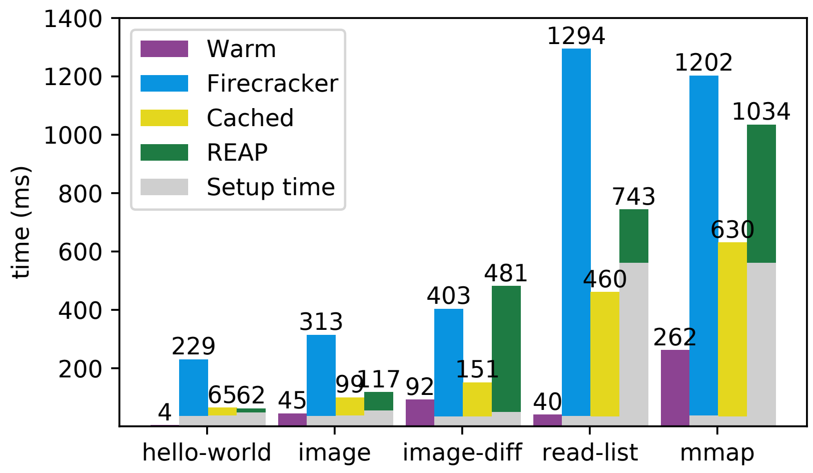


Figure 5.1. Time breakdown of function invocations. Primary color bars are function invocations. Gray bars are for VM setup, including starting the VMM, connecting virtual devices, restoring VM CPU state, etc.

expected, Warm outperforms all other settings. The `hello-world` function completes in 4 ms, much faster than all other settings. Warm is so fast because it does not have the overheads of restoring and setting up the VM, and it already has most of the guest VM state in physical memory.

Among the snapshot-based systems, Firecracker is the slowest. Firecracker uses OS on-demand paging, and a page is only read when accessed by the guest or prefetched when nearby pages are accessed. Small reads on the disk are slow (relative to memory) even on high performance NVMe SSDs. Cached has the best performance for all the functions except `hello-world`. Its contrast with Firecracker highlights the importance of avoiding costly disk reads in page fault handling. The function invocation times of Cached for the `image` and `image-diff` functions are close to that of Warm. For the `read-list` and `mmap` functions, however, Cached is significantly slower because, although Cached avoids all the major page faults that read from disk, minor page faults are still needed to install the page table entries for the pages in the host OS page cache.

REAP performs well for the `hello-world` and `image` functions, where the function is supplied with the same input data as the previous invocation. The invocation time is similar to that of Cached and Warm. However, in `image-diff` where the input data differs for the second invocation, its performance degrades. REAP relies on the VM using a stable set of pages across invocations. When the pages used are significantly different, it handles the pages that are not in the working set file at user level, reducing performance. The `read-list` and `mmap` functions have a large working set. As a result, the setup step takes a long time to load and install the working set. Once installed, though, invocation becomes fast. The `read-list` and `mmap` invocation steps are faster for REAP than Cached because the pages are installed into the host page table by REAP's `userfaultfd` handler.

The `mmap` function allocates anonymous memory in the guest. However, because the whole guest memory is mapped to the host memory file, the host does not know the guest is allocating anonymous memory. As a result, allocating guest memory causes disk reads on the host, which is much slower than allocating from anonymous memory on the host.

5.2.3 Page fault behavior

The distribution of page fault handling times is shown in Figure 5.2. We test the `image-diff` function invoked on the four systems. The *x* ticks represent times spent handling a page fault, and each bar between ticks counts the number of page faults whose time falls into that interval. Note that both axes are in log scale.

Warm has around 4,000 page faults, while all the snapshot-based systems have around 9,000. Warm VMs have many of their pages already loaded into physical memory, and the page faults are caused by accessing new pages not touched in the first invocation. Since warm VMs are booted from VM images, and the guest memory region is mapped to host anonymous memory, the warm page faults are quickly handled using anonymous memory, which is faster than file-backed mappings that go through the page cache layer. The average time is 2.5 microseconds, and more than 90% of the warm page faults take less than 4 microseconds. The total time of handling all the page faults is 12 ms.

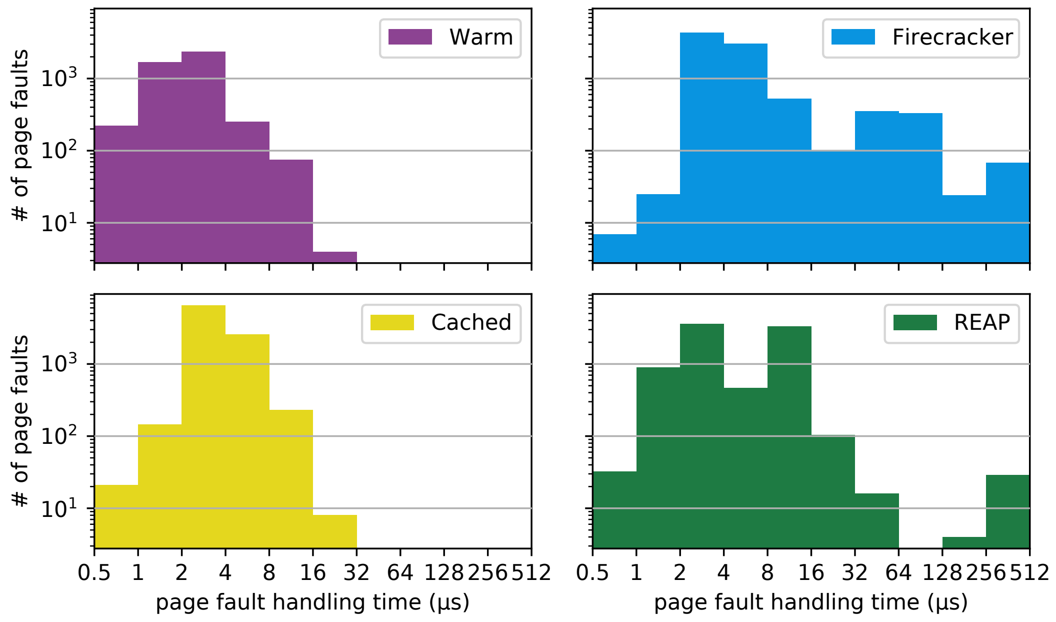


Figure 5.2. Distributions of page fault handling time for `image-diff` under different settings. Both axes are log-scale.

Cached handles more than 90% of the page faults in less than 8 microseconds, and the average time is 3.7 microseconds. All the Cached page faults are minor page faults that are served by the page cache. The time to handle page faults for Cached takes slightly longer than that of Warm because it has to access the page cache layer. The total page fault handling time is 35 ms.

Firecracker is the slowest among the four systems, with an average page fault time of 13.3 microseconds. Nearly 9% of the page faults take more than 32 microseconds, which are slow major page faults that read from disk. When handling a page fault from disk, the `readahead` mechanism in the host kernel fetches pages near the faulting page into the page cache to reduce future disk reads. The page faults shorter than 32 microseconds are mostly minor page faults served from the page cache, and they show a distribution similar to that of Cached. The total page fault handling time is 120 ms.

REAP page faults have an interesting distribution. Pages from the working set file are installed into the host page table by `userfaultfd` at the beginning of the invocation. Page faults on these pages are processed in less than 4 microseconds since the host page table entries already exist. Page faults outside of the working set causes the userspace `userfaultfd` process to read from the original memory file. Depending on whether the page in the memory file has been prefetched into the page cache, the handling can be relatively fast (8–64 microseconds) or slow ($>$128 microseconds). Userspace `userfaultfd` adds an overhead of several microseconds to each page fault outside of the working set. The average page fault time is 6.7 microseconds, and the total page fault handling time is 56 ms. Although REAP handles pages faults faster than Firecracker, its execution time is longer than Firecracker. With REAP, the guest cannot immediately resume after a page fault is handled, causing context switches that slow down guest execution.

5.2.4 Summary

We make several observations from these experiments: (1) The performance difference between Firecracker and Cached, and the time difference between handling minor and major page faults, highlight the benefits of caching pages in memory. The OS page cache can play an important role in accelerating VM page faults; (2) The working set of a function can change dramatically due to changes in input data or function execution flow. Therefore the page accesses in previous invocations should be treated as a reference, and restoring a snapshot should efficiently handle invocations whose working set substantially differs; (3) The anonymous page allocation in the guest memory is translated to unnecessary file-backed page fault on the host because of the semantic gap between the guest and the host.

5.3 FaaSnap

Based on the observations from Section 5.2, we propose FaaSnap, a snapshot loading mechanism that handles real-world FaaS snapshot paging more efficiently. We introduce several techniques and optimizations to improve various aspects of snapshot loading. We first describe the system design, and then detail each of the optimization techniques in turn.

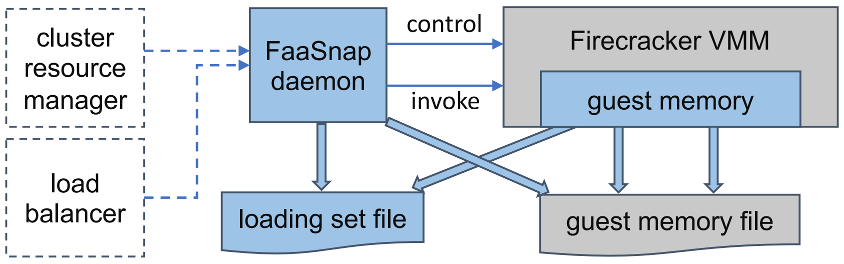


Figure 5.3. High level system architecture. Blue elements are FaaSnap components, and gray elements are existing Firecracker components. Dashed components are expected to interact with the FaaSnap daemon in real-world FaaS deployments, but are not needed in our use of FaaSnap.

5.3.1 System design

Figure 5.3 shows a system diagram of FaaSnap. The blue elements are FaaSnap components and the gray elements are existing Firecracker components. The dashed components are expected to interact with the FaaSnap daemon in real-world FaaS deployments, but are not needed in our experimental use of FaaSnap.

The FaaSnap daemon is the core system component of FaaSnap. It communicates with the Firecracker VMM and manages related resources, and it forwards invocation requests to the VMs. It is similar to the "MicroManager" component in Firecracker deployments in AWS [1]. The FaaSnap daemon manages local VM images, guest kernels, snapshot memory and working set files, active VMs, and network resources like namespaces and virtual network devices. All of the techniques described in this section are implemented in the FaaSnap daemon except for the provisioning of the guest memory. FaaSnap also exposes an API to allow remote clients to control resources and send invocation requests. In real-world deployed FaaS systems, the remote clients would be load balancers that route invocation requests and cluster-level resource managers that control the VM lifecycles.

We modify the Firecracker VMM for the provisioning of FaaSnap guest memory.

5.3.2 Concurrent paging

As shown in Section 5.2, default on-demand paging is costly because many slow VM page faults that need disk reads occur during an invocation. REAP, on the other hand, prefetches all of the previously accessed pages to avoid page faults during an invocation. The downside of this approach is that it results in a long initial loading step that blocks the invocation process. The problem is even more pronounced when the working set is large (Figure 5.1).

Instead of blocking the VM while waiting for the prefetch to complete, the FaaSnap daemon starts the VM immediately after setup, similar to the original Firecracker implementation. Once the daemon receives an invocation request, it starts a loader thread to prefetch the pages from the working set recorded in earlier invocations. FaaSnap starts the loader as a thread in the daemon instead of a thread in the Firecracker VMM so that it can start prefetching immediately when the daemon receives the invocation request, and does not need to wait for the VMM to start executing.

Thus FaaSnap supports concurrent page faults from both the VM and the FaaSnap loader. If a page is first accessed by the loader, the page fault handler will read and install the page into the page cache. When the page is later accessed by the VM, the page will be served from the page cache, resulting in a faster minor page fault instead of a blocking major page fault. Pages first accessed by the VM cause a blocking major page fault. Although it cannot guarantee that all the working set pages are first read by the FaaSnap loader, we show in Section 5.5 that concurrent paging reduces a significant portion of major VM page faults while removing the need for a long initial read step that blocks VM execution.

5.3.3 Working set group

To move disk reads out of the critical path of VM page fault handling, the daemon loader needs to prefetch pages before the VM accesses them. Ideally, the loader should access the pages in an order similar to that of the VM so that the loader has a higher chance of prefetching a page before the VM. A straightforward idea is to record the order of page accesses in the first invocation, and let the loader access the pages in the same exact order. However, loading the pages using the previous access order exhibits poor locality, leaving the Linux `readahead` mechanism ineffective in its prefetching. For the `image-diff` function execution, for instance, reading by access order takes the loader 100 ms longer than sequential address order, which slows down populating the page cache with prefetched pages.

Instead, FaaSnap uses an approximate order for loading. It divides the working set pages into several working set groups by their access order: e.g., the first N accessed pages are assigned group 1, the next N accessed pages are assigned group 2, etc. The loader reads groups in increasing group number, and reads pages within a group sequentially. In this way, the loader is more likely to access a page earlier than the guest while preserving disk access locality when loading. From our experiments, we find N = 1024 works well across the function benchmarks and use this value in our evaluations. Note that the working set group is different from Halite [95], where locality groups are used to predict clustering of pages instead of ordering of accesses.

5.3.4 Host page recording

To determine the working set of the guest VM, in previous work Zhang et al. [96] scanned the access bits of the page table entries and REAP used `userfaultfd` to record the address of every faulting guest page. In both methods, the working set obtained is limited to the faulting *guest pages*. However, the host kernel `readahead` mechanism fetches extra pages on each page fault that are not tracked with `userfaultfd` or access bits. While tracking only the faulting pages is a natural design decision, we find that, when handling invocations with different inputs, it improves performance if the working set includes not only the faulting guest pages but also the *host pages* cached by `readahead`. The reason is the pages touched by `readahead` can be accessed in future invocations when function inputs are different and the working set changes. In other words, `readahead` can “predict” some future guest memory accesses even if the pages are not touched in previous invocations.

Instead, FaaSnap uses the `mincore` syscall to construct the working set file. `mincore` scans the *present bits* in the page table entries to determine if pages in a memory range are present in memory. In our case, it detects if guest pages are in the host page cache. By calling `mincore` repeatedly, FaaSnap records the new pages loaded since the last `mincore` call. It assigns a working set group number to pages using the order they appear in the `mincore` scans. As an added benefit, `mincore` has lower overhead than `userfaultfd` for recording working set pages since it does not need to invoke a user-level process to handle and record a page fault. By including present pages instead of just accessed pages in the working set, FaaSnap is more tolerant of changes in the working set.

5.3.5 Per-region memory mapping

The Firecracker snapshot implementation maps the entire guest memory to the guest memory file. As a result, all guest page faults, including anonymous page faults, are translated into more costly file-backed page faults on the host, which degrades performance as shown in Section 5.2.

In the guest kernel, an anonymous page is initially attached to a read-only all-zero page. Any write to the anonymous page traps to the guest's copy-on-write page fault handler, which copies the zero page into a newly allocated guest physical page. This page copy traps into the file-backed page fault handler in the host kernel, which issues a disk read request. However, the read is unnecessary since the page is being overwritten with zeros.

The nature of the problem is a semantic gap between the host and the guest. The host kernel does not know the cause of the guest page fault. If the host kernel knows the guest is trying to access a newly allocated page, it can happily serve the page in the anonymous region. One solution to the problem is to use a paravirtualized kernel to explicitly provide the host kernel the cause of the page fault so that the host can handle it accordingly.

We choose a simpler approach. Instead of memory-mapping the entire guest memory file, FaaSnap only maps the pages that are non-zero to the guest memory file. Zero pages are instead `mmap`'d to anonymous host memory. Although guest zero pages are not necessarily anonymous pages in the guest, using anonymous memory in the host still guarantees that the pages are initialized to zero, thus providing correct semantics. When an invocation is finished, FaaSnap scans the guest memory file, merging consecutive zero pages into *zero regions* and non-zero pages into *non-zero regions*. A region is also assigned a group number, which is the lowest group number of any page in the region. During an invocation, our modifications to the Firecracker VMM `mmaps` zero regions to anonymous memory, and non-zero regions to the memory file. In this way, a page fault on the guest zero page will be handled by host anonymous memory instead of triggering a slow disk read.

Another optimization is the handling of freed pages. If the guest kernel frees a guest physical page, its contents no longer matter and it will be overwritten to zero when allocated again. However, Linux does not actively clear the contents of a freed page. The host has no way to know if a page has been freed and no longer needed. We modify the guest kernel so that it always *sanitizes* freed pages, writing zeroes to the page. As a result, FaaSnap excludes the freed pages from the set of non-zero pages, and future accesses to the freed pages will be fast anonymous page faults.

5.3.6 Loading set

As discussed above, the FaaSnap daemon uses `mincore` to determine the working set of a function invocation during the record phase. Since the working set often includes zero regions, which FaaSnap maps to anonymous memory, the loader does not need to prefetch the zero regions during the restore. We define the *loading set* as the working set pages excluding the zero pages. The group numbers of the loading set regions are derived from the working set. As a result, the loader only needs to prefetch the loading set regions.

The VMM needs to `mmap` every loading set region when setting up the VM for an invocation. Even for a simple `hello-world` function, there can be more than 1000 loading set regions, and the overhead to create large numbers of mappings is not negligible. However, we find that many loading set regions adjacent in the guest address space are only separated by a few non-loading set pages (i.e., either zero pages or non-working set pages). FaaSnap merges these adjacent regions by including the pages in between them. This relaxation greatly reduces the number of regions that need to be separately mapped, while only adding a small amount of additional data read. The distance threshold for merging two regions is empirically set to 32 pages, a value that reduces the number of regions to small enough while not adding too many unneeded pages. For `hello-world`, merging regions reduces the number of regions to less than 100, while total amount of data increases by only 5%.

5.3.7 Loading set file

Our experience is that, across a variety of functions, the loading set pages tend to be scattered throughout the guest physical address space. Scattered reads, though, usually lead to lower disk performance. Similar to REAP, FaaSnap stores the loading set into a compact file that only contains the loading set pages so that the loader reads it more efficiently.

In contrast to REAP, though, FaaSnap sorts the loading set regions first by their group numbers, then by their addresses. The file offsets and sizes of the regions are cached in the FaaSnap daemon. When a function is invoked, the FaaSnap daemon tells the VMM to `mmap` the loading set regions using the recorded file offsets and lengths. The loader then reads the loading set file in sequential order, caching the pages that are scattered in the guest address space. When the guest VM accesses those pages, they result in a minor page fault that installs the page in the guest. Furthermore, as described in Section 5.3.2, FaaSnap supports concurrent paging so that reading the loading set file does not block function execution.

5.3.8 Summary

Table 5.1 summarizes the four types of pages in the guest VM memory. The loading set pages (i.e., non-zero pages in the working set) are mapped to the loading set file using offsets

Table 5.1. Types of pages and their mapping in FaaSnap.

Type	Non-zero	Working set	Mapping
Loading set	Y	Y	loading set file
Cold set	Y	N	memory file
Released set	N	Y	anonymous
Unused set	N	N	

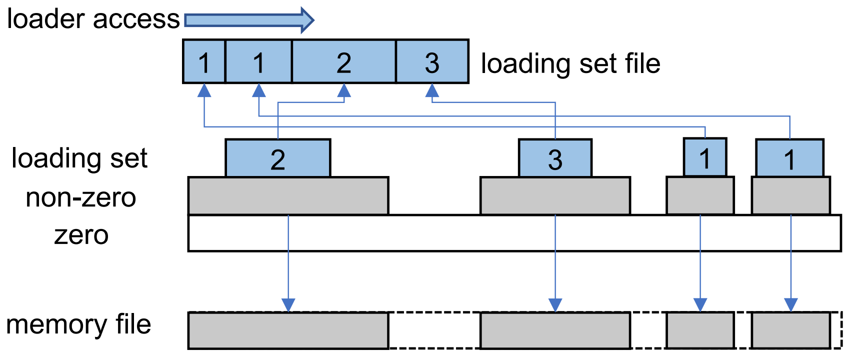


Figure 5.4. VMM guest memory mappings and backing files. Mappings are created from the bottom layer to the top layer using the overlapping semantics of the kernel, where upper layers override the pages of lower layers. Zero regions are mapped to anonymous pages (white bar). The cold set (non-zero regions not in the working set) is mapped to the memory file (gray bar). The loading set is mapped to the loading set file. The numbers in the loading set denote group numbers, lower group numbers are stored in earlier locations in the loading set file. The loading set file is read in sequential order.

recorded when the loading set file was created and used by the loader in the daemon during subsequent function invocations. The cold set are non-zero pages not accessed during the first invocation. These pages are usually more than 100 MB in size, and most of them are pages used in the guest booting process. The cold set are mapped to the original memory file using the same offset as in the guest memory. They are less likely to be accessed during the next invocation, so they are not included in the loading set file, but they still need to be mapped to the original memory file to ensure memory integrity in case they are accessed. The released set are zero pages touched in the first invocation, and are primarily pages freed by the guest kernel. The unused set are zero pages that are never touched. Both the released and unused sets can be safely mapped to anonymous memory.

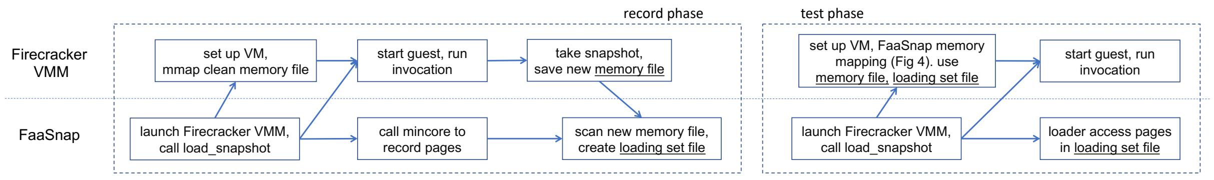

record phase
test phase
Firecracker VMM
FaaSnap
set up VM, mmap clean memory file
start guest, run invocation
take snapshot, save new memory file
launch Firecracker VMM, call load_snapshot
call mincore to record pages
scan new memory file, create loading set file
set up VM, FaaSnap memory mapping (Fig 4). use memory file, loading set file
start guest, run invocation
launch Firecracker VMM, call load_snapshot
loader access pages in loading set file

Figure 5.5. Flow chart of FaaSnap snapshot and restore.

One way to map these regions is to make non-overlapping `mmap` calls for each individual region. However, we can reduce the number of `mmap` calls by mapping smaller regions on top of existing ones in a hierarchy. First, an anonymous region for the entire guest address space is mapped. Then non-zero regions are mapped to the same offset in the memory file. Finally, the loading set regions are mapped to pre-recorded offsets in the loading set file. Figure 5.4 shows the VMM memory structure and the corresponding mapped files.

FaaSnap combines all the techniques described above, and Figure 5.5 shows a flow chart for the steps. In the first invocation, or record phase, the VM is started from restoring a "clean" snapshot. FaaSnap obtains the working set groups using repeated `mincore` syscalls to the memory file. After the invocation, a new snapshot is created to store the warm state. FaaSnap then scans the new memory file to find non-zero pages. The loading set is the intersection between the working set and non-zero pages. Adjacent loading set regions are merged to reduce the number of regions. The loading set is then stored into a compact loading set file in the order of group numbers and the region offsets are recorded.

In a subsequent invocation, or test phase, FaaSnap will use the new memory file and loading set file. Concurrent paging uses the host OS page cache to reduce guest major page faults. Per-region memory mapping allows different sets to be handled separately to improve performance. Overlapping `mmap` calls are used to simplify the mapping process. With all the optimizations combined, FaaSnap significantly reduces the guest VM's page fault handling time on the critical path.

5.4 Implementation

The majority of FaaSnap is implemented in the daemon with minor parts implemented in the Firecracker VMM for the per-region mapping technique. FaaSnap consists of ∼2,500 lines of Go code (not including REAP integration) and ∼200 lines of Rust code in the Firecracker VMM.

Code for functions is installed as Python files in the guest VM. We built a Flask-based server that runs in the guest and waits for HTTP invocation requests and invokes function code. The FaaSnap daemon supports operations like creating functions using installed images and kernels, booting VMs for a function, invoking functions on the booted VM, taking snapshots of a VM, restoring snapshots, etc. FaaS applications rely on external storage to store state, including input, output, and intermediate data, that persists beyond the lifetime of a function invocation. We run an in-memory Redis data store on the host for external storage for functions.

The daemon starts and manages the Firecracker VMM upon receiving user requests through an API. It communicates with Firecracker using HTTP via Firecracker's Unix sockets. Once the guest VM is started, it uses the guest IP address to connect to the Flask server (running in the guest) for invoking functions.

In the record phase, the daemon calls `mincore` on the mapped memory repeatedly to check for newly accessed pages to implement host page recording. Since the daemon only needs to record the 1024 recently accessed pages in a group, it waits for the guest to allocate enough new pages before calling `mincore`. The daemon polls `procfs` for the resident set size (RSS) of the guest. Once the RSS has more than 1024 new pages, it calls `mincore` to record them.

We extend the API call between the daemon and Firecracker VMM with additional arguments that specify the locations of non-zero regions and loading set regions. The daemon first allocates an anonymous region. It then uses the `MAP_FIXED` flag to place the overlapping non-zero and loading set regions onto the exact offsets of the anonymous region. The daemon then provides the whole memory region to KVM to use as the guest memory by issuing an `ioctl` call.

We modify the `free_pages_prepare` function in the guest kernel to sanitize freed pages. Sanitizing pages imposes overhead for the guest kernel (around 10% of execution time). Since sanitizing freed pages is only necessary during the record phase, we disable page sanitizing in the test phase. At the end of the record phase and before creating the snapshot, the FaaSnap daemon sends an HTTP request to the guest daemon, which signals the guest kernel to disable page sanitizing via the `procfs` interface.

The REAP developers generously support their system as an open-source project, and we integrated REAP as an optional mode for evaluation purposes. When receiving an invocation, the FaaSnap daemon registers the snapshot with REAP and activates REAP in a goroutine, which waits to receive the `userfaultfd` file descriptor through which it handles guest VM page faults on behalf of the kernel.

5.5 Evaluation

We evaluate the performance of FaaSnap, including function execution time, input size sensitivity, execution breakdown, the contributions of different optimizations to performance, and performance under bursty workloads and remote disks.

5.5.1 Methodology

Table 5.2 lists the functions we use for evaluation. The first three are synthetic, and the rest are from FunctionBench [49], SeBS [20], and Sprocket [7]. The functions cover a wide range of applications including web requests, multimedia, scientific computing, machine learning, and graph processing. To reflect the expected scenario where the same function will have different inputs in different invocations, we prepare two sets of inputs. For functions with variable inputs, input A is smaller than input B. There are two phases in a test, a record phase and a test phase. The first invocation happens in the record phase, whose warm state in the guest memory is stored in the snapshot or the working set/loading set file. The test phase is the actual test. We use input

Table 5.2. Functions used in the evaluation. Different inputs are used in the record and test phases to get realistic results.

	Description	Input A	Input B	Working Set A	Working Set B
Hello-world	a minimal function	n/a	n/a	11.8 MB	11.8 MB
Read-list	read an 512 MB Python list	n/a	n/a	526 MB	526 MB
Mmap	allocate anonymous memory	512 MB	512 MB	536 MB	536 MB
Image	rotate a JPEG image	101 KB JPEG	103 KB JPEG	20.6 MB	32.6 MB
Json	deserialize and serialize json	13 KB json	148 KB json	12.7 MB	14.4 MB
Pyaes	AES encryption	string of 20k	string of 22k	12.6 MB	13.2 MB
Chameleon	render HTML table	table size 30k	table size 40k	22.9 MB	25.1 MB
Matmul	matrix multiplication	matrix size 2000	matrix size 2200	113 MB	133 MB
FFmpeg	apply grayscale filter	1-sec 480p video, 338KB	1-sec 480p video, 381KB	179 MB	178 MB
Compression	file compression	13 KB file	148 KB file	15.3 MB	15.8 MB
Recognition	PyTorch ResNet image recognition	ResNet-50 cnn, 101 KB JPEG	ResNet-50 cnn, 103 KB JPEG	230 MB	234 MB
PageRank	igraph PageRank	graph size 90k	graph size 100k	104 MB	114 MB

A in the record phase, input B in the test phase, and vice versa to evaluate the expected scenario where the input size grows or shrinks.

We drop the page cache of all the relevant files, including the snapshot memory file and the working set file, before each test to ensure we measure performance when the pages are actually read from disk.

Our measurement platform is an AWS `c5d.metal` instance with a 96 vCPU Intel Xeon Platinum 8275CL CPU running at 3.00 GHz, 192 GB of memory, and 25 Gbps network bandwidth. It runs Ubuntu Linux with a 5.4.0 kernel. Each guest VM has 2GB of memory and 2 vCPUs, a typical configuration in real-world FaaS systems like AWS Lambda. The guest uses Debian Linux with a 4.14 kernel. The disk is an NVMe SSD with measured maximum read throughput of 1589 MB/s and IOPS of 285,000.

5.5.2 Execution time

We first evaluate overall function execution time for different snapshot methods, which includes both guest VM setup and function invocation time. We measure the time to execute the functions listed in Table 5.2 using Firecracker, Cached, REAP, and FaaSnap snapshots. Cached snapshots preload the snapshot memory file into the page cache before execution. While not practical, we use it as a reference for other systems. We run each test five times and show the average and standard deviation.

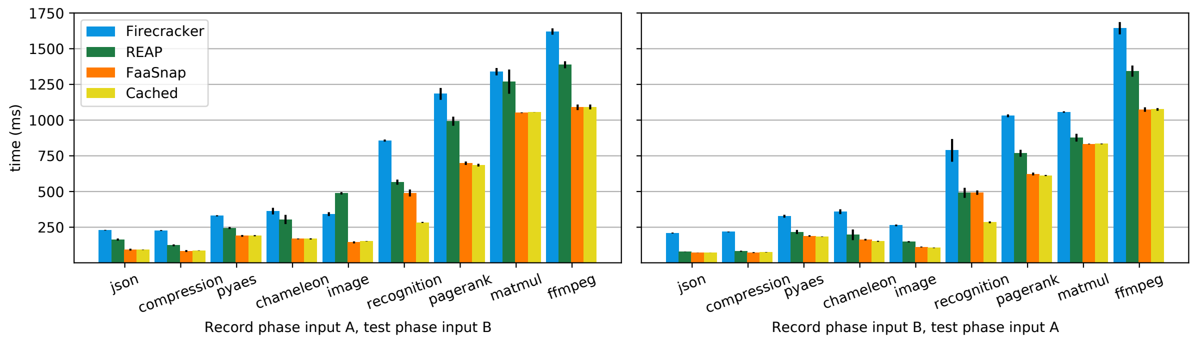


Figure 5.6. Execution time of the benchmark functions. Standard deviations are shown in error bars. Cached is used as a reference. FaaSnap shows similar results to Cached: most of the VM page faults are minor page faults, and the slow disk reads are taken out of the VM page fault critical path.

Figure 5.6 shows average execution time and standard deviation for the benchmark functions. The left subfigure uses input A in the record phase and input B in test phase, and the right subfigure reverses the inputs. Figure 5.7 shows the results of the three synthetic functions. These functions have the same input (or no input) for the record and test phases, and therefore are shown separately.

FaaSnap has the shortest execution time for all the functions compared to Firecracker and REAP snapshots. On average, it improves upon Firecracker by 2.0× and it improves upon REAP by 1.4×. Note that for the benchmark functions, FaaSnap's speedup over REAP is higher when the test phase uses larger input B (1.55x) than when the test phase uses smaller input A (1.16x). The reason is that larger inputs in the test phase trigger more page faults outside of REAP's

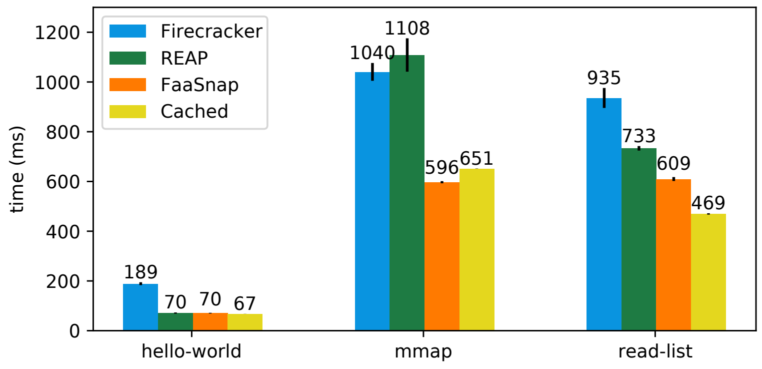


Figure 5.7. Execution time of the three synthetic functions.

working set file (which was created with a smaller input), and these page faults are handled with more overhead at user level via REAP's use of `userfaultfd`. FaaSnap's per-region memory mapping, loading set, and host page recording techniques help it handle the workloads not captured by the working sets more efficiently. FaaSnap's concurrent paging also prevents the initial long blocking of functions with large working sets.

Moreover, the performance of FaaSnap is close to Cached snapshots for most functions. FaaSnap loads most of the loading set pages to the host OS page cache before they are accessed by the guest VM. Performance with FaaSnap can sometimes even be faster than Cached because the per-region memory mapping technique allows page faults for zero pages to be handled in anonymous memory, which is faster than faulting from the page cache (Section 5.2). Cached snapshots outperform FaaSnap in the `read-list` and `recognition` functions because of their access patterns. These functions read existing pages aggressively, and the FaaSnap loader cannot always keep pace with the page faults created by the guest. On average FaaSnap snapshots are only 3.5% slower than Cached snapshots: FaaSnap provides performance using an SSD nearly equal to that of the in-memory page cache.

5.5.3 Input size sensitivity

To further evaluate the sensitivity of the snapshot methods to input variation, we perform another series of experiments. For each of the functions in Figure 5.6, we use inputs of the same

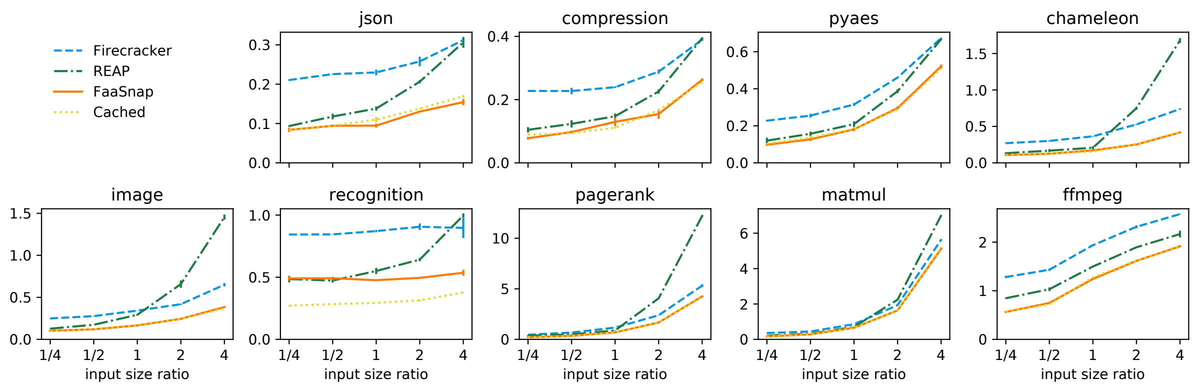


Figure 5.8. Execution time under varying input size ratios. All y-axes are in seconds. FaaSnap performs similarly to Cached for most functions, indicated by the orange solid line overlapping well with the yellow dotted line. REAP performance degrades when input size ratios are larger, shown by the steeper lines than others when the ratio is larger than 1.

sizes as input A in the record phase, and then vary the sizes of the inputs in the test phase (whose contents are also entirely different). In particular, we use inputs in the test phase whose sizes are 1/4$\times$ to 4$\times$ the size of the input in the record phase. We run each test three times and report averages and standard deviations.

Figure 5.8 shows the results of this experiment. Each graph shows the results for one of the nine functions that take variable inputs. Each curve in the graph corresponds to a different snapshot technique. Each point on a curve shows the execution time of a function for a particular ratio of input sizes in the test and record phases. In the graph for `ffmpeg`, for example, the point on the blue dashed line at $x = 4$ shows the execution time of `ffmpeg` using Firecracker when the size of the input video for the test phase is 4$\times$ the size of the input video used in the record phase to take the snapshot.

These results show that FaaSnap snapshots provide performance benefits across the range of input size ratios. As with the results in Figure 5.6, FaaSnap outperforms Firecracker and REAP for all of the functions, and performs similarly to Cached except for `recognition`. In contrast, for many functions REAP execution time significantly increases when the input size is larger than that of the record phase, especially for `chameleon`, `image`, and `pagerank`. At these larger input sizes, REAP performs worse than Firecracker for many of the functions.

Table 5.3. Performance analysis.

	Total time	Fetch time	Fetch size	Guest pagefault size	Page fault waiting time
REAP, ffmpeg	1408 ms	257 ms	201 M	20 M	780 ms
FaaSnap, ffmpeg	1070 ms	107 ms	146 M	32 M	866 ms
REAP, image	480 ms	51 ms	22 M	31 M	342 ms
FaaSnap, image	136 ms	55 ms	88 M	7.2 M	109 ms

FaaSnap, on the other hand, handles changing input sizes well and its benefits are resilient to changes in working set. Relative to Firecracker, the benefit of FaaSnap snapshots is roughly constant across differences in input sizes, effectively providing the performance of having the pages that benefit the test phase prefetched into the cache. Note that the impact of this benefit on overall execution time does decrease for larger input size ratios. As function execution time becomes dominated by the input itself, working set optimizations including FaaSnap are going to provide diminishing returns. For these situations, the goal of a system taking advantage of working sets is to help when it can but otherwise avoid degrading function execution time, which FaaSnap achieves but unfortunately REAP does not.

5.5.4 Performance analysis

To provide more insight into the performance differences between FaaSnap and REAP, we examine the execution breakdown of the `ffmpeg` and `image` functions in more detail. They show different behaviors under the two systems, and are representative of other functions. We collect metrics including total execution time, working set fetch time, working set fetch size, guest page fault size, and page fault waiting time. The page fault waiting time includes both the page fault service time (`kvm_mmu_page_fault`) and the time KVM waits for the guest CPU to be ready to run (`kvm_vcpu_block`). We collect the numbers using bpftrace [77] and `perf` and report them in Table 5.3.

While FaaSnap outperforms REAP when executing both functions, it does so for different reasons. For `ffmpeg`, the benefit with FaaSnap mostly comes from a shorter fetch time. REAP's fetching process includes not only synchronously reading the working set, but also installing the pages via `userfaultfd`, which slows down execution. FaaSnap starts the function concurrently with fetching, avoiding the initial fetching delay.

For `image`, FaaSnap fetching is slower than REAP. `image` with FaaSnap has a larger relative working set size, which is caused by the sparse access pattern of `image` resulting in more pages being recorded in its loading set under FaaSnap than in the working set under REAP. Despite faster working set fetching, though, REAP is much slower when running the function because of a much longer page fault waiting time: when serving each page fault, KVM blocks to wait for the guest CPU to be ready, resulting in extra context switches that increase waiting time. In contrast, FaaSnap has far fewer page faults and handles them in the kernel. This benefit, together with concurrent paging, makes `image` perform 3.5× faster on FaaSnap.

5.5.5 Optimization steps

To understand how the different optimizations contribute to the overall performance improvements of FaaSnap, we selectively measure incremental contributions of the optimizations. Starting with Firecracker as the baseline, we also measure FaaSnap using just concurrent paging (Section 5.3.2), then the combined optimizations that together support per-region mapping (Sections 5.3.2 to 5.3.5), and finally all FaaSnap optimizations combined. We focus on the `image` benchmark and measure the execution times as well as the number of page faults, total page fault handling time, and the number of disk read requests caused by VM page faults. We collect the data using the bpftrace [77] tool.

Figure 5.9 shows the results. The daemon loader uses concurrent paging to prefetch pages concurrently with the execution of the guest VM, which reduces the number of major page faults, total page fault time, and number of block read requests from the guest VM. Per-region mapping, however, has *more* major page faults and *fewer* block requests and a lower page fault handling time. These seemly conflicting numbers are the result of different paging orders. In concurrent paging, the FaaSnap loader reads the working set pages in the address space order,

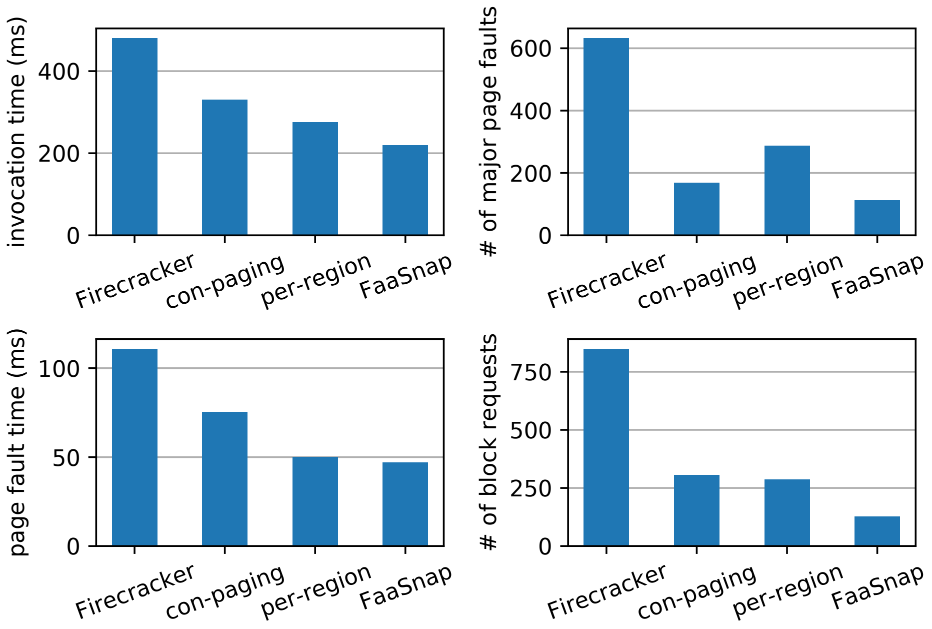


Figure 5.9. Optimization steps and their effects.

which does not correspond to the exact guest VM page access order. When the guest VM has a major page fault, it usually causes a disk read. Therefore, the average time of serving a page fault with just concurrent paging is high.

In comparison, with per-region mapping the daemon loader prefetches the pages in approximately the same order as the guest VM because of its use of working set groups. When a guest VM major page fault happens, it is more likely that the faulting page is being installed to the page cache from the disk by the daemon loader and does not cause another disk read. Therefore, the VM page fault handling time is shorter in per-region mapping, making the VM execution faster. Per-region mapping creates more major page faults since it is able to progress faster, but its major page faults are less “harmful” than major page faults in concurrent paging.

FaaSnap further reduces the loader’s read time using the loading set and loading set file optimizations. The loader can prefetch most of the pages before the guest accesses them, leading to the fewest number of major page faults, fewest number of block read requests, shortest page fault time, and shortest invocation time.

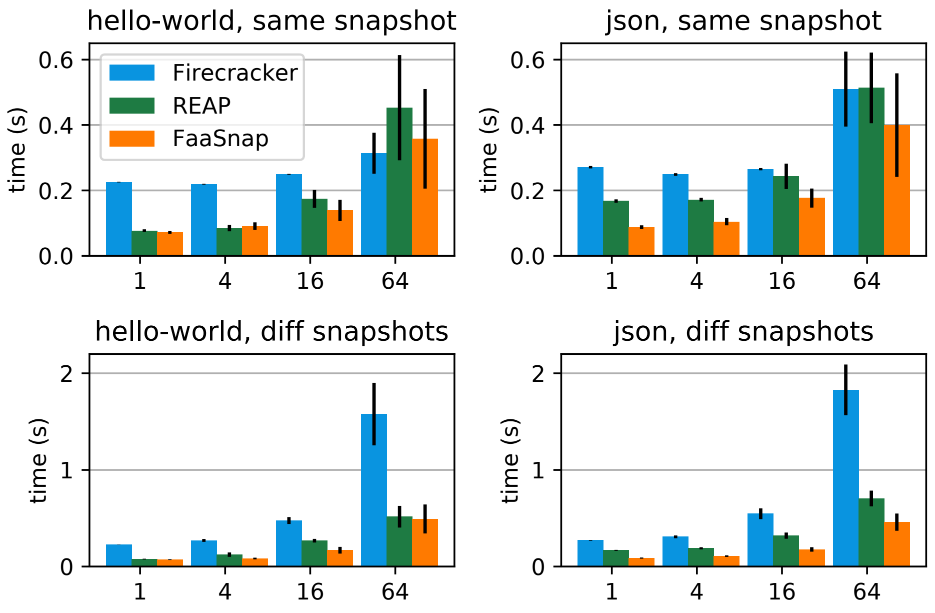


Figure 5.10. Performance with bursty workloads. Tests are performed using two functions, and using the same snapshot or different snapshots for each instance.

5.5.6 Bursty workloads

Burst-parallelism is an increasingly common invocation pattern in serverless computing [78]. Real-world events like IoT events and data analytics frameworks can create a large number of parallel invocations in a short time window, and it is important for platforms to be able to efficiently handle such workload patterns. We evaluate two kinds of bursty workloads, the burst of VMs from the same snapshot and from different snapshots: same snapshot represents bursty workloads from the same application while different snapshots represent those from different applications.

We evaluate bursty workloads using Firecracker, REAP, and FaaSnap. For the same snapshot, Firecracker and FaaSnap take advantage of the host OS page cache to avoid redundant disk reads when servicing page faults from the guest VMs. REAP bypasses the page cache to maximize read bandwidth. The FaaSnap loader uses a lock to ensure the loading set is accessed exactly once to avoid redundant accesses.

We measure workloads running 1–64 invocations of the `hello-world` and `json` functions at the same time. Figure 5.10 shows their average execution times and standard deviations. When using the same snapshot, both REAP and FaaSnap outperform Firecracker when parallelism is less than 64. Under higher parallelism, Firecracker benefits from the page cache when multiple guests access the same set of pages. The guests are in effect loading the cache for each other. FaaSnap is faster than REAP in all tests since REAP bypasses the page cache, missing the caching opportunity. When parallelism reaches 64, the CPU becomes the bottleneck and all settings take longer to execute and have higher variance.

When using different snapshots, Firecracker performance degrades quickly because the disk overheads from on-demand reading of all of the different snapshots increase quickly. Both REAP and FaaSnap run much faster than Firecracker with more efficient disk reads. REAP performs similarly to the case when it uses the same snapshot because it does not take advantage of the page cache. FaaSnap outperforms REAP, especially for `json` whose working set has more variance. Overall FaaSnap handles parallel guest page accesses more efficiently than Firecracker and REAP, better supporting bursty workloads.

5.5.7 Remote storage

In disaggregated storage environments, machines do not have local disks and attach remote block storage. To evaluate performance in such cases, we measure the invocation time while snapshots and related files are stored on remote block storage. We use an AWS Elastic Block Store (EBS) `io2` volume with 64K maximum IOPS and 1 GB/s maximum throughput. We measure the execution time of all the functions in Table 5.2 using Firecracker, REAP, and FaaSnap snapshots. We conduct the test three times and report the mean and standard deviations of execution time.

Figure 5.11 shows the results. Baseline Firecracker snapshot performance using remote EBS is on average 33% slower than using the local NVMe SSD, reflecting the effects of increased latency and lower bandwidth from remote disks. Both REAP and FaaSnap significantly outperform Firecracker for most functions. FaaSnap is faster than REAP in most functions,

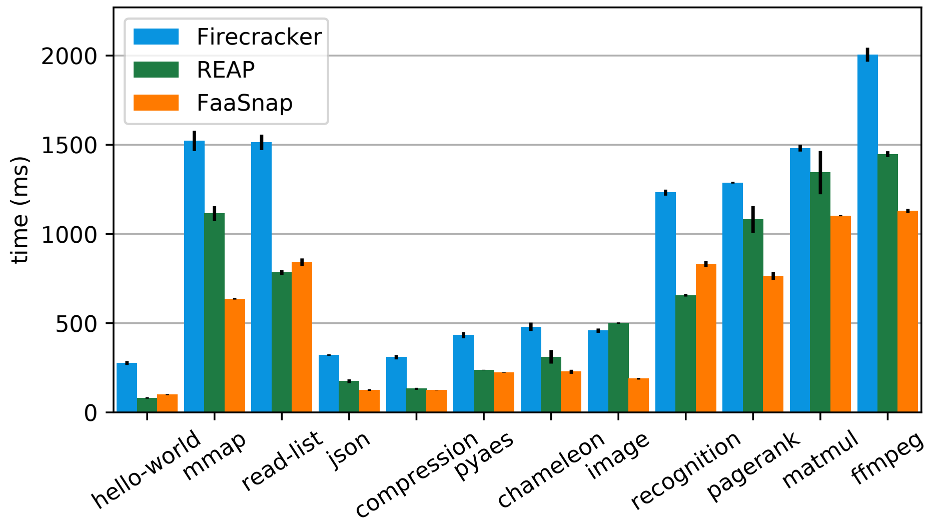


Figure 5.11. Performance using remote storage for snapshots and related files.

except `recognition`, `read-list`, and `hello-world`. In these functions, the working set is very stable, making REAP's blocking working set fetching more efficient. On average, while FaaSnap performance using EBS is 28% slower than using a local NVMe SSD, it is 2.06x faster than Firecracker and 1.20x faster than REAP.

The feasibility of FaaSnap on remote storage enables it to be deployed on any machine, not just those with high-speed local SSDs, thereby extending its deployment flexibility.

5.6 Discussion

5.6.1 Warm starts vs. snapshots vs. cold starts

A FaaS function invocation can be served in a warm VM, if a warm environment exists, using a snapshot, if there are existing snapshots, or by booting a cold VM if neither exists. The Azure Function traces [78] show that less than half of the functions are invoked every hour, and less than 10% are invoked every minute. For the most frequent functions, keeping warm VMs alive and using warm starts is the best choice. Snapshots are useful for less frequently executed functions where keeping warm VMs has more overhead than benefit. Snapshots are also useful for applications with large variance in workloads, such as applications with sudden bursts of workloads. In this case, keeping warm VMs reduces resource utilization while cold starts have high latency. As shown in Section 5.5.6, FaaSnap can serve bursts of function invocations efficiently. For very cold functions that are rarely invoked, snapshots are likely not worth the storage and management costs. Therefore, snapshots can be used to replace cold starts for functions invoked less frequently than those that benefit from warm VMs, and replace warm VMs when their utilization is low (e.g., on eviction).

5.6.2 Storage costs

While snapshots can improve performance, they do incur a real cost for cloud providers for storing and managing the snapshot files. Two factors that determine the storage cost are snapshot file sizes and storage location.

In general, the sizes of snapshot memory files are the same as the guest memory size since the snapshot is a full copy of the guest memory. These sizes are typically a few hundred MB to a few GB, which are comparable to function image sizes. In practice, since guest memory often contains zero pages, snapshot files can be saved as sparse files to reduce their sizes. In effect, though, snapshots increase the storage requirements for functions that use them. As a result, for very infrequent functions, providers can choose to not take snapshots at all to reduce overall storage requirements.

Snapshot files can be stored on local SSDs, remote block storage like EBS, or remote object storage like AWS S3. Storing snapshots on local SSDs provides the best performance, but it is a relatively limited and expensive resource. Remote storage for snapshots is cheaper and much more plentiful, but has higher latency for serving snapshots.

As a result, deploying snapshots represents a tradeoff for providers. While most of our experiments, as well as those performed by other snapshot optimization systems like REAP [88] and Catalyzer [26], measure performance using local SSDs, such results represent an upper bound in performance and should be interpreted in that light. The best case is if providers selectively use local SSDs for snapshot storage for functions invoked frequently, but not frequently enough to be served from warm VMs cached in memory (e.g., warm VMs can be evicted from memory via snapshot to local disk).

Though using network storage does introduce additional latency, it is still a viable alternative. Section 5.5.7 shows that when using EBS, FaaSnap snapshots still provide performance benefits. Snapshots for functions further down the invocation frequency distribution can be stored in the slowest tier object storage such as S3. Providers can also access snapshots in a hierarchical caching scheme. FaaSnap provides an even more fine-grained option for them since it divides guest memory into memory sets and loading set groups. In the future we plan to explore storing relatively small loading set files on local SSD and larger memory files on remote storage to reduce storage costs while satisfying the performance requirements of reading loading sets.

5.6.3 Memory footprints

When the working set estimate is a good match for subsequent invocations, such as the experiments in Section 5.5.2, the memory footprints of FaaSnap are similar to that of Firecracker snapshots. In those experiments, on average it consumes 6% more memory than Firecracker (anonymous and page cache combined), although not always (FaaSnap consumes less memory than Firecracker in 3 of the 12 functions). Prefetching the working set into the page cache does not significantly increase the memory footprint because the working set is likely going to be loaded on-demand in Firecracker snapshots. When the working set estimate is largely inaccurate and large portions of the loaded working set are not used by the guest VM, the memory footprint can increase for working set optimizations like REAP and FaaSnap.

5.6.4 Snapshot security

Reusing a snapshot for VMs has two security concerns. The first is the inherent risks when reusing an environment including in-memory state that persists across invocations. This situation for snapshots is similar to using warm VMs, and is considered acceptable in FaaS. The second is unique to restoring multiple VMs from the same snapshot. The restored instances will have the same initial states. Specifically, pseudo-random number generators (PRNGs) with identical states can lead to a security vulnerability for cryptography. Several solutions have been proposed including using a new `madvise` flag to wipe memory locations with high-value secrets when taking a snapshot [16], and using a special device to provide unique VM IDs to the restored VMs [62].

5.7 Conclusion

On-disk snapshots are a promising way to prevent the overhead of cold starts in serverless computing. Existing snapshot and restore approaches have inefficiencies like long initial blocking, sensitivity to working set changes, and a host-guest memory semantic gap. In this chapter I propose FaaSnap, which develops several techniques and optimizations such as concurrent paging and per-region mapping to reduce the costs of guest VM major page faults. Experimental results show that FaaSnap improves function execution by up to 3.5x than the state-of-the-art, and it is only 3.5% slower than snapshots cached in memory. FaaSnap handles bursty workloads well, which allows it to better support GPSL-style applications.

Efficient cold start using VM snapshots not only improves serverless application latency, but also reduces resource consumption for the platform, which eventually drives down the operational costs for users.

5.8 Acknowledgement

Chapter 5, in full, is a reprint of the material as it appears in Proceedings of the 17th European Conference on Computer Systems. “FaaSnap: FaaS Made Fast Using Snapshot-based VMs.” Lixiang Ao, George Porter, and Geoffrey M. Voelker. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion

This thesis is inspired by the emergence of serverless computing, a new cloud computing model. Initially proposed for cloud-based request handlers, serverless uses a high-level interface to simplify building applications, adopts a pay-as-you-go billing model to reduce users' operational costs, and enables fine-grained autoscaling based on current workloads.

I introduce the term general purpose serverless (GPSL) to describe a new trend in serverless applications. GPSL extends the request handler model of serverless computing to more complex applications that involve multiple serverless instances and cross-instance communication and coordination. Under the GPSL model, more applications like video encoding, parallel data analytics, and distributed compiling are possible.

In this thesis I developed Sprocket, an application framework tailored for video processing pipelines on serverless. By matching the massive parallel workers of serverless and internal parallelism of video content, Sprocket can process large amounts of data with low latency. As an application framework, Sprocket allows users to easily construct complex video processing pipelines using high-level modular building blocks. Sprocket handles underlying complexities like managing serverless resources, input/output, data dependencies, and mitigating stragglers.

Sprocket shows application framework level features benefit GPSL applications. However, some challenges cannot be solved on the application framework level. One challenge is the inefficiency of data transfers between serverless instances, which is a result of a lack of direct network support. Another challenge is the delays due to scheduling and cold start latencies, which are determined by the platform's scheduling policies and runtime design.

I developed Particle to enable direct network support among serverless instances. Due to the ephemeral nature of serverless, traditional virtual networks impose significant overheads when a burst of new serverless instances is created. Particle uses a lightweight network stack that avoids costly operations involving network namespaces and virtual devices. As a result, Particle helps serverless platforms better support the networking requirements of GPSL applications.

I developed Sophon to tackle the scheduling inefficiencies in serverless platforms. OpenWhisk scheduling suffers from problems like container thrashing, which causes unnecessary cold starts, low system throughput and increased latency. To address these problems, Sophon adopts container-grained scheduling. By maintaining the resource state of containers, Sophon avoids the container thrashing problem. Experimental results show Sophon increases system throughput, which helps reduce operational costs. It also reduces the number of cold starts and reduces serverless request latency, which is important for GPSL applications since the slow requests can become performance bottlenecks.

Finally I developed FaaSnap to improves the performance of cold starts. Although efficient scheduling policies, as demonstrated by Sophon, can prevent many unnecessary cold starts, in other cases cold starts are unavoidable. FaaSnap adopts snapshot-based virtual machines for serverless environments. It uses techniques including concurrent paging, per-region mapping, and host page recording to reduce the latency of cold starts. Similar to Sophon, reducing the request latency prevents GPSL performance bottlenecks. Moreover, FaaSnap can efficiently handle bursty workloads, a common pattern in GPSL applications such as Sprocket.

In this thesis I demonstrate systems, including Sprocket, Particle, Sophon, and FaaSnap, provide application framework level and platform level features, which greatly simplify implementation, reduce network overheads, and improve scheduling and cold start performance of GPSL computing. These systems expand the potential of future GPSL computing, paving the way for the adoption of serverless to a wider range of applications.

Bibliography

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, (NSDI'20), Santa Clara, CA, February 2020. USENIX Association.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference*, (USENIX ATC'18), pages 923–935, Boston, MA, USA, July 2018. USENIX Association.

[3] Amazon Web Services. Amazon Elastic Container Service. https://aws.amazon.com/ecs/, 2020.

[4] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proceedings of the Sixth European Conference on Computer Systems (EuroSys)*, pages 287–300, Salzburg, Austria, April 2011. ACM.

[5] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 185–198, Lombard, IL, April 2013. USENIX Association.

[6] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 265–278, Vancouver, BC, Canada, October 2010. USENIX Association.

[7] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing*, (SoCC'18), pages 263–274, Carlsbad, CA, October 2018. ACM.

[8] ARM. Quagga Routing Suite. https://www.nongnu.org/quagga/, 2018.

[9] Avengers Trailer. https://www.youtube.com/watch?v=eMobkagZu64.

[10] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 261–272, Dallas, TX, May 2000.

[11] AWS. AWS Lambda – Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/, 2021.

[12] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, pages 1151–1162, Hanover, Germany, April 2011.

[13] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[14] Calico. Calico. https://www.tigera.io/project-calico/, 2022.

[15] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[16] Adrian Costin Catangiu. Introduce MADV_WIPEONSUSPEND. https://lwn.net/Articles/825230/, 2020.

[17] Cisco Visual Networking Index: Forecast and Methodology, 2016–2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html.

[18] Colbert Interview. https://www.youtube.com/watch?v=Y6XXMGUb5kU.

[19] containerd. An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/, 2021.

[20] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21. Association for Computing Machinery, 2021.

[21] CoreOS. Flannel. https://coreos.com/flannel, 2019.

[22] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, (NSDI'19), pages 177–191, Boston, MA, USA, February 2019. USENIX Association.

[23] MPEG Dash Industry Forum. http://dashif.org/.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, pages 137–149, San Francisco, CA, December 2004. USENIX Association.

[25] Docker. Docker. https://www.docker.com/, 2022.

[26] 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.

[27] 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.

[28] Earth. https://www.youtube.com/watch?v=wnhvanMdx4s.

[29] Firecracker. Firecracker Snapshotting. https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md/, 2021.

[30] 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 *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC'19, pages 475–488, July 2019.

[31] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthike yan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Siva raman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th ACM/USENIX Symposium on Networked System s Design and Implementation (NSDI)*, Boston, MA, USA, March 2017. USENIX Association.

[32] 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*, pages 386–400, 2021.

[33] Google Cloud Functions. https://cloud.google.com/functions/.

[34] Google Cloud Vision API. https://cloud.google.com/vision/.

[35] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open MPI: A Flexible High Performance MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, PPAM'05, pages 228–239. Springer, 2005.

[36] gVisor. gVisor. https://gvisor.dev/, 2021.

[37] Apache Hadoop. http://hadoop.apache.org/.

[38] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *Proceedings of the 8th Workshop on Hot Topics in Cloud Computing (HotCloud)*, Denver, CO, USA, June 2016. USENIX Association.

[39] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 295–308, Boston, MA, March 2011. USENIX Association.

[40] Qi Huang, Petchean Ang, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito IV, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. SVE: Distributed Video Processing at Facebook Scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017. ACM.

[41] iproute2. Iproute2 routing commands. https://git.kernel.org/pub/scm/network/iproute2/iproute2.git, 2019.

[42] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, 2007. ACM.

[43] Jake Edge. A seccomp overview. https://lwn.net/Articles/656307/, 2015.

[44] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 445–451, Santa Clara, CA, September 2017. ACM.

[45] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.

[46] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Centralized Core-granular Scheduling for Serverless Functions. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, (SoCC'19), pages 158–164, Santa Cruz, CA, USA, November 2019. ACM.

[47] Apache Kafka. https://kafka.apache.org/.

[48] Kata Containers. Kata Containers. https://katacontainers.io/, 2020.

[49] 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.

[50] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 18, pages 427–444, 2018.

[51] Knative. Kubernetes-based platform to deploy and manage modern serverless workloads. https://knative.dev/, 2020.

[52] Kubeless. The Kubernetes Native Serverless Framework. https://kubeless.io/, 2020.

[53] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 75–86, Indianapolis, Indiana, June 2010. ACM.

[54] Yongchul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A Study of Skew in MapReduce Applications. The 5th Open Cirrus Summit, 2011.

[55] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewTune: Mitigating Skew in Mapreduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 25–36, Scottsdale, Arizona, May 2012. ACM.

[56] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, 2009.

[57] Ian Lewis. The Almighty Pause Container. https://www.ianlewis.org/en/almighty-pause-container, October 2017.

[58] Linux Containers. Container and virtualization tools. https://linuxcontainers.org/, 2021.

[59] 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. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[60] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, (SOSP'17), pages 218–233, Shanghai, China, November 2017. ACM.

[61] Matt Fleming. A thorough introduction to eBPF. https://lwn.net/Articles/740157/, 2017.

[62] Microsoft. Virtual Machine Generation ID. http://go.microsoft.com/fwlink/?LinkId=260709, 2012.

[63] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud'19, 2019.

[64] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing*, (HotCloud'19), Renton, WA, USA, July 2019. USENIX Association.

[65] Microsoft Computer Vision and Cognitive Services API. https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/.

[66] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 439–455, New York, NY, USA, 2013. ACM.

[67] Nature. https://www.youtube.com/watch?v=eMobkagZu64.

[68] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference*, (USENIX ATC'18), pages 57–70, Boston, MA, USA, July 2018. USENIX Association.

[69] Official Kubernetes. Pod Overview. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, 2020.

[70] OpenFaaS. Serverless Functions Made Simple. https://www.openfaas.com/, 2020.

[71] Apache OpenWhisk. Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/, 2021.

[72] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, (SOSP'13), pages 69–84. ACM, November 2013.

[73] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 19, pages 193–206, 2019.

[74] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A Balanced Large-scale Sorting System. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 29–42, Berkeley, CA, USA, 2011. USENIX Association.

[75] Muhammad Raza and Chrissy Kidd. The Cloud in 2022: Growth, Trends, Market Share & Outlook. https://www.bmc.com/blogs/cloud-growth-trends/, 2021.

[76] AWS Rekognition. https://aws.amazon.com/rekognition/.

[77] Alastair Robertson. bpftrace. https://github.com/iovisor/bpftrace, 2021.

[78] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference*, (USENIX ATC'20), pages 205–218. USENIX Association, July 2020.

[79] 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.

[80] Sintel. https://www.youtube.com/watch?v=qR5vOXbZsI4.

[81] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*, pages 275–287, 2007.

[82] Apache Spark. http://spark.apache.org/.

[83] AWS Step Functions. https://aws.amazon.com/step-functions/.

[84] Tears of Steel. https://www.youtube.com/watch?v=OHOpb2fS-cM.

[85] Apache Tez. https://tez.apache.org.

[86] Shelby Thomas, Lixiang Ao, Geoffrey M Voelker, and George Porter. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 16–29, 2020.

[87] Tim Wagner. Serverless Networking is the next step in the evolution of serverless. https://bit.ly/30kFoY9, 2019.

[88] 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.

[89] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 148–162, 2005.

[90] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference*, (USENIX ATC'18), pages 133–145, Boston, MA, USA, July 2018. USENIX Association.

[91] Simple, resilient multi-host containers networking and more. https://github.com/weaveworks/weave.

[92] Apache Yarn. https://hortonworks.com/apache/yarn/.

[93] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–42, San Diego, CA, December 2008. USENIX Association.

[94] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 377–392, Boston, MA, March 2017. USENIX Association.

[95] Irene Zhang, Tyler Denniston, Yury Baskakov, and Alex Garthwaite. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 1–12, 2013.

[96] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C Barr. Fast Restore of Checkpointed Memory using Working Set Estimation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 87–98, 2011.

[97] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery.