Hermod: Principled and Practical Scheduling for Serverless Functions

Kostis Kaffes
Stanford University
Stanford, CA, USA
kkaffes@stanford.edu

Neeraja J. Yadwadkar
University of Texas at Austin
Austin, TX, USA
neeraja@austin.utexas.edu

Christos Kozyrakis
Stanford University
Stanford, CA, USA
kozyraki@stanford.edu

ABSTRACT

Serverless computing has seen rapid growth due to the ease-of-use and cost-efficiency it provides. However, function scheduling, a critical component of serverless systems, has been overlooked. In this paper, we take a first-principles approach toward designing a scheduler that caters to the unique characteristics of serverless functions as seen in real-world deployments. We first create a taxonomy of scheduling policies along three dimensions. Next, we use simulation to explore the scheduling policy space and show that frequently used features such as late binding and random load balancing are sub-optimal for common execution time distributions and load ranges. We use these insights to design Hermod, a scheduler for serverless functions with two key characteristics. First, to avoid head-of-line blocking due to high function execution time variability, Hermod uses a combination of early binding and processor sharing for scheduling at individual worker machines. Second, Hermod is cost, load, and locality-aware. It improves consolidation at low load, it employs least-loaded balancing at high load to retain high performance, and it reduces the number of cold starts compared to pure load-based policies. We implement Hermod for Apache OpenWhisk and demonstrate that, for the case of the function patterns observed in real-world traces, it achieves up to 85% lower function slowdown and 60% higher throughput compared to existing production and state-of-the-art research schedulers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '22, November 7–11, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9414-7/22/11…$15.00
https://doi.org/10.1145/3542929.3563468

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

cloud computing, serverless, scheduling

ACM Reference Format:
Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: Principled and Practical Scheduling for Serverless Functions. In *Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3542929.3563468

1 INTRODUCTION

Function-as-a-Service computing (FaaS) is becoming increasingly popular, primarily due to its ease of use [13]. Users just need to write functions in a high-level language, specify events and endpoints as execution triggers, and pay only for the resources used during function execution at fine (sub-second) granularity. These functions are commonly called *serverless functions* since infrastructure tasks such as resource provisioning, scheduling, scaling, and security are handled by specialized computing platforms [46] without burdening the users. These platforms exist both as part of cloud providers' offerings (AWS Lambda [4], Google Cloud Functions [10], Azure Functions [9]) and as open-source frameworks that can be deployed anywhere [5, 8, 11, 38, 77].

Recent research has optimized various aspects of serverless platforms like the function startup-latency [17, 23, 31, 64, 74], memory footprint [17, 37, 64], and inter-function communication [18, 48, 62, 78]. In this paper, we focus on *scheduling serverless functions across and within machines*, an optimization critical to FaaS performance. Our analysis is guided by a real-world production trace of Azure Functions [74]. These real-world serverless functions are characterized by highly-variable execution times, burstiness, and skewed invocations. We take a principled approach and create *a taxonomy of scheduling policies* that encompasses a broad set of techniques drawn from prior work on cluster and task scheduling and existing serverless frameworks.

Using simulation, we explore the policy space and conclude that commonly used techniques are sub-optimal for

the characteristics of serverless functions: First, we show that *even idealized Late Binding is sub-optimal for highly-variable workloads as short invocations can get stuck in the scheduler queue, waiting for longer ones to finish execution.* We can eliminate this problem if all invocations make forward progress using a processor-sharing policy. To do so, we need to schedule invocations for execution as soon as they enter the system, i.e., use early binding. This conclusion goes against the conventional wisdom that late binding should always be preferred [67, 85] or approximated [54, 55]. Second, *we conclude that both random and locality-based load balancing across servers are ineffective at high load; least-loaded balancing is necessary to avoid performance loss due to load imbalance.* Third, we identify the practical issues when using least-loaded balancing in real-world systems like Apache OpenWhisk [5]. At low loads, the spreading of function invocations leads to the usage of more servers than necessary and more function cold starts. Hence, it achieves low hardware efficiency and increases function execution time.

Based on these findings, *we develop Hermod, a locality-aware hybrid scheduler that consolidates function invocations at a low load while reverting to least-loaded balancing at high load.* By consolidating invocations only at low load, Hermod achieves higher efficiency without sacrificing performance. By reverting to least-loaded balancing for higher loads, it reduces queuing in overload situations. Hermod is cost, load, and locality aware; it takes into account locality when making scheduling decisions without sacrificing resource efficiency or causing load imbalance. Hermod also incorporates the other lessons from our analysis by *combining early binding with processor-sharing* to avoid head-of-line blocking.

We implement Hermod for Apache OpenWhisk [5], a popular open-source serverless platform. We evaluate Hermod's performance on workloads modeled after Azure and Twitter production traces as well as workloads with characteristics similar to emerging use cases of the serverless paradigm such as interactive analytics. We show that Hermod achieves 85% lower slowdown at low load - both at the median and the tail - than the currently-used OpenWhisk policy and supports 60% higher load than commonly-used late binding approaches [67, 85] for the Azure trace-based workload. Being locality-aware, it achieves up to 50% lower slowdown than an idealized least-loaded policy that improves upon state-of-the-art request schedulers [54] while using up to 60% fewer servers. Additional experiments demonstrate that Hermod is robust to workloads with different function mixes and execution time distributions.

The key contributions of this paper are the following:

- We introduce a taxonomy that decomposes scheduling approaches for serverless functions into three fundamental decisions and enables systematic research.
- Using the characteristics of serverless functions as they appear in a real-world trace, we show that commonly used scheduling techniques, such as late binding and random load balancing, are sub-optimal for serverless workloads.
- We use these conclusions to design Hermod, a load and locality-aware scheduler that combines early binding, hybrid load balancing, and processor sharing at individual workers.
- We implement Hermod for Apache OpenWhisk [5], the leading open-source serverless platform, and demonstrate performance and efficiency improvements over existing scheduling approaches.

2 BACKGROUND

2.1 Life-cycle of a Serverless Function

Figure 1 describes the life-cycle of a serverless function invocation in a generic serverless platform.

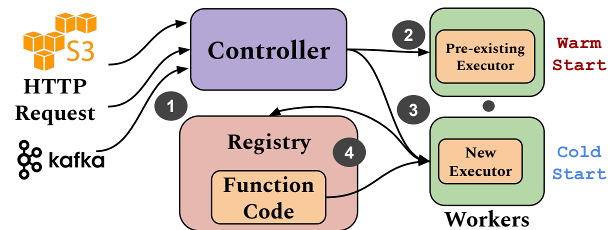


Figure 1: Generic Serverless Platform Architecture

- Functions can be triggered in various ways, such as through HTTP requests, writes to a message queue, timer events, or uploads to cloud storage. Each function invocation is received by a Controller ❶ that terminates SSL connections and enforces access control and rate limiting.
- The Controller then schedules invocations to Workers for execution. One option is to forward the invocation to a Worker with a "warm" executor (VM [17], container [5], or lightweight process [76]) running the function's code ❷. In this case, little initialization is needed and there is low overhead; this is a "warm start" of the serverless function.
- It is possible that there is no pre-existing executor for the invoked function or that all such executors are full. In that case, the Controller forwards the invocation to a Worker with available compute and memory capacity ❸.
- When a Worker with no pre-existing executors receives an invocation, it fetches the function code from a registry, and starts a new executor ❹ incurring additional overhead. This is called a "cold start" of a function.

There can be many variations of this architecture. In some platforms, there is a separate load balancing module that manages the scheduling of invocations to Workers. It is also possible that some data store is used to persistently store critical information, e.g., access control lists. Persistent messages

queues can be used instead of simple network protocols such as HTTP for Controller-Worker communication.

Regardless of the exact architecture, scheduling decisions are taken in two components, the *Controller* and the *Workers*. The Controller decides how invocations are scheduled to Workers while individual Workers decide how their resources are distributed among the invocations they execute.

2.2 Characteristics of Serverless Functions

Microsoft recently released a production trace of its serverless offering [74]. The trace includes data about 445M function invocations across Microsoft Azure's entire infrastructure over a 14-day period. We use this trace along with information published by other cloud providers [13] to infer the following characteristics of serverless functions:

Short and Highly-variable Execution Times: Functions typically live in the range of hundreds of milliseconds to minutes, and are billed at sub-second granularity. The Azure trace showed that function execution times can be modeled using a heavy-tailed Log-normal distribution indicating extreme variability. The median execution time is only 600 milliseconds while the 99% execution time is more than 140 seconds. Similar observations were made in AWS Lambda [13].

Skewed Function Popularity: Just 0.6% of the functions account for 90% of the total invocations in the Azure trace.

Burstiness: Arrivals rates of individual functions are bursty with an average burstiness index of -0.26, close to that of a Poisson process [51]. However, the total number of invocations across Microsoft's data centers does not vary much over time, following diurnal patterns that are characteristic of large-scale cloud systems [25, 74, 79].

2.3 Mismatch between Characteristics and Scheduling of Serverless Functions

Most of the prior research on serverless platforms has focused on mechanisms that reduce the function cold-start cost and frequency. Specialized virtual machines [17], lightweight and secure operating systems and language abstractions [76], snapshots [80], and check-pointing methods [23, 31] have been able to reduce the start-up cost to less than 10 milliseconds while various techniques have been used to cache function data [34, 61, 71] or predict future invocations [74] and avoid cold starts altogether.

Much less focus has been given to scheduling policies for serverless functions. Scheduling has proven to be a critical factor to the performance of cloud systems [29, 30, 60, 66, 67, 85]. Serverless platforms can also be considered as Remote Procedure Call (RPC) systems; better scheduling has provided up to an order of magnitude performance improvement for such systems [47, 54, 68]. Nevertheless, existing schedulers use techniques and policies that are ill-suited for the characteristics of serverless functions:

Existing Schedulers for Serverless functions: Apache OpenWhisk [5] is one of the most widely used open-source serverless platforms. However, its default scheduler employs pure locality-based load balancing, co-locating invocations of the same function to a randomly-selected Worker without taking account of the load, a technique shown to be unable to handle *highly-skewed workloads* [44, 56]. Other more advanced schedulers, such as Atoll [77] and Orion [57], leverage historic function execution data and user-provided performance constraints to optimize the execution of function DAGs. In this work, we take an opaque-box, provider-centric approach, only considering single-function execution without any function-specific information.

Kubernetes-based Frameworks: Serverless platforms such as OpenFaaS [8], Kubeless [11], and vHive [80] are built on top of Kubernetes and employ a different type of scheduling. They treat serverless functions similarly to classic server workloads and employ auto-scaling. Once the CPU or memory utilization exceeds some pre-defined threshold, more workers are spun up to handle the extra load. If utilization drops below a threshold, the load is consolidated to fewer workers. Such coarse-grain and reactive policies cannot handle the *burstiness* associated with serverless workloads, and lead to high tail latency and slowdown [65, 69]. That is why modern serverless platforms such as Fission [12] and Knative [14] that run on top of Kubernetes have abandoned utilization-based auto-scalers, using other metrics such as requests in flight [16] or custom user-defined metrics [15] instead. However, these platforms employ very simple load-balancing approaches, e.g., round-robin, that are not cost- or load-aware.

General Task Schedulers: Tasks scheduled by scientific computing and analytics frameworks have common characteristics with serverless functions, i.e., short execution times and burstiness. Some of these frameworks focus on improving data locality by queuing tasks or migrating them between machines so that they execute close to their corresponding data [60, 70]. However, the overhead associated with cold-starts [84] makes migrations unappealing for serverless functions. Task schedulers that avoid task migrations or do not place hard locality constraints could prove more suitable. Sparrow [67] and Pigeon [85] - two popular schedulers for data analytics frameworks - employ late binding, i.e., task-to-worker assignments are delayed until workers are ready to run the task. However, late binding is sub-optimal for workloads with *highly-variable* execution times: shorter tasks can suffer slowdown due to getting stuck behind tasks with much longer execution times. Interleaving of shorter and longer tasks is desirable for such workloads to avoid this phenomenon, known as head-of-line blocking [47, 59]. Hawk [28] and Eagle [26] improve upon Sparrow by differentiating between

short and long tasks. A set of nodes is set aside for short tasks to avoid head-of-line blocking. Long tasks are centrally scheduled while short tasks use distributed schedulers and the probing techniques employed by Sparrow. These systems require knowledge of each task's execution time and work-stealing to compensate for low-quality scheduling decisions made by distributed schedulers. These requirements make such schedulers incompatible with serverless functions due to their irregular execution times and the high cold-start cost.

Cluster Schedulers: There is a large body of work on cluster schedulers that can manage thousands of jobs, belonging to different users and applications. Some of these schedulers (Borg [82], Yarn [81], Mesos [39], Mercury [49]) allow different frameworks to request and receive cluster resources, leaving the allocation of resources to tasks to the individual framework. Such resource managers can be used by serverless frameworks but they are orthogonal to serverless function scheduling. The serverless framework will still need to decide when to request resources, i.e., servers or CPU cores, and how to schedule function invocations to these resources.

Other cluster schedulers, despite being more suitable for scheduling individual tasks, focus more on long-running tasks incurring too high overhead to short-running serverless functions. The Quincy [42] and Firmament [36] frameworks model scheduling as a min-cost max-flow (MCMF) optimization problem over a flow network. While these frameworks provide high-quality decisions, they suffer from long scheduling delays and often require job migrations exacerbating the cold-start problem of serverless functions. Stratus [24] tightly packs tasks to machines to improve efficiency and reduce costs. We use a similar approach in one of the two modes of operation of Hermod. Medea [35] optimizes the scheduling of long-running applications, an important objective which is however irrelevant to short serverless function invocations.

3 TAXONOMY OF SCHEDULING POLICIES

3.1 Scheduling Analysis

The first step to designing a scheduler for serverless functions is to identify the scheduling decisions made during the life-cycle of a function invocation (§ 2.1):

(1) When should an invocation be scheduled to a Worker?
(2) Which Worker should handle each invocation?
(3) Which intra-Worker scheduling policy should be used?

The first two decisions are enforced at the *Controller* while the third is enforced at individual *Workers*.

We now introduce a taxonomy that describes the policies resulting from different answers to the aforementioned scheduling questions. We use this taxonomy to formalize our analysis in the following sections and identify the policies that cater to the characteristics of serverless functions.

When should an invocation be scheduled to a Worker? There are two major options, early and late binding. In early binding scheduling, no queuing takes place at the Controller, i.e., function invocations are assigned to Workers for execution as soon as they reach the Controller. Early binding is commonly used when queuing at the Controller would consume too much memory to store the functions' arguments, such as in OpenWhisk [5], when the Controller-Worker communication latency is high and needs to be hidden (Canary [70], R2P2 [54], Mind the Gap [40]), or in some distributed settings where schedulers do not have a global view of the system and must choose the best available among a limited number of nodes (Hawk [28], Eagle [26]). If late binding is used, invocations are queued at the Controller until there are available resources to guarantee their immediate execution. Late binding is assumed to provide better performance than early binding since it completely avoids load imbalances at Workers and thus is preferred in systems that can tolerate the overheads associated with it (Firmament [36], Quincy [42], Medea [35], Sparrow [67], Pigeon [85]).

Where should a function invocation execute? Regardless of whether early or late binding is used, we need to determine a load balancing policy that selects a Worker to run a function on. This is the decision most cluster schedulers (Quincy [42], Firmament [36], Medea [35], Stratus [24]) make. One approach - used by OpenWhisk [5] - is to optimize for locality, packing invocations of the same function in as few Workers as possible to minimize the number of cold starts. Another approach that provides good load balancing in expectation with low overhead is random assignment [21]. Finally, there is a group of load-aware policies, e.g., always selecting the least-loaded Worker. Both large-scale task schedulers (Hawk [28], Eagle [26], Kairos [27]) and small-scale microsecond-level request schedulers (RackSched [89], R2P2 [54]) have used or approximated least-loaded balancing.

Which intra-Worker scheduling policy should be used? If early binding is used, there is queuing at the Workers and a scheduling policy must control the way each Worker's resources are used by different invocations. Different policies are optimal for different types of workloads. The Shortest-Remaining-Processing-Time (SRPT) scheduling policy has long been known to be optimal for minimizing mean response time and is commonly used in environments where information about the task execution time is available (Eagle [26]). It is hard to use SRPT in most setups - including serverless function scheduling - because the processing time is rarely known a priori. Hence, some schedulers, e.g., Kairos [27], attempt to approximate it by using the Least

Attained Service scheduling policy [63]. A policy with similar characteristics is Processor Sharing (PS) where each task receives an equal share of the processor's capacity. Linux's Completely Fair Scheduler (CFS) [2] is an approximation of PS that is applicable to real systems. Since most systems that employ early binding delegate scheduling to the operating system, they end up using CFS - and thus approximate PS - for intra-Worker scheduling. Another practical policy is First-Come-First-Serve (FCFS) which has been used by both task schedulers such as Hawk [28] and low-latency systems (R2P2 [54], HovercRaft [53], IX [21]) due to its simplicity and low overhead.

Notation: We use notation similar to Kendall's from standard queuing theory [50] to describe the different scheduling policies that can be created by combining the aforementioned options. We describe a policy through the use of 3 parameters, $T/LB/S$, where:

- **T** describes whether early or late binding is used. The two possible values for T are E and L. E denotes early binding and L denotes late binding.
- **LB** describes the load balancing policy used to select an available Worker. Three possible values for LB are LOC (locality-based balancing), LL (least-loaded balancing) and R (random balancing).
- **S** describes the scheduling policy used in the Workers. The two practical such policies that we use are Processor-Sharing (PS) and First-Come-First-Serve (FCFS). We discuss policies that require knowledge of the request execution time in § 3.4.

For example, with this notation we can describe the OpenWhisk scheduler that uses early binding (E), locality-based load balancing (LOC), and processor sharing at the Worker servers (PS) as $E/LOC/PS$.

The mismatch between existing schedulers and the needs of serverless functions as demonstrated in the Microsoft Azure trace (§ 2.3) prompts us to go back to the drawing table. Guided by the taxonomy, we use simulation to explore the scheduling policy space and draw conclusions that help us design an effective policy for serverless function scheduling.

3.2 Simulation Setup

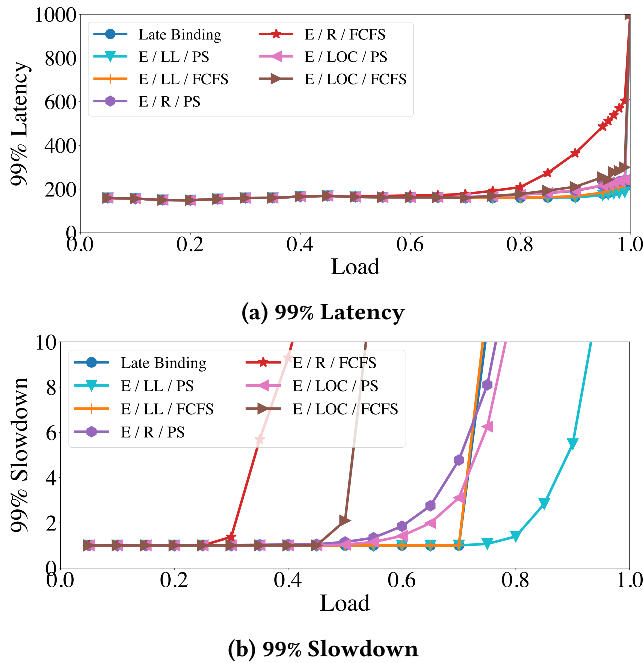
We built a discrete event simulator that allows us to rapidly evaluate and compare different scheduling policies and architectures. Serverless function invocations are generated following tunable distributions for the inter-arrival and the execution times. We focus our analysis on the Azure trace and thus use highly-variable Log-normal execution times and open-loop Poisson arrivals [72, 74, 87]. A Controller schedules incoming activations to Worker servers. Each Worker has a number of cores determining the number of invocations it can execute in parallel and a memory capacity that limits the maximum number of invocations - running and waiting - it can host at any point in time. To mimic our testbed, we set the capacity of each server in terms of invocations to be 8× its number of cores. We implemented different load balancing and intra-Worker scheduling policies. The goal of the simulator is to help us understand the behavior of serverless workloads, not to perfectly mimic the behavior of a real system. To model the trace's skew, invocations belong to 50 distinct functions, one contributing 98% of the load and the rest making up the rest of the load equally. To avoid making arbitrary assumptions, the simulator does not model overheads such as the container start-up time. Thus, the simulator allows us to identify which policy is optimal from a queueing theory perspective. In Section 4, we discuss how the policy design, informed by our simulation results, needs to also take into account these second-order effects.

3.3 Pruning the Policy Space

We start our analysis from a simple example where we have **four Worker servers with 12 cores each**. We compare all 12 possible policies we defined in Section 3.1:

- $L/*/*$ **(Late Binding):** Using late binding scheduling, function invocations are scheduled to a Worker in an FCFS manner only when resources are available. If no resources (cores) are available, invocations are queued at the Controller. The simulator does not capture second-order effects such as interference. Hence, the the load balancing and the intra-Worker scheduling policy do not matter when late binding is used; if a function invocation is scheduled to a Worker, our simulator assumes that it will run uninterruptedly. This policy is used by popular task scheduling systems such as Pigeon [85] and Sparrow [67].
- $E/LL/FCFS$ **(Early Binding / Least-Loaded Balancing / FCFS):** The Controller implements a Join-Shortest-Queue policy where invocations are queued locally at each Worker where they execute FCFS. This policy is used by some RPC systems such as R2P2 [54].
- $E/LL/PS$ **(Early Binding / Least-Loaded Balancing / PS):** Similar to the previous policy, but the queued invocations timeshare each Worker.
- $E/LOC/FCFS$ **(Early Binding / Locality-Based Balancing / FCFS):** The Controller forwards all invocations belonging to the same function to a randomly-selected Worker where they execute FCFS. If the Worker's capacity is reached, we select a different random Worker.
- $E/LOC/PS$ **(Early Binding / Locality-Based Balancing / PS):** Similar to the previous policy, but the queued invocations timeshare each server.
- $E/R/FCFS$ **(Early Binding / Random Load Balancing / FCFS):** The Controller randomly assigns invocations to Workers where they execute FCFS.

- $E/R/PS$ **(Early Binding / Random Load Balancing / PS):** Similar to the previous policy, but the queued invocations timeshare each core.



(a) 99% Latency

(b) 99% Slowdown

Figure 2: Tail latency and slowdown as a function of the load for a synthetic workload with the characteristics of the Azure trace [74], i.e., Log-normal execution times with parameters $\mu = -0.38$ and $\sigma = 2.36$, in a single server.

In Figure 2a we can see the 99% latency for the different policies as a function of the offered load indicated by the fraction of the server's capacity. For a load greater than 1, the system becomes unstable and the queues expand indefinitely. We observe that most policies perform similarly and almost optimally; the tail latency explodes for a load close to 1. Based on these latency results, we get the false impression that scheduling does not matter much.

Next, we study the 99% slowdown for the same setup. Slowdown is defined as:

$$\text{slowdown} = \frac{\text{function latency}}{\text{function execution time}}$$

The closer this ratio is to 1, the smaller the delay a user experiences compared to what she expected based on the function execution time. Slowdown is a metric commonly used in the networking literature when reasoning about the performance of request-based applications [47, 59]. It is more insightful than latency as a performance metric as it can reveal pathological cases of system behavior. For example, in a workload where half of the requests are short, e.g., 100ms, and half are long, e.g., 10sec, the 99% latency - a metric commonly used for performance analysis - is not useful. If the 99% latency is reported to be 10sec, that would give users no information about the performance of their short requests. However, if the 99% slowdown is reported to be 1.5, users would know that most of both the longer and the shorter requests do not experience excessive delays.

We observe that some policies, e.g., Late Binding and $E/LL/FCFS$, perform significantly worse in terms of slowdown, while tail latency had painted a completely different picture. If we used tail latency as the metric, we would assume everything is good and would not analyze further to find out the real problem. Hence, in the rest of this paper we use slowdown as the main metric for our analysis. We make the following observations in Figure 2b:

Observation 1: Processor Sharing in the Workers significantly outperforms FCFS-based scheduling - even Late Binding - regardless of the load-balancing policy used. The optimality of PS over FCFS in terms of tail performance for heavy-tailed workloads has been proven by Boxma and Zwart [22]. Intuitively, functions with light-tailed execution time take about the same time to finish. Therefore, one can minimize the slowdown by running invocations to completion in their arrival order [21]. In the case of heavy-tailed workloads, the slowdown can increase dramatically if short invocations get stuck behind longer ones. PS allows short invocations to receive a slice of the processor and bypass the longer ones. However, PS is incompatible with late binding. Invocations need to be assigned to servers/cores for execution as soon as they enter the system. Otherwise, they are not able to time-share compute resources and suffer from head-of-line blocking.

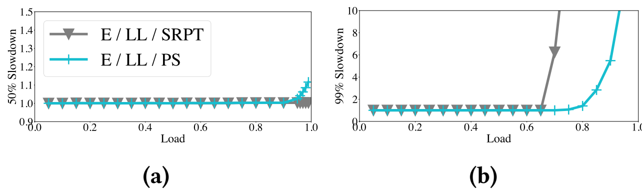
Lesson Learned 1: *Due to head-of-line blocking, Late Binding schedulers and Early Binding schedulers that use FCFS cannot handle the high execution time variability present in serverless workloads.*

Observation 2: The slowdown for both random and locality-based load balancing starts to go up for a load as low as 0.55. Random load balancing results in high latency and slowdown for invocations with highly-variable execution time since it is likely for imbalances across Workers to appear. Similarly, locality-based balancing - as used by Openwhisk - leads to significant load imbalances. Workers serving "hot" functions quickly become overloaded, leading to high slowdown even for a medium load. Least-loaded balancing performs very well keeping the slowdown low for a load as high as 0.8.

Lesson Learned 2: *Random and locality-based balancing, similar to what OpenWhisk is using, is ineffective at high load. Least-loaded balancing provides better performance from a queueing theory perspective.*

3.4 Simple Scheduling can be Better

The two intra-Worker scheduling policies we considered so far, Processor-Sharing and First-Come-First-Serve, do not assume knowledge of the function execution times. Even though this characteristic makes these policies usable in any environment, one could argue that an intelligent scheduler could leverage knowledge from past function invocations to predict future function execution times, similar to what Shahrad et al. [74] did for inter-arrival times. To see if such knowledge benefits a serverless scheduler, we evaluate the performance of a scheduler that assumes perfect knowledge of function execution times and uses the Shortest-Remaining-Processing-Time (SRPT) policy for intra-Worker scheduling. We chose SRPT as it has been shown [20] that it performs better than PS in terms of mean and median slowdown.



(a) **(b)**

Figure 3: Slowdown in a 4-server, 12-core setting under the SRPT and PS scheduling policies.

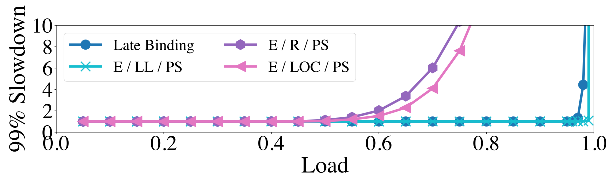


Figure 4: 99% slowdown in a 100-server, 12-core setting.

Observation 3: In Figure 3, we observe that, as expected, the best performing SRPT-based policy ($E/LL/SRPT$) slightly outperforms the best PS-based policy ($E/LL/PS$) in terms of median slowdown at high load. We omit the worse-performing SRPT configurations for brevity. However, we also observe that the SRPT policy performs significantly worse than the PS-based one in terms of *tail* slowdown (99%) which is our metric of interest. The slowdown increases for load as low as 0.7 for the $E/LL/SRPT$ scheduler while for the $E/LL/PS$ scheduler the slowdown is less than 10 for load as high as 0.9. While SRPT is optimal in terms of slowdown on average, its tail performance is worse than that of PS at high load because it can lead to starvation for longer requests. We expect SRPT's performance to be worse in a realistic environment where perfect knowledge of execution times is unavailable and scheduling decisions depend on estimates.

Lesson Learned 3: We demonstrated that, counterintuitively, using perfect knowledge of the execution time of each individual function invocation through SRPT, a well-studied policy, does not offer performance benefits over PS for *tail* slowdown. Hence, in the rest of the paper we consider only execution-time-agnostic policies which are more generally applicable.

3.5 Scheduling at Scale

As Figure 4 shows, our main conclusions are still valid for a large-scale, 100-server, 1200-core setup. $E/R/PS$'s and $E/LOC/PS$'s slowdown explodes at a relatively low 0.6 load. Late Binding performs much better than in smaller setups because occurrences of head-of-line blocking reduce as the number of Workers increases. However, we see that $E/LL/PS$ still outperforms Late Binding at very high loads (>0.96). That, together with the inherent complexities and overheads associated with Late Binding, convinces us that $E/LL/PS$ is the best policy to use regardless of the scale of the setup.

4 HERMOD DESIGN

Based on the conclusions of the simulations in § 3, we developed Hermod, a scheduler that is both *locality and load-aware*. Hermod caters to the unique characteristics of serverless functions by combining *early binding* and *processor sharing* with *hybrid load balancing* that adapts to changes in the load. Moreover, it keeps track of warm available containers for each function, avoiding cold-starts when possible.

4.1 Why Hybrid Load Balancing is Necessary

Based on the simulations, the ideal scheduling policy for a serverless workload combines **early binding** with **least-loaded** balancing and **processor-sharing** scheduling at the Workers ($E/LL/PS$). However, an $E/LL/PS$ policy suffers from practical shortcomings:

Low Resource Efficiency: Least-loaded balancing spreads function invocations across all servers in a deployment. Particularly at low load, this results in poorly-utilized servers increasing costs [24], despite existing strategies to harvest unused cloud resources. Consolidating the workload in fewer servers is especially important in public clouds since performance metrics are hidden from the providers prohibiting them from easily harvesting spare resources without violating customer SLOs [41, 43, 75, 86, 88].

Increased Cold Starts: In Section 3, we showed that a scheduler that is locality-aware but not load-aware, like the OpenWhisk one, is suboptimal from a queueing theory perspective. However, function code locality is an essential consideration for serverless schedulers since cold starts can cause high function invocation latency. By spreading invocations across a larger number of servers, a scheduler that uses least-loaded balancing causes more function cold starts. At

low loads, there is no tendency for function invocations to be scheduled to a Worker with warm executors. Our goal is to bridge this gap by building a scheduler that is both locality and load-aware.

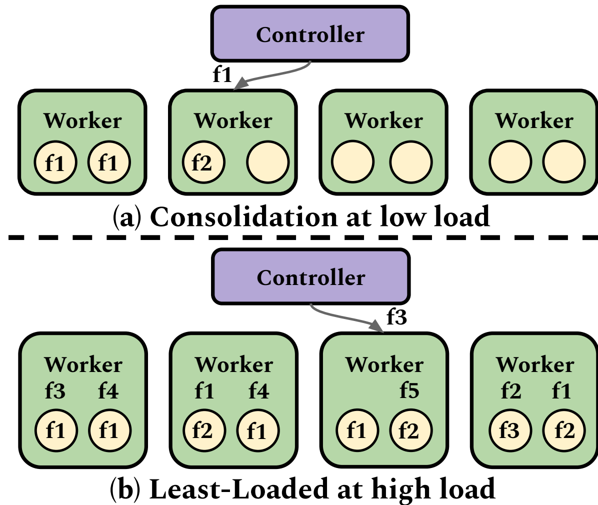


Figure 5: Hermod (a) packs function invocations to machines in a core-aware manner at low load and (b) uses least-loaded balancing when all cores are full. The yellow circles represent each Worker's CPU cores.

4.2 Hermod ($E/H/PS$)

Hermod uses *early binding and processor-sharing scheduling* to avoid head-of-line blocking that can cause high slowdown at high load *without assuming knowledge of the function execution times*. To address the shortcomings of least-loaded and late-binding schedulers, Hermod employs locality-aware *hybrid load balancing* (H), using one of two different load balancing modes depending on the load.

Low load: At low load, i.e., when there are available cores in some Worker, Hermod packs functions to Workers with available cores (Figure 5a). Starting from an arbitrary Worker, Hermod fills it with a number of invocations equal to the number of Worker cores. The same process is applied iteratively for all Workers in the deployment. When ongoing invocations finish execution, Hermod directs new incoming invocations to the corresponding servers filling them up. This policy of assigning up to N invocations to each N-core Worker, guarantees that there is no queuing at low load [67] while it consolidates function invocations.

High load: When all Workers host invocations equal to their number of cores, load balancing reverts to least-loaded (Figure 5b). This way, Hermod reduces queuing in overload situations. We chose not to explore different ways to define the threshold due to the lack of relevant information in published traces, e.g., CPU vs. IO mix for each function. When such information becomes available, we plan to develop approaches that dynamically set the threshold depending on the specific function mix. Hermod can accurately make the distinction between the two modes of operation by synchronously communicating with each Worker. This incurs low overhead since the Controller-Worker communication latency (O(1msec)) is much lower than function execution times (O((1sec)) and the bookkeeping is very lightweight, involving just updating a counter on the Controller side.

Locality: The hybrid load balancing described above alleviates the cold-start problem by consolidating invocations to a smaller number of servers. To improve even further, we explicitly make Hermod's load balancing policy locality-aware. Hermod keeps track of the available warm containers in each Worker. It first looks for a warm container in one of the non-empty Workers with available capacity when it operates at low-load mode. If such a Worker exists, Hermod steers the invocation there. If not, it prioritizes consolidation by selecting a non-empty Worker without a warm container instead of an empty Worker with a warm container. When it operates under high load, Hermod breaks ties using locality. If multiple Workers are equally loaded, it selects the one with a warm container for the incoming invocation if such a Worker exists.

In § 6.3 we show that Hermod improves locality compared to existing schedulers. Moreover, Hermod is compatible with recently proposed predictive approaches [74] that reduce function cold starts. In such designs, the controller spins up warm containers in anticipation of future function invocations. The only difference for Hermod is that its hybrid load balancing will run earlier, i.e., when the container is spun up instead of when the invocation arrives.

In summary, Hermod achieves the best of both worlds: high consolidation with low slowdown and low cold-start rate at low load while also maintaining low slowdown at high load.

4.3 Scalability

In order for cloud schedulers to scale, they need to be able to operate in a distributed fashion. Hermod's design is compatible with techniques used in distributed schedulers, such as sharding and power-of-k choices. In a sharded setup, each instance of Hermod can load-balance requests among the subset of Workers it owns. In a power-of-k choices architecture [58, 67], the scheduler samples k Workers and chooses one of them to send an incoming invocation to according to some scheduling policy. Hermod can operate by using

hybrid load balancing to choose one of the k sampled Workers. Moreover, in § 6.6 we show that Hermod does not affect OpenWhisk's scalability.

5 IMPLEMENTATION

We built Hermod for one of the widely adopted open-source serverless platforms, Apache OpenWhisk [5]. OpenWhisk's architecture is similar to that of a generic serverless platform as described in Section 2.1 allowing Hermod to be easily ported to a different platform or environment. The fact that OpenWhisk is used to power IBM's commercial serverless offering [7] increases the potential impact of the performance and efficiency improvements we achieve. However, our results are valid for other platforms beyond OpenWhisk. For example, for a framework like Fission that keeps a pool of warm containers around, Hermod can keep track of the placement of these containers and consider them when making the scheduling decision, like how it makes decisions based on available warm containers due to previous executions in OpenWhisk.

In this section, we first describe the details of OpenWhisk's architecture (§ 5.1). Then, we describe the three different schedulers we use as baseline in our evaluation (§ 5.2). In § 5.3, we discuss implementation details.

5.1 OpenWhisk Architecture

Function invocations pass through a reverse proxy that terminates SSL connections and hosts a public-facing HTTP endpoint before reaching the Controller. The Controller uses a separate persistent data store, e.g., CouchDB [1], to store the system state, including the list of registered functions and their code. Within the Controller lies a Load Balancer module which manages the scheduling of invocations to Workers. The OpenWhisk Workers are called Invokers. There is usually one per server and they use containers as executors. The Controller uses a reliable message bus, Apache Kafka [3], to send function invocations to Invokers for execution.

Cold starts: If a function invocation reaches an Invoker that does not have an available warm container for the specific function, we say that a cold start has occurred. In OpenWhisk two types of cold starts can happen. First, it is possible that the Invoker cached the function code or executable during a previous execution of the same function. In that case, the Invoker just needs to start a container with the function's runtime and inject the code or binary. Second, function code might not be cached at the Invoker. In that case, the Invoker fetches the function code from CouchDB and injects it to the corresponding container. OpenWhisk keeps containers alive for a 15-minute interval after their last invocation. Hermod takes that time interval into account when considering locality and can be easily adapted for different keep-alive policies [74].

5.2 Baseline Schedulers

We use the following schedulers as baselines:

OpenWhisk Scheduler ($E/LOC/PS$): The default OpenWhisk scheduler implements early binding, using locality-based load balancing and processor sharing at the Invokers (delegating scheduling to Linux's CFS), i.e., $E/LOC/PS$ using the notation we introduced in Section 3. Each Invoker has a fixed capacity for invocations based on its server's available memory. OpenWhisk' load balancer assigns invocations belonging to the same function to the same Invoker to reduce cold starts. A hash is calculated for every function and an Invoker is selected based on that hash. All invocations of that function are scheduled to that Invoker. If the Invoker selected from a function's hash is unhealthy or full, a new Invoker is randomly selected. This procedure is repeated until all Invokers have been checked at which point a healthy Invoker is randomly selected even if it does not have available capacity. If no healthy Invokers are available, the load balancer returns an error. Invocations are not queued anywhere.

This scheduler has shortcomings. Although the locality-based balancing helps reduce cold starts in some cases (§ 6.3), it is unsuitable for highly-skewed workloads. As we noted earlier, in real-world serverless workloads a small fraction of the functions accounts for most of the load (§ 2.2). This leads to server overloads and reduced performance (§ 6.2).

Late Binding: Late binding performs worse than early binding for workloads with highly-variable execution times at high load (§ 3.3). However, since many existing scheduling systems like Sparrow and Pigeon [67, 85] use such a policy, we implemented it for OpenWhisk to use it as a baseline in our evaluation. Function invocations are packed to Invokers until their core capacity is reached, i.e., until the number of active invocations becomes equal to the number of cores.Once this happens, additional invocations are queued at the Load Balancer. This policy is core-aware [48]: it requires the load balancer to take into account the number of cores in each Invoker, a departure from the original OpenWhisk scheduler model which only considers the memory capacity.

Least-Loaded ($E/LL/PS$): Based on the simulations, the ideal scheduling policy for a workload with the characteristics of serverless functions combines early binding with least-loaded balancing and processor-sharing at the workers. We implement such a scheduler for OpenWhisk to showcase that Hermod can outperform it due to improved locality without suffering from its efficiency shortcomings at low load.

5.3 Implementation Details

The new schedulers we introduce are implemented on top of OpenWhisk public repository [6]. For the Late Binding and Least-Loaded policies, an array is used to store Invoker load information. For Hermod, we use two such arrays, one for each mode of operation, and one map that keeps track of warm containers. The memory overhead of these arrays is minimal; in a setup with N Invokers $4 \times N$ bytes are needed for the Late Binding and Least-Loaded policies and $8 \times N$ bytes for Hermod. In our setup (described in § 6.1) this amounts to 32 and 64 bytes respectively. The overhead of the map is higher as, in the worst case, it needs to have an entry for each invocation that can fit in the cluster's memory. However, even in a large-scale deployment, this only amounts to a few megabytes. Late Binding has the additional memory overhead of storing the queued invocations at the Controller. Since this overhead was not a bottleneck factor in our setup and the proposed Hermod scheduler does not do queuing at the Controller, we do not quantify the exact memory overhead. In OpenWhisk, all invocations and completions go through the Controller. Hence, it is trivial to keep track of the load and the warm invocations on each Invoker. Similarly, switching Hermod's operating mode between "packing" and "least-loaded" is simply done with an if statement without any overhead. The three new policies are implemented in 2074 SLOC in total.

6 EVALUATION

We aim to answer the following questions:

(1) How does the scheduling policy affect a workload's performance? How effective is Hermod? (§6.2)
(2) How does scheduling affect cold-starts? (§6.3)
(3) What is the impact of different scheduling policies on resource consumption? (§6.4)
(4) Is Hermod robust across different execution time distributions? (§6.5)
(5) What is Hermod's overhead? (§6.6)

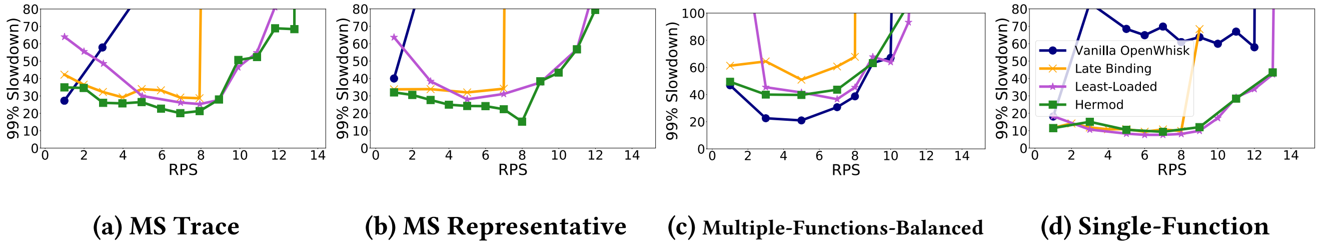
6.1 Experimental Methodology

Experimental setup: We use a cluster of 9 machines, connected through an Arista 7050-S switch with 48 10GbE ports. All machines have one Intel Xeon E5-2630 CPU operating at 2.3GHz running Ubuntu LTS 16.0.4 with the 4.4.0 Linux kernel and 32GB memory capacity. Each CPU has 12 cores. Hyperthreading is disabled to improve performance predictability. One machine hosts the OpenWhisk Controller and CouchDB while the rest of the machines host an Invoker each. Each Invoker can use up to 26,624MB, defined through the *userMemory* OpenWhisk parameter. Our setup has 96 Invoker cores in total, which is more than 2.5× the compute capacity used for the evaluation in [74]. We opted to do the evaluation in a private cluster to avoid the performance variability associated with the public cloud and focus only on the effect of scheduling on each workload's performance. We are confident our findings will translate to larger deployments since the proposed Hermod scheduler introduces no additional overhead compared with the vanilla OpenWhisk scheduler (Section 6.6). Similar to existing related work [73, 74], we use FaasProfiler as the load generator and modify it to collect runtime metrics across different servers that host OpenWhisk Invokers. The experiment for each data point runs for 1 hour.

Baselines: We compare Hermod with the following schedulers: (i) Vanilla OpenWhisk Scheduler, i.e., the built-in scheduler of a production-grade system, (ii) Late Binding Scheduler, i.e., an oracle version of the Sparrow [67] scheduler that is aware of the load in all cluster workers without needing sampling, (iii) Least-Loaded Scheduler, i.e., an improved version of the R2P2 [54] scheduler that uses processor-sharing to avoid head-of-line blocking.

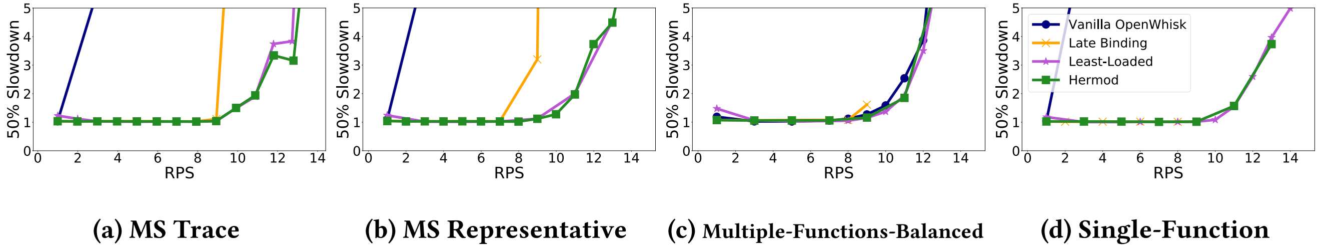
Workloads: To evaluate the performance of Hermod, we use five workloads, two derived from production traces, and three that test emerging serverless use cases and extreme scenarios.

First, we use **MS Trace**, a workload directly derived from the Azure production trace [74]. The trace includes information about function invocations across Microsoft's data centers and tens of thousands of different functions. To generate this workload, we use the same methodology that the authors of the paper that accompanied the trace release used for their analysis: (a) We select 50 different functions from the trace and scale the number of invocations to produce different load levels. To model the highly skewed function invocations found in the original trace, we randomly select one function of high popularity and 49 functions of medium popularity. Since the trace does not specify the implementation language of each function, we opted to use Javascript due to its prevalence in serverless computing [13]. Using a different language would not alter our conclusions since functions written in different languages face similar overheads [84]. All functions perform busy spinning for the duration of their execution. We implement busy spinning by repeating a simple numerical operation that we have timed in advance. Unfortunately, we did not have access to information such as the CPU/IO mix for each function. (b) We use a Log-normal distribution with $\mu = -0.38$ and $\sigma = 2.36$ for function execution times similar to the original trace. Invocations across all functions follow this distribution, capturing variability in running times even within a function. Similar characteristics also appear in Amazon's serverless offering [13], enhancing the validity of this workload.



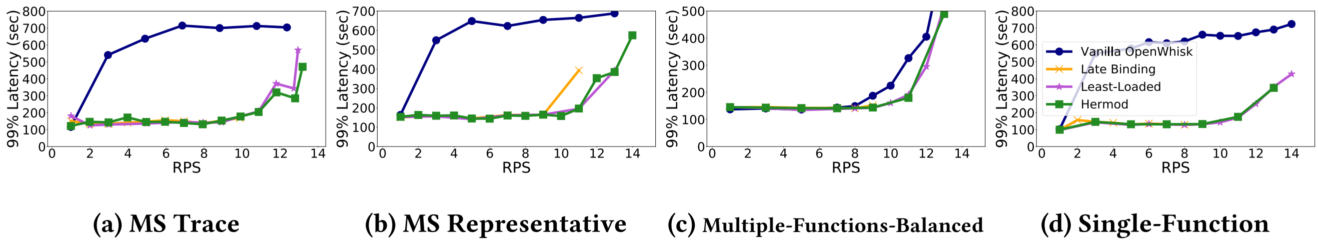
(a) MS Trace (b) MS Representative (c) Multiple-Functions-Balanced (d) Single-Function

Figure 6: 99% slowdown as function of the load in requests per second (RPS) for four different workloads.



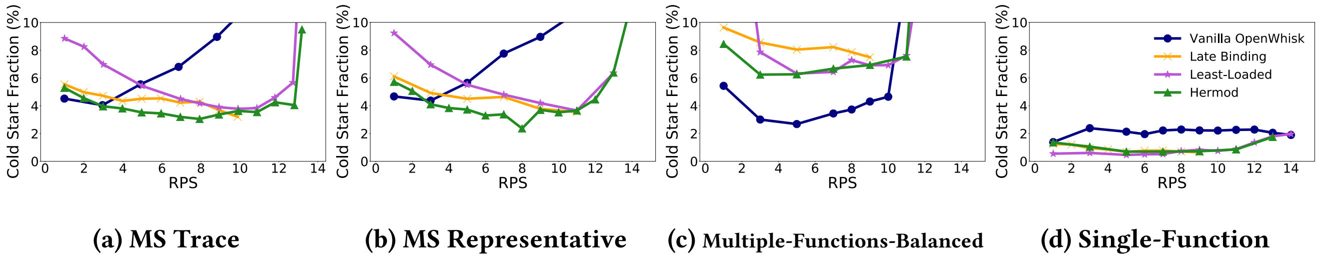
(a) MS Trace (b) MS Representative (c) Multiple-Functions-Balanced (d) Single-Function

Figure 7: 50% slowdown as function of the load in requests per second (RPS) for four different workloads.



(a) MS Trace (b) MS Representative (c) Multiple-Functions-Balanced (d) Single-Function

Figure 8: 99% latency as function of the load in requests per second (RPS) for four different workloads.



(a) MS Trace (b) MS Representative (c) Multiple-Functions-Balanced (d) Single-Function

Figure 9: % of cold start function invocations as a function of the load in requests per second (RPS).

We also need to ensure that workloads with different characteristics that might emerge in the future perform well under Hermod. To showcase Hermod's robustness, we consider additional workloads based on different scenarios across three axes: arrival pattern, skew, and execution time distribution.

- **Arrival Pattern:** To avoid making arbitrary low-level assumptions regarding missing data in the Microsoft production trace, such as how function arrivals are distributed within each minute, we use open-loop Poisson processes to model function invocation arrivals, following the standard practice in existing relevant research [87]. To model the original trace's skew, we use again 50 different functions, one accounting for 90% of the overall load while the rest equally account for the remaining 10%. We use the execution time distribution from the MS Trace. We refer to this workload as **MS Representative**.

- **Skew:** The *MS Trace* models the case of a highly skewed workload. We also evaluate the performance of Hermod in two very different scenarios. First, in the **Single-Function** workload, all function invocations belong to a single function. This pattern is characteristic of analytics applications where many functions are spun up to do the same type of processing [19, 32, 33, 45, 52, 83] and showcases Hermod's performance under extreme skew. Second, for the **Multiple-Functions-Balanced** workload, we deploy 50 different functions, each equally contributing to the overall load. This pattern is characteristic of an entirely homogeneous workload with zero skew. We use the execution time and interarrival time distributions from the MS Representative workload for both of these workloads.
- **Execution Time Distribution:** As mentioned before, *MS Trace*'s invocations have extremely high execution time variability. We want to ensure that Hermod provides good performance for workloads with more homogeneous execution times. To validate that, we create a **Homogeneous-Execution-Times** workload with the skew and inter-arrival distribution of the MS Trace workload but with execution time that follows a light-tailed exponential distribution. We set the mean to 8.9 seconds, similar to the Log-normal distribution of the MS Trace workload. The performance of this workload is evaluated in § 6.5.

We chose not to consider the memory allocation separately as the Azure trace provides the per application (group of functions) rather than the per function allocation making a faithful reproduction impossible. Thus, we set the memory used by each function to 256MB across all workloads, with each Invoker having a capacity of up to 26624/256 = 104 function invocations. If we had variable memory sizes, that number would differ depending on the function mix in each server. However, Hermod would still be able to schedule invocations to servers with warm containers and track when the server runs out of memory.

6.2 Performance Analysis

Figure 6 compares the 99% slowdown of four different workloads under the four different schedulers; we make the following observations. First, similar to what we observed in the simulations, Late Binding and the Vanilla OpenWhisk schedulers can support a lower load than the rest of the schedulers. For Late Binding, the 99% slowdown explodes for a 39% lower load (8 vs. 13 RPS) than the Least-Loaded and Hermod schedulers for the MS Trace, MS Representative, and Single-Function workloads. For Vanilla OpenWhisk, the 99% slowdown explodes from very low loads due to the skew. Second, Hermod provides up to 50% lower slowdown at low and medium load compared to the locality-unaware Least-Loaded scheduler. Third, for the Multiple-Functions-Balanced workload, we observe a different behavior. Each function equally contributes to the overall load allowing the locality-based balancing used by Vanilla OpenWhisk to distribute the load equally among the Invokers and achieve 99% slowdown better than the other policies at low load.

Shown in Figure 7, the 50% slowdown shows similar characteristics to those of the 99% slowdown, with the Hermod and Least-Loaded schedulers outperforming the Vanilla OpenWhisk and Late Binding ones. We also observe in Figure 8 that, as the simulations showed (Figure 2a), the different schedulers perform similarly in terms of tail latency, with the Vanilla OpenWhisk one performing poorly in the real-world skewed workloads.

Conclusion: The observations confirm the conclusions drawn by our simulations that the least-loaded policy used by Hermod can support higher load both than Late Binding and pure locality-based approaches (Vanilla OpenWhisk). Moreover, the fact that Hermod is locality-aware allows it to achieve a lower slowdown than pure load-aware scheduling. Hermod retains these performance gains even for improbable homogeneous workloads, sacrificing little slowdown. We also show that the Vanilla OpenWhisk scheduler is optimized for the "wrong" workload, i.e., a completely homogeneous mix of functions that does not appear in practice.

6.3 Cold Start Analysis

Figure 9 presents the percentage of cold starts for the four workloads. Hermod triggers very few cold starts for the MS Trace and MS Representative workloads as it steers invocations to Invokers with warm containers when possible. The Least-Loaded scheduler has a high cold start rate at low and medium load: containers for all 50 different functions need to be started in all 8 Invokers. The higher the load, the fewer the cold starts under the Late Binding scheduler since it is more likely that an invocation is first queued at the Controller and hence is scheduled to a warm container finishing the execution of a previous invocation.

The Vanilla OpenWhisk scheduler generates more cold starts for the Single-Function, MS Trace, and MS Representative workloads because it tends to overload individual Invokers. An overloaded Invoker is less likely to have a warm container available; a new one often needs to be created. However, validating our slowdown measurements, Vanilla OpenWhisk has fewer cold starts than the other schedulers for the Multiple-Functions-Balanced workload due to its sticky load balancing. Invocations of the same function are more likely to be scheduled to the same Invoker finding a warm container.

Conclusion: Hermod, being both locality- and load-aware, causes fewer cold starts, explaining the lower slowdown

shown in Figure 6. Predictive policies [61, 74], which are orthogonal to Hermod, could reduce cold starts even more.

6.4 Resource Consumption

We present the average resource utilization when using different schedulers for the MS Trace workload (Figure 10). For this experiment, we collect utilization metrics every second and consider a server to be utilized if a serverless function uses it over the 1-second monitoring period. We omit the results for the other workloads since they are almost identical.

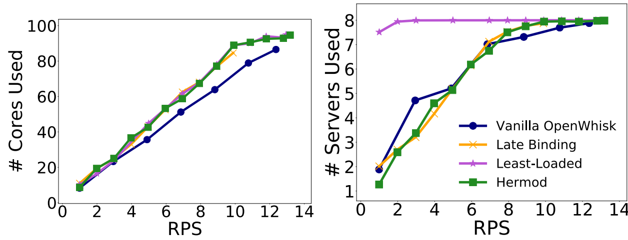


Figure 10: Average # Cores and # Servers utilized during the execution of the MS Trace workload.

The number of cores used is similar across policies. However, when we consider server utilization, the advantages of Hermod are apparent. It uses fewer servers than the Least-Loaded scheduler at low load by packing invocations, while it achieves lower slowdown. The other two schedulers (Late Binding and Vanilla OpenWhisk) use about the same number of servers as Hermod while providing worse performance.
Conclusion: Hermod achieves a lower slowdown than the Least-Loaded scheduler while providing better consolidation.

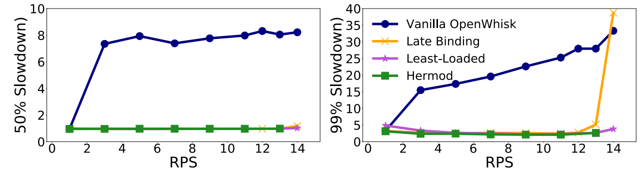
6.5 Different Execution Time Distributions

In Figure 11, we observe that the slowdown for the *Homogeneous-Execution-Times* workload under Hermod - both at the median and the tail - is very similar to the slowdown under the Least-Loaded and Late Binding policies. The Vanilla OpenWhisk scheduler again causes a higher slowdown at low load due to skewed function load and sticky load balancing.

Conclusion: Hermod matches or exceeds the performance achieved by existing schedulers even for workloads with homogeneous execution times, demonstrating its robustness.

6.6 OpenWhisk's Overheads

In this section, we examine whether OpenWhisk's inherent overheads affect our conclusions regarding Hermod. The table below shows each scheduler's highest throughput, i.e., the maximum number of zero-work function invocations completed in one second. In this scenario, the



(a) 50% Slowdown **(b) 99% Slowdown**

Figure 11: Slowdown as a function of the load for the Homogeneous-Execution-Times workload.

Controller/Scheduler is the bottleneck since the Invokers never exceed 40% utilization. The throughput is similar for all schedulers showing that Hermod does not cause any additional overhead.

Scheduler	Throughput (RPS) ± std
Vanilla OpenWhisk	3833.8 ± 43.8
Late Binding	3830.2 ± 93.4
Least-Loaded	3832.4 ± 58.0
Hermod	3818.4 ± 32.0

Moreover, we see that the throughput achieved in the rest of our more realistic experiments is much smaller than the maximum one achieved by all schedulers. Hence, any difference in performance we observe between the different schedulers results from the scheduling policies they use rather than the mechanisms they employ.

We also examine the effect of OpenWhisk's overheads on the slowdown. Under normal circumstances, the path of a warm invocation through the HTTP server, CouchDB, Load Balancer and Apache Kafka to the Invoker takes only a few milliseconds. In particular, the scheduling decision takes less than 0.5msec, contributing little to the overall overhead. Thus, when the system operates under low load, the median slowdown is approximately 1, i.e., the minimum possible. However, the 99% slowdown is 20-30 even for low load, as shown in Figure 6. This is an artifact of OpenWhisk's cold-start overheads, as creating and initializing a Docker container in the Invoker takes a few tens milliseconds. More efficient container start-up mechanisms (e.g., pre-warming, snapshotting, etc.) are orthogonal to our work as they could reduce the 99% slowdown across all schedulers, but they would not change each worker's load and thus the shape of the curve.

7 CONCLUSION

This paper presents Hermod, a scheduler that caters to the unique characteristics of serverless functions and achieves near optimal performance. Hermod uses three key techniques: early binding, hybrid load balancing, i.e., consolidation at low load and least-loaded balancing at high load, and

processor sharing scheduling at the individual workers. Implementing Hermod for Apache OpenWhisk we demonstrate that it can provide up to 85% lower slowdown and support up to 60% higher load compared to existing approaches for a real-world production workload. Additional experiments demonstrate that Hermod is robust to workloads with different function mixes and execution time distributions. Hence, we believe that Hermod provides an answer to how scheduling for serverless functions should be done.

REFERENCES

[1] 2005. Apache CouchDB. https://couchdb.apache.org/.

[2] 2007. Completely Fair Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[3] 2011. Apache Kafka. https://kafka.apache.org/.

[4] 2014. AWS Lambda. https://aws.amazon.com/lambda/.

[5] 2016. Apache OpenWhisk. https://openwhisk.apache.org/.

[6] 2016. Apache OpenWhisk Git Repository. https://github.com/apache/openwhisk.

[7] 2016. IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[8] 2016. OpenFaaS. https://www.openfaas.com/.

[9] 2017. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[10] 2018. Google Cloud Functions. https://cloud.google.com/functions/.

[11] 2018. Kubeless. https://kubeless.io/.

[12] 2019. Fission. https://fission.io/.

[13] 2020. State of Serverless. https://www.datadoghq.com/state-of-serverless/.

[14] 2021. Knative. https://knative.dev/.

[15] 2022. Fission Auto-scaler. https://fission.io/blog/autoscaling-serverless-functions-with-custom-metrics/.

[16] 2022. Knative Auto-scaler. https://knative.dev/docs/serving/autoscaling/scale-bounds/.

[17] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[18] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[19] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[20] Nikhil Bansal and Mor Harchol-Balter. 2001. Analysis of SRPT Scheduling: Investigating Unfairness. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Cambridge, Massachusetts, USA) *(SIGMETRICS '01)*. Association for Computing Machinery, 279–290. https://doi.org/10.1145/378420.378792

[21] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[22] Onno Boxma and Bert Zwart. 2007. Tails in Scheduling. *SIGMETRICS Perform. Eval. Rev.* 34, 4 (March 2007), 13–20. https://doi.org/10.1145/1243401.1243406

[23] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, Heraklion, Greece, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[24] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. 2018. Stratus: Cost-Aware Container Scheduling in the Public Cloud. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. Association for Computing Machinery, Carlsbad, CA, USA, 121–134. https://doi.org/10.1145/3267809.3267819

[25] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China, 153–167. https://doi.org/10.1145/3132747.3132772

[26] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-Aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. Association for Computing Machinery, Santa Clara, CA, USA, 497–509. https://doi.org/10.1145/2987550.2987563

[27] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2018. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. Association for Computing Machinery, Carlsbad, CA, USA, 135–148. https://doi.org/10.1145/3267809.3267838

[28] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid Datacenter Scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 499–510. https://www.usenix.org/conference/atc15/technical-session/presentation/delgado

[29] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Salt Lake City, UT, USA).

[30] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)* (Kohala Coast, HI, USA).

[31] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 467–481. https://doi.org/10.1145/3373376.3378512

[32] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[33] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[34] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, Virtual, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[35] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, Porto, Portugal, Article 4, 13 pages. https://doi.org/10.1145/3190508.3190549

[36] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 99–115. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[37] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 181–195. https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[38] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[39] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. 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* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 295–308.

[40] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, Princeton, NJ, USA, 60–68. https://doi.org/10.1145/3365609.3365856

[41] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 519–532. https://www.usenix.org/conference/atc18/presentation/iorgulescu

[42] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, Big Sky, Montana, USA, 261–276. https://doi.org/10.1145/1629575.1629601

[43] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. 2019. Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. Association for Computing Machinery, Santa Cruz, CA, USA, 272–285. https://doi.org/10.1145/3357223.3362734

[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[45] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, Santa Clara, CA, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[46] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[47] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[48] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. Association for Computing Machinery, Santa Cruz, CA, USA, 158–164. https://doi.org/10.1145/3357223.3362709

[49] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 485–497. https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos

[50] David G. Kendall. 1953. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics* 24, 3 (1953), 338–354. http://www.jstor.org/stable/2236285

[51] Eun-Kyeong Kim and Hang-Hyun Jo. 2016. Measuring burstiness for finite event sequences. *Physical Review E* 94, 3 (Sep 2016). https://doi.org/10.1103/physreve.94.032311

[52] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[53] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. https://doi.org/10.1145/3342195.3387545

[54] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens.

In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 863–880. https://www.usenix.org/conference/atc19/presentation/kogias-r2p2

[55] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2020. Sol: Fast Distributed Computation Over Slow Networks . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 273–288. https://www.usenix.org/conference/nsdi20/presentation/lai

[56] Jialin Li, Jacob Nelson, Xin Jin, and Dan Ports. 2018. *Pegasus: Load-Aware Selective Replication with an In-Network Coherence Directory*. Technical Report UW-CSE-18-12-01. University of Washington. https://www.microsoft.com/en-us/research/publication/pegasus-load-aware-selective-replication-with-an-in-network-coherence-directory/

[57] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[58] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.* 12, 10 (Oct. 2001), 1094–1104. https://doi.org/10.1109/71.963420

[59] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, Budapest, Hungary, 221–235. https://doi.org/10.1145/3230543.3230564

[60] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[61] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. *OFC: An Opportunistic Caching System for FaaS Platforms*. Association for Computing Machinery, 228–244. https://doi.org/10.1145/3447786.3456239

[62] Ingo Müller, Rodrigo Bruno, Ana Klimovic, John Wilkes, Eric Sedlar, and Gustavo Alonso. 2020. Serverless Clusters: The Missing Piece for Interactive Batch Applications?. In *10th Workshop on Systems for Post-Moore Architectures (SPMA 20)*.

[63] Misja Nuyens and Adam Wierman. 2008. The Foreground-Background Queue: A Survey. *Perform. Eval.* 65, 3–4 (March 2008), 286–307. https://doi.org/10.1016/j.peva.2007.06.028

[64] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[65] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[66] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China, 184–200. https://doi.org/10.1145/3132747.3132766

[67] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, Farminton, Pennsylvania, 69–84. https://doi.org/10.1145/2517349.2522716

[68] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China, 325–341. https://doi.org/10.1145/3132747.3132780

[69] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy Proportionality and Workload Consolidation for Latency-Critical Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. Association for Computing Machinery, Kohala Coast, Hawaii, 342–355. https://doi.org/10.1145/2806777.2806848

[70] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. 2018. Decoupling the Control Plane from Program Control Flow for Flexibility and Performance in Cloud Computing. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, Porto, Portugal, Article 1, 13 pages. https://doi.org/10.1145/3190508.3190516

[71] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. arXiv:2104.13869 [cs.DC]

[72] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open versus closed: A cautionary tale. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 06)*. USENIX Association, San Jose, CA.

[73] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296

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

[75] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, Dresden, Germany, Article 33, 17 pages. https://doi.org/10.1145/3302424.3303945

[76] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[77] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. Association for Computing Machinery, Seattle, WA, USA, 138–152. https://doi.org/10.1145/3472883.3486981

[78] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: Ephemeral Endpoints for Serverless Networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, Virtual Event, USA, 16–29. https://doi.org/10.1145/3419111.3421275

[79] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, Heraklion, Greece, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[80] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots *(ASPLOS 2021)*. Association for Computing Machinery, Virtual, USA, 559–572. https://doi.org/10.1145/3445814.3446714

[81] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. Association for Computing Machinery, Santa Clara, California, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[82] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.

[83] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[84] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Boston, MA, USA, 133–145. http://dl.acm.org/citation.cfm?id=3277355.3277369

[85] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. 2019. Pigeon: An Effective Distributed, Hierarchical Datacenter Job Scheduler. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. Association for Computing Machinery, Santa Cruz, CA, USA, 246–258. https://doi.org/10.1145/3357223.3362728

[86] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. 2017. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Association for Computing Machinery, Belgrade, Serbia, 575–588. https://doi.org/10.1145/3064176.3064181

[87] Y. Zhang, D. Meisner, J. Mars, and L. Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 456–468.

[88] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 755–770. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-yunqi

[89] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta. https://www.usenix.org/conference/osdi20/presentation/zhu