ServerMore: Opportunistic Execution of Serverless Functions in the Cloud

Amoghavarsha Suresh, Anshul Gandhi
PACE Lab, Stony Brook University
Stony Brook, NY, USA
{amsuresh,anshul}@cs.stonybrook.edu

ABSTRACT

Serverless computing allows customers to submit their jobs to the cloud for execution, with the resource provisioning being taken care of by the cloud provider. Serverless functions are often short-lived and have modest resource requirements, thereby presenting an opportunity to improve server utilization by colocating with latency-sensitive customer workloads. This paper presents ServerMore, a server-level resource manager that opportunistically colocates customer serverless jobs with serverful customer VMs. ServerMore dynamically regulates the CPU, memory bandwidth, and LLC resources on the server to ensure that the colocation between serverful and serverless workloads does not impact application tail latencies. By selectively admitting serverless functions and inferring the performance of black-box serverful workloads, ServerMore improves resource utilization on average by 35.9% to 245% compared to prior works; while having a minimal impact on the latency of both serverful applications and serverless functions.

CCS CONCEPTS

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

1 INTRODUCTION

Serverless computing is an emerging paradigm enabled by cloud computing wherein customers only need to submit their jobs (referred to as functions) and the provider will execute it for them on their servers. Compared to the traditional "serverful" cloud computing model where customers are responsible for their possibly elastic resource requirements (e.g., via renting the number and type of VMs needed), serverless computing shifts the burden of resource management to the cloud service provider [37]. Given the ease of access and the many benefits, such as virtually unlimited capacity and economical computing power, many different domains are translating their applications, where possible, to serverless computing functions [11, 35, 37, 41].

A valuable opportunity presented by serverless functions, that we explore in this paper, is that they can improve resource utilization in traditional VM-centric customer clouds by consuming the spare and idle resources left behind after provisioning VMs on a server. Today's serverless functions are ideally suited for this opportunity because of their specific properties—they are often short-lived with modest resource requirements [49]. However, there are several challenges that must be addressed when attempting to safely colocate customer's serverless functions with customer VMs to alleviate the critical under-utilization problem [7] in clouds:

(1) The key problem is to ensure that serverless functions get executed ***without impacting the performance of serverful*** (i.e., VMs, in our context) applications. Note that *both* serverless functions and serverful applications are customer jobs in our context, and thus latency-sensitive.
(2) The customer VMs can have varying resource configurations with ***dynamically changing arrival patterns*** [50]. Thus, the serverless colocation must adapt, in real-time, to the serverful resource needs.
(3) Serverless functions can have ***diverse resource requirements and execution times***. Colocating a serverful application with different functions can result in a variety of interference scenarios, including scenarios where the resource contention is too severe to resolve.
(4) Colocating serverful applications with serverless functions can result in contention for ***different resources simultaneously***, including CPU, memory, and cache. Despite the resource partitioning mechanisms provided by modern servers [17, 18], avoiding contention for a single resource is non-trivial, let alone for multiple resources. Worse, serverful applications can have *different sensitivity to contention for different resources* [4, 22].

SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3486979

(5) A practical challenge that exacerbates colocation with customer VMs is that the cloud provider ***may not have visibility of application performance within the VM***. This makes it difficult for the provider to protect the tail latencies of serverful applications.

Prior work in black-box performance management of customer cloud applications has focused on colocating customer applications with *latency-insensitive* provider or batch jobs that can be *paused or throttled* to maintain acceptable customer application performance [1, 21, 22, 27, 50]. We discuss related work in detail in Section 2. By contrast, in our setup, serverless workloads are *customer-centric* and are being colocated with latency-sensitive customer VMs with the goal of improving resource utilization.

We present ServerMore, a server-level resource manager that opportunistically executes customer serverless jobs on spare and idle resources colocated with customer VMs. ServerMore is designed with the ***objective*** of *maximizing the resource utilization of cloud servers while ensuring that the serverful latency is within an acceptable range.* We aim for a serverful tail latency (P99) degradation threshold of <10%. Further, despite the serverless functions being executed in a "best-effort" manner in cloud offerings [31, 45], we aim to minimize the tail latency of serverless functions.

Given the *black-box* nature of customer VMs, ServerMore does not rely on monitoring their latency, unlike prior works [4, 27]. Instead, ServerMore uses a proxy statistic to infer VM application performance, and accordingly determines the spare capacity that can be advertized to serverless workloads. Further, ServerMore dynamically responds to variations in serverful workload, adjusting the advertized capacity as needed.

To account for the *diverse nature of serverless functions*, ServerMore quickly characterizes incoming functions and accordingly decides on whether or not to colocate their subsequent invocations with customer VMs. Given the modest resource needs of many functions, ServerMore colocates multiple functions to significantly improve server utilization and carefully regulates the resource allocation between functions. Note that, in contrast to serverful workloads, the latency and execution details of serverless functions are visible to providers; major cloud providers like AWS already offer monitoring tools for serverless functions [39].

ServerMore actively regulates the *sharing of multiple resources* (CPU, memory bandwidth, and LLC), unlike much of the prior work that is focused on a single resource [21, 50, 51]. In fact, when colocating with serverless functions, we show that managing only one resource, such as CPU or LLC, is not enough to provide serverful performance isolation. By managing multiple resources, ServerMore is able to also safely colocate functions on the cores allocated to serverful applications.

We implement ServerMore by modifying Apache OpenWhisk and employing a light-weight user-space daemon. Across multiple latency-sensitive VM applications and diverse serverless functions, we experimentally show that ServerMore improves resource utilization significantly. Under all scenarios that we experiment with, including time-varying workload, ServerMore ensures that the serverful tail latency degradation is below 10%. We also empirically compare with prior works that provide performance isolation on colocated cloud servers and show that ServerMore improves resource utilization on average by 35.9% to 245% across a variety of colocation scenarios while meeting serverful performance targets and minimizing serverless function latency.

2 RELATED WORK

In recent years, the problem of resource under-utilization has received increasing attention from the research community. In this section, we summarize the relevant prior works and contextualize the problem we are addressing in this paper.

White-box resource management: Heracles [27] proposes to colocate latency-critical (LC) workloads with best-effort batch jobs by using the SLO of the latency-critical workload as a feedback signal to allocate resources for batch jobs. Heracles performs core, cache, and network bandwidth regulation, but is only applicable to workloads whose SLOs are available. Borg [48] and Bistro [14] are cluster-level schedulers that colocate LC applications with batch jobs; however, these private-cloud schedulers have complete visibility of the application latency. PARTIES [4] proposes to colocate multiple LC workloads by tracking their individual SLOs; PARTIES regulates resources based on how close each workload is to its SLO. However, in a public cloud, customer applications on serverful instances are black-boxes whose performance information cannot (or should not) be accessed by providers [22, 30, 33].

Black-box resource management: $PerfIso$ [21] colocates LC workloads with batch jobs by maintaining a buffer of idle cores. To determine the right number of buffer cores, $PerfIso$ performs offline characterization of the LC workload. Such profiling of customer workload may not be possible in a public cloud setup since the workload would not be in the provider's control and identifying representative workload behavior would be infeasible [22, 50]. *Scavenger* is a black-box solution to colocate LC workloads in a public cloud with batch jobs. *Scavenger* aggressively throttles the resource allocation of batch jobs to maintain an acceptable range of IPC for the serverful workload. We show, experimentally, the many benefits of ServerMore over $PerfIso$ and *Scavenger* in Section 5.

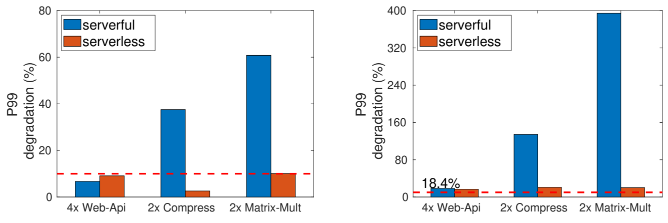
Recently, a pair of works [1, 50] have proposed a new class of VMs, called Harvest-VM, to run batch jobs along with customer VMs. The resources provided for Harvest-VMs can be changed at any time including eviction of the VM. The focus in Ambati et al. [1] is on harvesting the spare unallocated resources. In SmartHarvest [50], the focus is on harvesting resources which have been allocated to customer VMs but are unused and can thus be allocated to Harvest-VMs. In both the works, the focus is on predicting and exploiting only CPU cores that are idle, and not other resources. Further, these works do not directly react to latency degradation of the LC application. In contrast, we infer the performance of the serverful application and accordingly exploit the spare CPU, LLC, and memory bandwidth resources. Finally, the implementation and historical traces used in these works is not publicly available for comparison, and given the reliance of the predictors on the training data, the implementation is non-trivial to emulate.

Serverless resource management: In Archipelago [42], a new serverless platform is proposed with deadline-aware scheduling and proactive spawning of sandboxes to reduce latency of the functions. However, Archipelago does not focus on improving resource utilization. ENSURE [44] uses the SLO of serverless functions to efficiently colocate multiple serverless functions. Both the above works are orthogonal to ServerMore as they do not consider serverful workloads, which continue to be popular with customers [37]. Other works have focused on specific problems with serverless scheduling such as coldstarts [34, 40, 43], latency of serverless function chains [46], and resource under-utilization in function chains [16].

3 MOTIVATION AND CHALLENGES

In this section, we study the performance impact of colocating serverful workloads with serverless functions under various resource regulation mechanisms supported by modern servers. In this work, we focus on potential interference for resources *within* a server: CPU, LLC, and Memory. We do not consider local disk contention since serverless functions typically employ distributed storage [26, 36].

Applications: In this motivation section, we use TensorFlow Serving, or TF-Serving, a latency-sensitive application from PerfKit Benchmarker [13], as our serverful application. The TF-Serving application is run on a VM, referred to as *primary VM*, with 4 vCPUs and 32 GB memory with the VM being pinned to 4 physical cores. We use the following serverless functions to stress different resources: Matrix-Multiplication (100% CPU usage), SeBS-Compress (32% CPU usage), Web-Api (8% CPU usage), Memory-Stream (memory bandwidth intensive), LLC-Reg (LLC intensive), and LLC-Rand (LLC intensive). Web-Api [3] and SeBS-Compress are functions from recent works [6]. The memory and LLC stress serverless functions are based on the popular memory stress benchmark STREAM [28], with their working set size proportional to the resource they stress. LLC-Reg and LLC-Rand have regular and random access patterns, respectively. The serverless functions are allocated 512 MB upon invocation.



(a) TF-Serving RPS=4 (b) TF-Serving RPS=12

Figure 1: CPU Interference: Colocation of TF-Serving serverful workload with different functions under varying degrees of CPU requirement. The red dashed line indicates the 10% P99 degradation threshold that we aim to stay under.

Methodology: To study the performance impact, we use a server with 10 physical cores (and 11 cache-ways); the server supports cache and memory bandwidth partitioning via CAT and MBA [20], respectively. Section 5.1 has detailed information on our testbed. Unless specified otherwise, we run the TF-serving application for 150 seconds with a constant load of 12 requests per second (RPS). For the serverless workload, multiple requests/copies of the same function are dispatched in batches, with each batch being dispatched after the previous batch has been served; we vary the number of simultaneous copies of the function that are executed across experiments. We use the 99th percentile latency (P99) as our performance metric. Similar to recent studies on cloud application performance, we set the acceptable latency degradation target to 10% [50]. We run each experiments 3 times and report the average P99 across the runs.

3.1 CPU interference

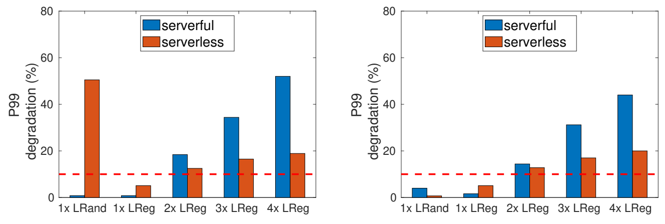
We start by analyzing CPU interference to answer the question *is it feasible, with respect to performance, to share cores between serverless functions and serverful applications?* Figure 1(a) shows the degradation in P99 relative to no colocation when TF-Serving (with load of 4 RPS) is colocated with various CPU-stressing serverless functions. The x-axis labels indicate the number of simultaneous serverless requests in each batch. Both serverful application and the serverless functions are run on the same set of cores. As is typically the case (e.g., Amazon Lambda [38]), the serverless functions are allotted CPU proportional to their memory, i.e., $allot\_cpu = (requested\_memory/1024)$; this allottment is enforced through the cgroups CPU isolation mechanism of *cpu-shares*.

Colocating with Web-Api has the lowest impact (6-9%) on P99 degradation of both serverful and serverless functions, with both being in the acceptable limit of 10% P99 degradation. Web-Api has low CPU requirement and despite 4 copies of the function running simultaneously, the impact is minimal. In contrast, only 2 copies each of the relatively CPU heavy functions of SeBS-Compress or Matrix Multiplication can have severe impact on the serverful application's P99 latency degradation with as much 61% degradation in the case of MM colocation.

In Figure 1(b), we increase the TF-Serving load to 12 RPS. The average CPU utilization of TF-serving at 12 RPS is 87%, which is significantly higher than the 36% CPU utilization at 4 RPS. When colocating with Web-Api, SebS-Compress, and Matrix Multiplication, the P99 of serverful exhibits 18.4%, 134%, and 394% degradation, respectively. Clearly, the serverful workload has an impact on performance degradation; while Web-Api provided acceptable degradation when TF-Serving was run at RPS of 4, this is no longer the case.

Takeaway 1. *At low levels of serverful CPU utilization,* cpu-shares *can be used to safely (with respect to tail latency) colocate some serverless functions but not others. As the serverful load changes, the potential candidate functions for safe colocation will change as well.*

3.2 LLC interference



(a) No isolation (b) Cache partitioning

Figure 2: LLC Interference: Colocation of TF-Serving workload with LLC-sensitive serverless functions.

3.2.1 Colocation without LLC isolation: Figure 2(a) shows the results of colocating LLC-Reg and LLC-Rand serverless functions with TF-Serving on the same socket, hence sharing LLC, but run on different cores. We see that colocation with LLC-Rand results in significant (50%) P99 latency degradation for serverless. When colocating with multiple copies of LLC-Reg, TF-Serving experiences P99 degradation of 18–52%, with LLC-Reg facing 12–18% latency degradation. With multiple copies of LLC-Reg involved in a colocation experiment, the LLC interference can happen among the serverless functions *and* between serverful and serverless functions. In fact, with multiple applications contending for the limited amount of LLC capacity, there is also contention for memory bandwidth, which exacerbates the performance degradation.

3.2.2 Colocation with LLC isolation: Modern CPUs equipped with Intel Resource Director Technology (RDT) [20] provide functionality to effectively partition the LLC among the processes. Specifically, Intel RDT allows us to specify a resource control tag called class of service (COS) [17], which can be used to control the available resources to a group of processes, applications, VMs, or containers. The cache specific control provided by RDT is popularly known as Cache Allocation Technology (CAT), that allows us to specify physical cache lines that can be used by a COS. In our setup, we partition the 11 cache-ways proportional to the core allocation, resulting in 5 and 6 cache-ways exclusively dedicated to the VM and the serverless functions, respectively.

Figure 2(b) shows our LLC interference results when using LLC partitioning as described above; we use the same set of applications as in Section 3.2.1. We see that, with LLC partitioning, the LLC-Rand function no longer experiences significant performance degradation. However, we see that colocating multiple copies of LLC-Reg *continues* to impact P99 latency of both serverful and serverless applications, although the degradation is slightly lower than without LLC partitioning in Figure 2(a). This is not necessarily a failure of the LLC partitioning provided by CAT since with multiple applications simultaneously contending for limited LLC, the contention spills over to memory bandwidth.

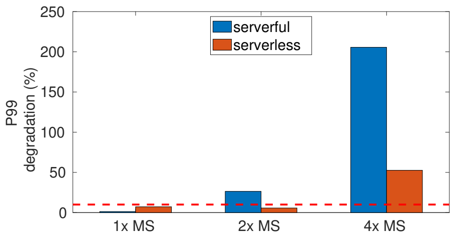
Takeaway 2. *Providing performance isolation for a single resource may not be enough to maintain acceptable tail latencies.*

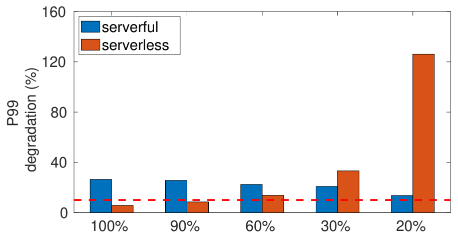
3.3 Memory Bandwidth interference

We now consider memory bandwidth interference. Note that a serverless function request is specified with a required amount of memory. Thus, a serverless function can only be run on a server if sufficient memory is available and hence memory capacity interference is not a concern.

3.3.1 Colocation without isolation. Figure 3(a) shows the results of colcating TF-serving with Memory-Stream (MS) serverless function (with an average memory bandwidth usage of 8.5 GBps) on the same socket and NUMA domain, but run on different cores. We see that with 2 and 4 copies of serverless functions, TF-Serving experiences non-trivial P99 degradation of 26.4% and 205.6% respectively, with the serverless workload facing 5.7% and 52.6% degradation.

3.3.2 Colocation with Memory Bandwidth isolation. Modern servers equipped with Intel Memory Bandwidth Allocator (MBA) [20] provide two mechanisms to regulate memory bandwidth—*throttling* and *capping*. *Throttling* regulates the memory accesses of a COS by specifying throttle value as a percentage of the maximum bandwidth.The throttle percentages are approximate and serve as a hint for how much throttling should be applied [18]. *Capping* involves memory bandwidth monitoring to track and cap the memory bandwidth for a COS.


(a) No isolation


(b) 2 copies of Memory-Stream with isolation

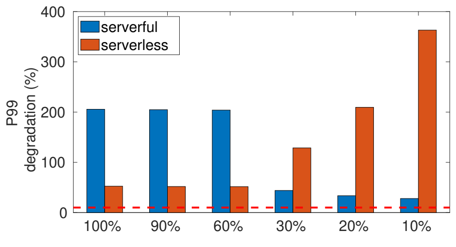

(c) 4 copies of Memory-Stream with isolation

Figure 3: Memory Bandwidth Interference: Latency degradation with and without the use of *Throttling* memory bandwidth isolation. Note that the x-axis scale varies across the subfigures.

Figure 3(b) shows the results of colocating 2 copies of Memory-Stream with TF-Serving when using the *throttling* mechanism. We apply the *throttling* mechanism to serverless functions. The throttle value represents the maximum allowable bandwidth with 100% representing no throttling and 10% representing maximum throttling. While throttling reduces the latency degradation of serverful, aggressive throttling (only 20% bandwidth available to serverless) is needed to limit serverful P99 degradation to below 10%, which comes at the expense of severe degradation for serverless.

Figure 3(c) shows the results of colocating 4 copies of Memory-Stream with TF-Serving when using the *throttling* mechanism. The latency degradation is considerably higher here when compared to Figure 3(b) where only 2 copies of Memory-Stream were colocated. In fact, even with the maximum throttling of 10%, TF-Serving still faces 28% latency degradation. These results show that while throttling can alleviate serverful degradation, the exact throttling value is not obvious and in some cases even maximum throttling is insufficient. Further, throttling hurts serverless performance. The colocation results with *capping* are similar to *throttling* (thus omitted) and have similar tradeoffs.

Takeaway 3. *Resource regulation mechanisms provided by modern servers require workload-specific tuning for performance isolation. Despite such tuning, some serverless functions cannot be safely colocated with serverful applications.*

4 DESIGN OF SERVERMORE

We consider a *public cloud* setup wherein customers can request VMs or submit serverless jobs. To minimize contention, no other workload (such as provider batch jobs) is scheduled on this customer-centric cloud. While VM requests include requirements for all resources, serverless requests only indicate the required amount of memory, as is the case for AWS Lambda [38] and Google Cloud Functions [15]. The CPU resources for serverless functions are then allocated proportional to the memory requirement.

We assume that there is a continuous stream of customer requests for VMs and serverless functions, and that there is enough cloud capacity to handle all requests. Our ***objective*** is to *maximize the resource utilization of the cloud servers while ensuring that the serverful latency is within an acceptable range. Further, despite serverless being executed in a "best-effort" manner in cloud offerings [31, 45], we aim to minimize the tail latency of serverless functions.*

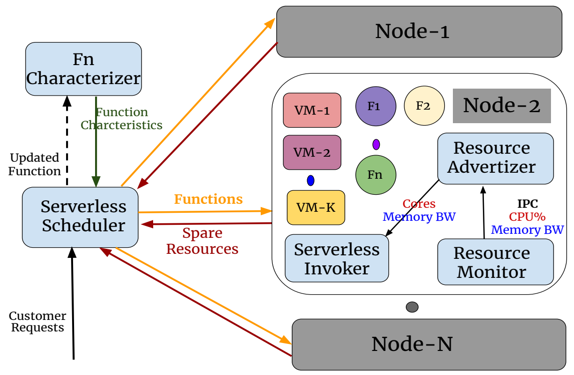
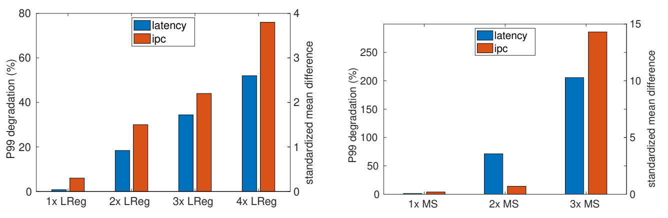


Figure 4: Illustration of the ServerMore design. We show the ServerMore components inside one server/node (Node-2) for ease of presentation. VM-i and Fj represent customer VMs and serverless functions, respectively.

The takeaways from Section 3 suggest that there is an opportunity to colocate functions with latency-sensitive serverful applications by careful regulation of *multiple resources*, but the candidate functions to colocate must be *dynamically selected* depending on the serverful workload and latency. In practice, another challenge that we must overcome is that *customer VMs are black-box in nature*, and thus the provider cannot (or should not) characterize the latency of the application running inside the customer VM [22, 30, 33].

Overview of ServerMore: The design of our solution, ServerMore, is guided by the need to address the above challenges and the specific constraints and opportunities intrinsic to serverful and serverless workloads. Figure 4 shows the different components (in blue) of ServerMore. At a high-level, a Resource Monitor within a server (or node) tracks



(a) Colocation with LLC-Reg (b) Colocation with Memory-Stream

Figure 5: Illustrating the correlation between latency degradation and the standardized mean difference of IPC (ratio of mean difference to standard deviation) of serverful workload when colocated with LLC-Reg and Memory-Stream.

the resource usage of serverful workload(s). Based on the usage and server capacity, the Resource Advertizer computes the spare capacity that can be employed by serverless functions. The Serverless Scheduler *selectively admits* incoming customer serverless functions for colocation based on the advertized capacity and based on the usage characterization of individual functions (collected via the Fn-Characterizer). We now discuss the design decisions for these components in the following subsections.

4.1 Inferring the application performance of black-box customer VMs

A natural approach to avoid performance degradation for colocated serverful workloads is to immediately react to high tail latencies. While prior work on colocation assumed that application latency is visible to the provider [4, 12, 27], this is not the case for customer VMs in a public cloud.

While application performance within the VM cannot be monitored by the cloud provider, the resource usage and hardware performance counters related to the VMs can be easily tracked by the provider via the hypervisor and OS. In ServerMore, the Resource Monitor tracks the *Instructions Per Cycle* (IPC) counter of the customer VM and uses this as a *proxy* for the performance of the VM-deployed application. While not perfect, IPC has been shown to be well correlated with application performance [22], especially for CPU-bound workloads [23, 52].

For non–CPU-bound workloads, we find that the raw IPC value is not as well correlated with latency because of the diminished change in IPC for a corresponding change in latency. Instead, we find that the standardized mean difference of IPC, smd_{ipc} (ratio of mean difference to standard deviation) is much better correlated with the latency degradation due to colocation. Figure 5 highlights the correlation between the P99 degradation for TF-Serving and smd_{ipc} when colocated with LLC-Reg and Memory-Stream. *smd* is a statistic, often referred to as Cohen's *d* statistic, that is commonly employed when measuring the effect size between two means [5, Chapter 2]. An advantage of employing *smd* is that it is easy to adapt to changes in the serverful workload by updating the mean and standard deviation of its IPC (see Section 4.3).

4.2 Characterizing serverless functions

Takeaways 1 and 3 suggest that different serverless functions can naturally induce different types and intensities of interference. To account for these differences and safely colocate functions with customer VMs, it is necessary to *characterize* the incoming functions. Unlike black-box serverful VMs, the latency (and resource usage) of serverless functions can be characterized by cloud providers since the function's performance is visible. Major serverless providers, including AWS Lambda [39], already offer tools to monitor and characterize serverless functions.

We use `docker stats` [8] and Intel's `PQoS` [19] to characterize serverless functions, specifically determine their (i) CPU utilization, (ii) LLC sensitivity, and (iii) memory bandwidth requirements. For a newly created/modified serverless function, the Serverless Scheduler runs the first new invocation in a dedicated server, shown as Fn-Characterizer in Figure 4, and returns the result to the customer. Subsequent invocations need not be actively characterized and are considered for colocation. If an already characterized function exhibits significantly different characteristics at runtime, such as a much higher total CPU usage, it will re-characterized at the next invocation.

LLC-sensitivity: We define a function to be LLC-sensitive if a substantial portion of its working set size fits in the LLC. To determine whether a serverless function is LLC sensitive, we run one offline invocation of the function on the Fn-Characterizer by allotting a single cache-way to it (via Intel's CAT [17]). If the function is LLC sensitive, reducing the available cache will result in an increase in its utilized memory bandwidth The above sensitivity test can result in false positives (functions that are not LLC sensitive but flagged as such). However, a false positive will only impact resource utilization and not create undue LLC contention; we find this to be an acceptable tradeoff.

Classifying functions for CPU colocation: Motivated by Takeaway 1, we aim to only share serverful cores with *lightweight* serverless functions. Based on our analysis in Section 3, we classify a function as *lightweight* if it is non–LLC-sensitive and has CPU utilization of less than 25%.

4.3 Dynamic resource management for safely colocating serverless functions

To safely colocate serverless functions next to serverful workloads to improve server utilization, ServerMore dynamically regulates the resource usage of multiple resources (see Takeaways 1 and 2), as discussed next.

Algorithm 1 Computing spare CPU capacity for serverless

```
1:  while true do
2:      $shared\_cores \leftarrow 0.0$
3:      foreach $cur\_vm \in VMs$ do
4:          $cur\_ipc, cur\_cpu\_util \leftarrow read\_metrics(cur\_vm)$
5:          $\mu_{ipc}, \sigma_{ipc} \leftarrow update\_metrics(cur\_ipc, window\_size)$
6:
7:          foreach $core \in cur\_vm$ do
8:              $\mu_{cpu}, \sigma_{cpu} \leftarrow update\_metrics(cur\_cpu\_util)$
9:              $vm\_cpu\_limit \leftarrow \mu_{cpu} + \sigma_{cpu} + buffer_{cpu}$
10:             $max_{cpu} \leftarrow 100 - vm\_cpu\_limit$
11:             $min_{cpu} \leftarrow 0.5 \times max_{cpu}$
12:
13:             $smd_{ipc} \leftarrow max\{0, min\{1, (\mu_{ipc} - cur\_ipc)/\sigma_{ipc}\}\}$
14:             $ipc\_factor \leftarrow 1 - (smd_{ipc}/c_{cpu\_ipc})$
15:             $cpu\_alloc \leftarrow ipc\_factor \times (max_{cpu} - min_{cpu})$
16:             $cpu\_alloc \leftarrow cpu\_alloc + min_{cpu}$
17:
18:             $shared\_cores \leftarrow shared\_cores + cpu\_alloc$
19:     $spare\_cpu \leftarrow shared\_cores + exclusive\_cores$
20:     $sleep(metric\_sample\_period)$
21: end
```

4.3.1 Sharing CPU between serverful and serverless. ServerMore's CPU resource management logic, which runs periodically (once every second in our implementation), is shown in Algorithm 1. We first determine the amount of resource capacity on the server that should be reserved for serverful; the remaining capacity can then be spared for serverless. All cores on a server not allocated to customer VMs are reserved for serverless, and referred to as *exclusive cores*. The remaining cores, referred to as *shared cores*, are primarily allocated to customer VMs, but their spare capacity can be used to host additional serverless functions, thereby aggressively increasing server utilization.

Since the serverful workload's behavior can change over time, we track variations in its resource usage. The Resource Monitor component inside the server captures short-term behavior by computing the standard deviation (σ_{cpu}) of serverful's resource usage and long-term behavior by computing the mean (μ_{cpu}) usage; both are computed over a moving window (lines 4, 8 in Algorithm 1). We then reserve $\mu_{cpu} + \sigma_{cpu} + buffer_{cpu}$ capacity for serverful, where $buffer_{cpu}$ capacity is used for handling abrupt bursts in serverful workload (line 9). The unreserved capacity and exclusive cores are then advertized (lines 18–19) to Serverless Scheduler for hosting serverless functions; we discuss the Serverless Scheduler in Section 4.4.

Ideally, serverless functions should be able to utilize the advertized capacity without impacting serverful performance.

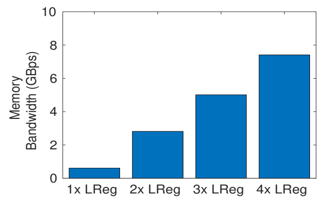


Figure 6: Super-linear increase in memory bandwidth usage of LLC-Reg.

However, resource partitioning is not perfect; e.g., L1 and L2 caches for shared cores can be under contention between serverless and serverful. To safeguard against such instances, we rely on feedback from the black-box serverful VM in the form of IPC. Specifically, we compute the standardized mean difference of IPC (smd_{ipc}), which is the difference in (moving-window-based) average IPC and currently monitored IPC divided by the standard deviation of IPC. A large (positive) value of smd_{ipc} indicates a statistically significant drop in serverful IPC, suggesting latency degradation. In such cases, we shrink the advertized capacity (lines 13–16) by a factor of $(1 - smd_{ipc}/c_{cpu\_ipc})$, where c_{cpu_ipc} is a parameter.

4.3.2 Sharing CPU between serverless functions. Since the advertized capacity may be sufficient to host multiple serverless functions, there can be contention *among* functions as well. Existing serverless platforms typically allocate a *fixed* amount of CPU, proportional to the memory requested by a function [49]. In ServerMore, for serverless functions, we use `cpu-shares` (a CPU allocation/isolation mechanism) from Linux cgroups to allocate CPU proportional to the memory requested as $allot\_cpu = (requested\_memory/1024)$.

In contrast to the existing practice of providing a fixed amount of CPU, the choice of `cpu-shares` is beneficial to functions since `cpu-shares` is a *soft limit* and is applicable only when there is CPU contention. Thus, by using `cpu-shares`, ServerMore improves the latency of functions in the average case (when not every colocated function is competing for CPU) while ensuring that the worst case performance (when all functions are competing for CPU) is similar to that under existing practices.

4.3.3 Sharing memory bandwidth. Given the challenges and limitations (see Section 3.3.2) in employing MBA to regulate memory bandwidth interference, we instead rely on feedback from the serverful workload to regulate the sharing of memory bandwidth between serverful and serverless. As with CPU, we first determine the bandwidth that should be reserved for serverful, and then advertize the rest. Our memory bandwidth regulation logic is similar to Algorithm 1 and is thus omitted.

4.3.4 Sharing LLC. In ServerMore, we fairly partition the LLC between serverful and serverless, allocating cache-ways among them in proportion to the number of cores allotted to the VMs and the exclusive cores allotted to serverless. Sharing LLC *between* multiple serverless functions is non-trivial since the mode of LLC partitioning is to devote exclusive cache-ways and there can be many more functions than there are cache-ways. Another issue with LLC sharing is that if insufficient LLC is provided to an LLC-sensitive function, its memory bandwidth requirement can sharply increase, impacting other workloads (see Section 3.2.2). Figure 6 shows the super-linear increase in memory bandwidth usage by the LLC-Reg function colocated with TF-Serving as we linearly increase the number of simultaneous copies of LLC-Reg. To address these issues, in ServerMore, we allow at most one LLC-sensitive function to be colocated with serverful workloads. There is no restriction on the number of colocated non–LLC-sensitive functions as they are, by definition, not impacted by LLC contention.

4.4 Selectively admitting serverless functions

The Serverless Scheduler reads in the advertized spare capacity and uses the function characterizations (from Section 4.2) to determine whether an incoming function can be colocated in the spare capacity. Specifically, for a server, let *spare_cpu* and *spare_mem* denote the spare CPU and memory bandwidth, respectively, that can be used by serverless (see *spare_cpu* in Algorithm 1 for reference). Further, let *spare_llc* denote the number of additional LLC-sensitive functions that can be accommodated on that server; note that $spare\_llc \in \{0, 1\}$. Then, if an incoming function has characterized CPU and memory bandwidth usage less than *spare_cpu* and *spare_mem*, respectively, the Serverless Scheduler will admit the function for colocated execution on that server if the function is not LLC-sensitive or if $spare\_llc = 1$. For colocation on shared cores, only *lightweight* functions are considered (see Section 4.2). If the function is admitted for colocation, the *spare_cpu*, *spare_mem*, and *spare_llc* variables of that server are decremented accordingly and re-advertized for admitting future functions. Note that serverless functions can experience coldstarts [44] in our setup. While the Serverless Scheduler can be modified to only admit functions that have a warm container, we choose not to employ this restrictive selection given our focus on improving resource utilization.

4.5 Sensitivity analysis for algorithm parameters

To determine the parameter values for our resource regulation algorithms, we conduct sensitivity analysis using the applications from Section 3 in addition to Data-Serving, Web-search, and Web-serving applications from the CloudSuite benchmark suite [10]. We note that our evaluation results in Section 5 do *not* employ these "training" workloads.

Function	Class	Runtime	CPU
Image Resizing (IR)	LW	0.48 s	8.6%
Email Gen (EG)	LW	0.24 s	12%
Stock Analysis (ST)	LW	0.78 s	15%
File Encrypt (FE)	LW	0.71 s	14%
Sentiment-Review (SR)	LW	0.37s	18%
Nearest-Neighbors (NN)	HW (C)	4.4 s	68.5%
Rodinia CFD (CFD)	HW (C)	37.1 s	88.3%
Sorting (SO)	HW (C, M)	11.2 s	90%
Dot-Product (DP)	HW (C, L)	48.2 s	100%
Structured-Grid (SG)	HW (C, M)	50.9 s	100%

Table 1: Serverless functions used in our evaluation and sensitivity analysis. For non-lightweight (or heavyweight, HW) functions, we indicate the dominant resource (C: CPU, L: LLC, M: memory bandwidth) in parentheses.

For CPU regulation, we colocate the serverless functions with serverful applications and measure the latency impact for c_{cpu_ipc} = 0.5, 1.0, 1.5, 2.0; *window_size* = 5, 10, 20, 30 seconds; and $buffer_{cpu}$ = 10%, 15%, 20%, 30%. Based on our experiments, we choose c_{cpu_ipc} = 1.0, *window_size* = 10s, and $buffer_{cpu}$ = 20% as these values provided the most spare CPU while minimizing the impact on serverful latency. Similarly, for memory bandwidth regulation, we set c_{mbw_ipc} = 0.5, *window_size* = 10s. Among the parameters, the *window_size* and c_{cpu_ipc} are largely insensitive, while $buffer_{cpu}$ and c_{mbw_ipc} are generally sensitive to the configured values. The values for the latter are chosen conservatively to prefer low latency at the expense of lower resource utilization.

5 EVALUATION RESULTS

This section presents our experimental evaluation of ServerMore, including comparisons with existing approaches. We start by discussing our experimental methodology. We then present results for colocation with a single serverful application and multiple serverful applications. We end this section with an ablation study to analyze the significance of each resource regulation design in ServerMore.

5.1 Experimental methodology

In this section, we describe our experimental setup, ServerMore implementation, and the applications, baselines, and metrics used for evaluation.

5.1.1 Testbed. We evaluate ServerMore on a cluster of 6 nodes each equipped with an Intel Xeon Silver 4114 CPU (with 10 physical cores in all), 192 GB memory, 14 MB of

LLC (11 cache-ways), and 50 GB/s memory bandwidth. To focus our results on the impact of interference, we colocate serverful and serverless applications on the same socket. We use a single node to host the VMs (for serverful workloads) and the containers (for serverless functions). We use Linux KVM to deploy the VMs and Apache OpenWhisk [47] to deploy the serverless functions. The client for both serverful and serverless workloads are hosted on a single node. Of the remaining four nodes, we use one to host the OpenWhisk controller, and the rest to co-host Ceph file system and Redis database to emulate distributed services required for serverless functions to read and write data.

5.1.2 Implementation of ServerMore. ServerMore is implemented via a user-space daemon and modifications to Apache OpenWhisk [47]. The user-space daemon performs the role of Resource Advertizer (see Figure 4) and is implemented in ~500 lines of Python code. The daemon periodically (once every second) samples the monitoring metrics (IPC, CPU utilization, memory bandwidth) and recomputes the spare resources that can be advertized for serverless.

We use `mpstat` to measure CPU usage and Intel `PQoS` [19] to measure memory bandwidth usage. We modify the `PQoS` tool to monitor the hypervisor processes to obtain the IPC for each running VM. The advertized resources are read by the Serverless Invoker (node-level serverless resource manager), which passes this information to the Serverless Scheduler. The Serverless Scheduler in turn uses the updated resource offerings to make function admission decisions. Both Serverless Invoker and Scheduler are implemented as modifications to Apache OpenWhisk, which required ~1200 lines of Scala code. The user-space daemon, and the performance monitoring, together consume less than 1% of CPU.

5.1.3 Applications and traces. The serverful applications we use for evaluation are: (i) ***Moses***: a state-of-the-art real-time machine translation system, (ii) ***Xapian***: an open-source search engine, (iii) ***Sphinx***: a speech recognition system, and (iv) ***Memcached***: a widely used in-memory key-value store [29]. All applications, except Memcached, are from the TailBench [24] benchmark suite. These applications have a wide variety of tail latency (P99) ranges, allowing us to comprehensively evaluate our solution: $< 1ms$ for Memcached; 8–12ms for Moses and Xapian; and 2–4s for Sphinx.

We use real world traces, scaled to our testbed, where applicable. For Moses and Xapian, we use two NLANR [32] traces—*Large Variations (LV)* and *Dual Phase (DP)*—shown in Figure 7. For Memcached, we use the mutilate tool to generate Facebook's *ETC* trace [2] (with an average of 10K requests/sec). Sphinx is a CPU-intensive workload and has relatively higher latency compared to other severful workloads; we thus use a constant load of 1 query per second (QPS) to drive Sphinx.

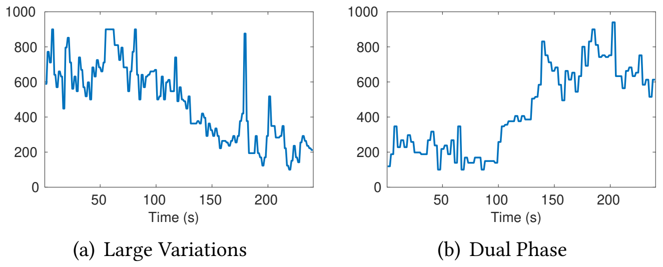


Figure 7: Traces used to drive the Moses and Xapian serverful applications, based on NLANR traces [32]. The y-axis is in units of requests per second (RPS).

For serverless, we use functions from publicly available sources, including FunctionBench [25] and SeBS [6]; details of these functions are presented in Table 1. We note that these functions are different from the "training" functions used in our motivation study (Section 3) and sensitivity analysis (Section 4.5). We use an open-loop load generator with an arrival rate of 4 requests/sec. For every request, the load generator randomly chooses from among the function pool listed in Table 1. We vary the function-mix by controlling the ratio of lightweight to heavyweight functions and consider three scenarios: Low (30% heavyweight), Medium (40% heavyweight), and High (50% heavyweight).

5.1.4 Metrics. Our objective is to improve the server's resource utilization while minimizing the performance impact on customer workloads. To measure the latency impact, we monitor the P99 latency of applications. We target a *maximum P99 degradation of 10% for serverful workloads*, similar to recent works on colocation [50]. Since serverless offerings are "best-effort" in practice [31, 45], we do not set a P99 target and instead attempt to minimize the P99 degradation of functions.

For improvement in resource utilization, we focus on serverless functions' usage since ServerMore aims to improve server utilization by colocating functions with serverful applications. The primary mode of colocation in ServerMore is to allocate CPU to functions. We thus report the *time-averaged number of cores used by serverless functions* as a proxy for resource usage improvement. We run each experiment 3 times and report the average of the metric across the runs, similar to [50].

5.1.5 Baseline. We experimentally compare ServerMore to two baseline black-box resource managers (RM) proposed in recent works:

(1) ***PerfIso*** [21]: is a black-box RM which colocates latency-sensitive serverful jobs with batch jobs. *PerfIso* improves resource utilization by running batch jobs in the unallocated cores of a server and maintains a fixed number of buffer cores to limit interference. To emulate *PerfIso*, we maintain a number of buffer cores and tune this number during evaluation. While *PerfIso* does not provide LLC partitioning or

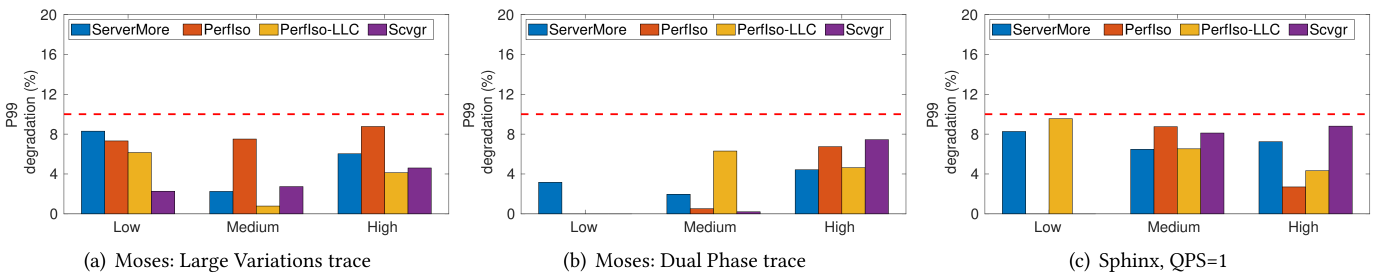


Figure 8: P99 latency degradation of Moses and Sphinx applications when colocated with different serverless mixes.

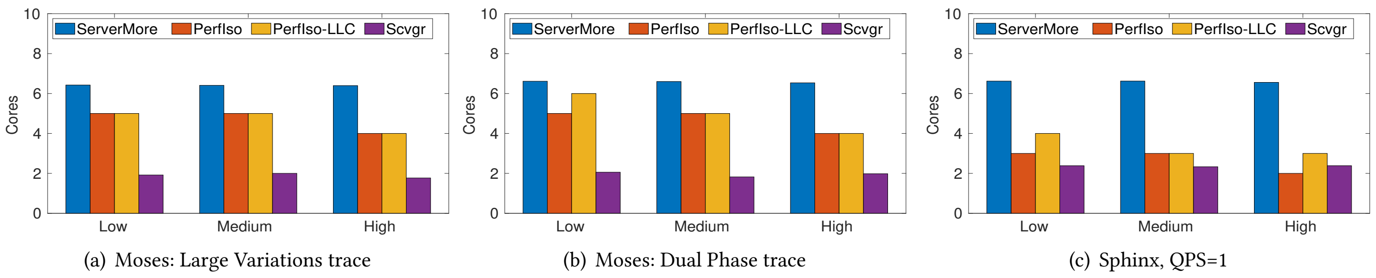


Figure 9: Number of cores used by serverless functions upon colocation with Moses and Sphinx serverful applications.

memory bandwidth regulation, we additionally implement a version of *PerfIso* with LLC partitioning, which we refer to as $PerfIso_{LLC}$.
(2) ***Scavenger*** [22]: A black-box RM which monitors the raw IPC value of the latency-sensitive VM and aggressively throttles the colocated provider batch-job containers to minimize the VM's performance degradation. *Scavenger* does not use cache partitioning or memory bandwidth regulation; however, *Scavenger* does indirectly regulate the cache pressure asserted by batch jobs via *cpu-quota*. To emulate *Scavenger*, we modify OpenWhisk to use *cpu-quota*.

Both baselines only leverage dedicated cores that are not allocated to the customer VMs for hosting batch jobs.

5.2 Colocation with a single VM

In this subsection, we consider colocation with a single serverful application running in a VM with 4 vCPUs, 16 GB of memory, and 5 exclusive cache-lines, with the vCPUs pinned to four physical cores. On the same socket, the remaining 6 cores and 6 cache-lines are exclusive to the serverless functions.

Moses under Large Variations: We begin our evaluation results with colocation of different serverless request mixes with Moses, which is a memory-bandwidth-intensive serverful application. We drive the Moses workload via the Large Variations (LV) trace shown in Figure 7(a). The LV trace is a challenging real-world trace with a wide range of RPS (100 to 900 req/s) and abrupt spikes, with an average

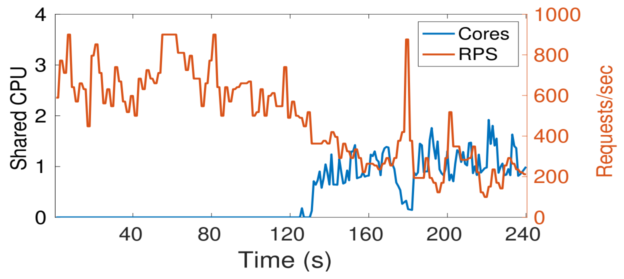
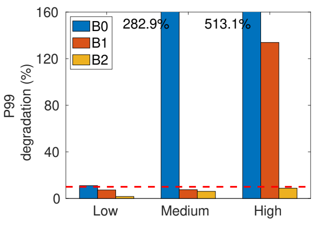


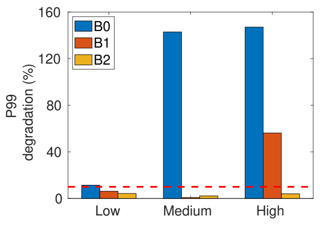
Figure 10: Timeline of shared cores (left y-axis) advertised by ServerMore for serverless function execution upon colocation with Moses under the load of Large Variations trace (right y-axis).

load of 486 RPS. The serverful P99 latency degradation and the serverless core usage of different resource managers are shown in Figure 8(a) and Figure 9(a), respectively. *For all serverless function mixes, ServerMore maintains the P99 degradation well below the 10% threshold and is able to utilize, on average, 6.41 cores for executing serverless functions.*

In Figure 10, we show the timeline of the shared cores (in blue) advertised by ServerMore for serverless function execution under the Large Variations trace (in orange). In the first half of the trace, we see that ServerMore does not advertise any shared cores since the serverful load is high. However, as the serverful load decreases in the second half, ServerMore responds by offering shared cores for serverless execution, thus aiming to improve resource utilization. The figure also highlights the responsiveness of ServerMore as it adapts to a changes in serverful load in a timely manner, thereby minimizing the impact on serverful latency.



(a) $PerfIso$



(b) $PerfIso_{LLC}$

Figure 11: P99 latency degradation for different buffer core sizes (*B*) of $PerfIso$ and $PerfIso_{LLC}$ when colocated with Moses and driven by Large Variations trace.

For $PerfIso$ and $PerfIso_{LLC}$ we experiment with the number of buffer cores (starting from 0) and determine the minimum number of buffer cores, for each function-mix, for which the P99 degradation is below 10%. We show the P99 degradation for different buffer core values (*B*) in Figure 11. For 0 buffer cores (i.e., using 6 cores to run serverless functions), the serverful P99 under *Medium* and *High* function mixes can be exceedingly high, with the P99 degradation being as high as 513% for $PerfIso$ under *High* mix. By increasing the buffer cores to 1, the performance of both $PerfIso$ and $PerfIso_{LLC}$ improves, but for the *High* function mix the P99 degradation (133% and 56%, respectively) is still much higher than the 10% threshold. Only with 2 buffer cores do $PerfIso$ and $PerfIso_{LLC}$ provide safe colocation for the *High* mix. The P99 degradation under $PerfIso$ for 0 buffer cores highlights the role played by the serverless function mix. As we move from *Low* to *High*, more *heavyweight* functions are admitted, making colocation *increasingly challenging*.

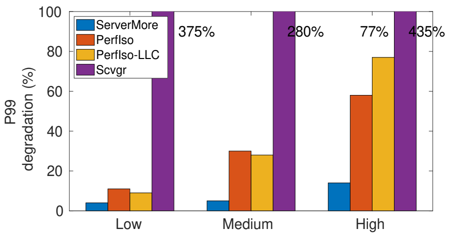
For colocation with Moses, both $PerfIso$ and $PerfIso_{LLC}$ utilize about 4.67 cores for serverless functions across function mixes. Under the *High* mix, they only utilize 4 cores for serverless, compared to the 6.4 core usage for ServerMore. Across all function mixes, ServerMore provides **37.3% more cores** to serverless compared to $PerfIso$ and $PerfIso_{LLC}$. *Scavenger*, with its aggressively resource throttling, imposes only 2–4% P99 degradation for serverful, but only manages to utilize 1.8–2 cores for serverless under different function mixes. Across all mixes, ServerMore provides ***230% more cores*** for serverless compared to *Scavenger*.

In Figure 12(a) and 13(a), we respectively show the P99 of the serverless functions and the percentage of the serverless functions that were admitted, when colocated with Moses (Large Variations trace). By carefully *characterizing and selectively admitting functions*, ServerMore is able to host many serverless functions. Further, by managing the interference *between* functions, ServerMore imposes ***minimal P99 degradation on accepted functions.*** By comparison, the baselines do not focus on protecting the performance of the colocated functions, resulting in much fewer (often at least 50% fewer than ServerMore) accepted functions and much higher latency degradation for the accepted functions. In fact, under *Scavenger*, the serverless functions face significant P99 degradation (280%–435%).

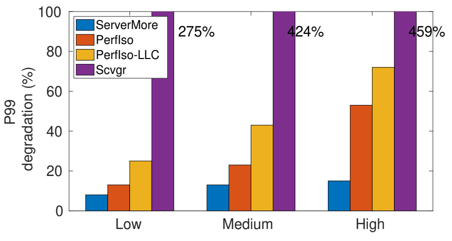
Moses under Dual Phase: Figures 8(b) and 9(b) show the serverful P99 degradation cores utilized for serverless functions by different resource managers when colocating with Moses driven by the Dual Phase (DP) trace (shown in figure 7(b)). The DP trace ranges from 100 to 900 RPS with an average of 440 RPS; in contrast to the LV trace, there is a noticeable phase change in DP as it moved from low to high RPS. We see that *ServerMore has low P99 degradation, ranging from 2% to 4.4% for different mixes, and on an average utilizes 6.58 cores for hosting serverless functions.* $PerfIso$ and $PerfIso_{LLC}$ both require at least one buffer core to meet the 10% P99 degradation threshold. Consequently, they have lower core utilization (by 29.1% and 24%, respectively) compared to ServerMore. *Scavenger* continues to have low impact on the serverful P99 degradation (0.5%–7.5%), but only utilizes 1.95 cores, on average, across functions mixes; this serverless core usage is 70% lower than that under ServerMore. As with the LV trace, the serverless P99 degradation (Figure 12(b)) and percentage of functions admitted (Figure 13(b)) continue to be significantly better for ServerMore than the baselines.

Sphinx: Figure 8(c) and 9(c) show the serverful P99 degradation and cores utilized for serverless functions, respectively, when colocating serverless with the memory-bandwidth-intensive serverful Sphinx application. ServerMore continues to perform well with serverful P99 degradation in the 6.5%–8.2% range and serverless core usage of about 6.6, on average, across function mixes. *Scavenger* continues to have lower P99 degrdation (maximum of 8.8%) but with a low serverless core utilization of only 2.36. Both $PerfIso$ and $PerfIso_{LLC}$ require a larger buffer of at least 3 cores (except in one case) to meet the P99 degradation threshold of 10%. In terms of serverless, *ServerMore provides, on average across functions mixes, 147%, 98%, and 179% more core utilization than $PerfIso$, $PerfIso_{LLC}$, and Scavenger, respectively.* The serverless P99 degradation (Figure 12(c)) and percentage of functions admitted (Figure 13(c)) continue to be significantly better for ServerMore than the baselines.

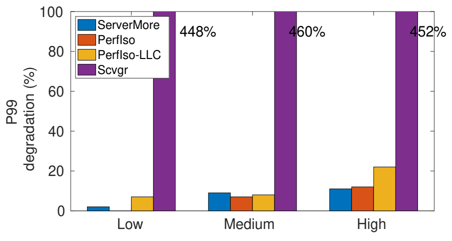
A practical problem with the $PerfIso$ approach of maintaining a buffer is that the *number of buffer cores needed to meet the P99 threshold varies with the application.* In contrast, by assessing the performance degradation of serverful (via smd_{ipc}) and *dynamically advertizing spare capacity for*



(a) Moses: Large Variations trace

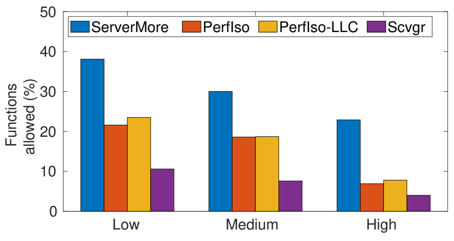


(b) Moses: Dual Phase trace

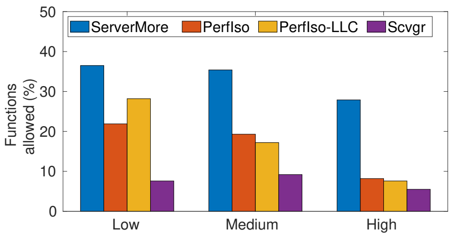


(c) Sphinx, QPS=1

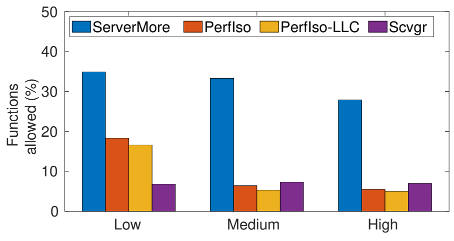
Figure 12: P99 latency degradation of serverless functions when colocated with Moses and Sphinx serverful applications.



(a) Moses: Large Variations trace



(b) Moses: Dual Phase trace



(c) Sphinx, QPS=1

Figure 13: Percentage of serverless functions accepted upon colocation with Moses and Sphinx serverful applications.

multiple resources to serverless, ServerMore is able to automatically *adapt to variations* and phase changes in the serverful workload.

Xapian and Memcached: In Figures 14 and 15, we respectively show the P99 degradation and serverless core usage of different resource managers when colocating functions with serverful Xapian and Memcached applications. Both Xapian and Memcached, in the load regime we consider, are not memory-bandwidth-intensive. As before, all resource managers continue to meet the P99 threshold. ServerMore continues to provide significant core utilization improvement over *Scavenger*; the increase in core usage over *Scavenger* afforded by ServerMore is 249%, 273%, and 293%, respectively, for Xapian with LV trace, Xapian with DP trace, and Memcached.

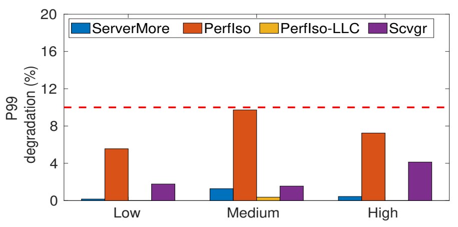
The core usage improvement of ServerMore over $PerfIso$ and $PerfIso_{LLC}$ is less pronounced since the serverful applications do not require any buffer cores for memory bandwidth protection. Nonetheless, by allowing serverless functions to safely execute on the serverful cores, ServerMore still provides a modest 9–15% core usage improvement over $PerfIso$ and $PerfIso_{LLC}$. The results of P99 degradation of serverless and the percentage of functions admitted show similar trends as Figure 15, and are thus omitted.

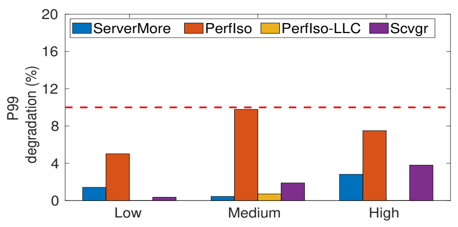
5.3 Colocation with multiple VMs

We now consider the scenario where two serverful applications are hosted on a server. Both serverful VMs are identical, with 4 vCPUs, 16 GB of memory, and 4 cache-lines each. We allocate 2 cores and 3 cache-lines exclusively to serverless.

Figures 16(a) and 16(b) respectively show the serverful P99 degradation and serverless core utilization when colocating functions with the co-hosted Moses and Xapian VMs; both serverful workloads are driven by the LV trace. Given the increased serverful load, *Scavenger*, with its aggressive throttling of serverless, results in negligible serverless utilization and minimal P99 serverful degradation. ServerMore, $PerfIso$, and $PerfIso_{LLC}$ continue to comply with the P99 degradation target. While $PerfIso$ and $PerfIso_{LLC}$ do not require any buffer cores, ServerMore still outperforms them (by about 33.9%) in terms of serverless core usage by allowing functions to safely execute intermittently on the eight serverful cores.

Figures 17(a) and 17(b) show our results when colocating functions with the co-hosted Moses and Sphinx VMs. Compared to the results when colocating with Moses and Xapian, $PerfIso$ cannot safely colocate functions without using buffer cores, resulting in serverless core utilization of only 1 core. $PerfIso_{LLC}$ does better than $PerfIso$, providing an average of 1.67 cores. By contrast, ServerMore manages to not only use both exclusive serverless cores, but also utilizes some of the spare capacity of serverful cores, resulting in an average utilization of 2.57 cores. *Scavenger* continues to perform poorly with minimal colocation.


(a) Xapian: Large Variations trace


(b) Xapian: Dual Phase trace

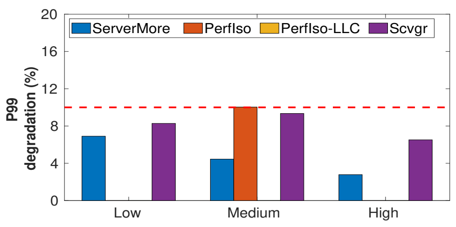
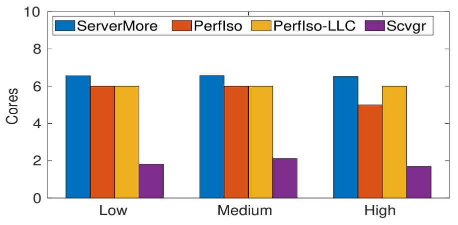
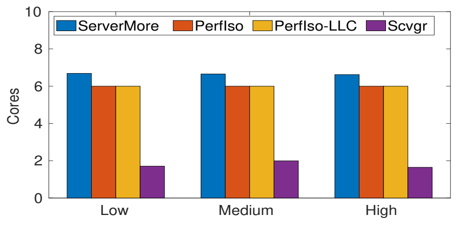

(c) Memcached, QPS=10K

Figure 14: P99 latency degradation of Xapian and Memcached applications when colocated with different serverless mixes.


(a) Xapian: Large Variations trace


(b) Xapian: Dual Phase trace

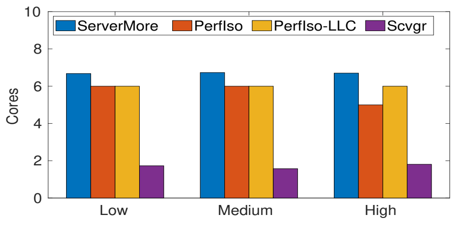
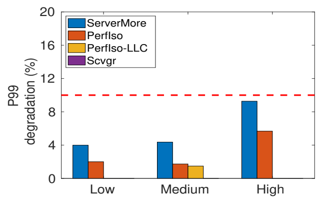

(c) Memcached, QPS=10K

Figure 15: Number of cores used by serverless functions upon colocation with Xapian and Memcached serverful applications.


(a) Serverful latency degradation

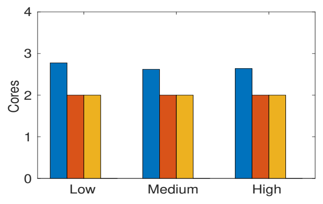
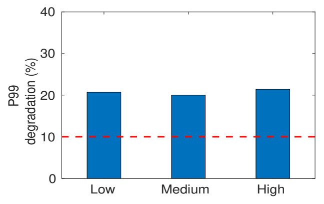

(b) Serverless cores used

Figure 16: Results of colocating serverless with Moses and Xapian (under LV trace) running on two 4-core VMs.


(a) LLC regulation disabled

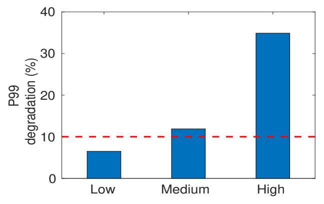
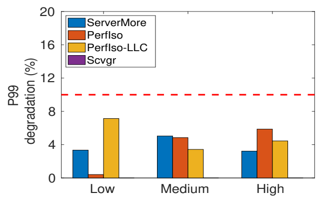

(b) Memory bandwidth regulation disabled

Figure 18: Results of colocating Moses (LV trace) with serverless functions under ServerMore with partially disabled resource regulation.


(a) Serverful latency degradation

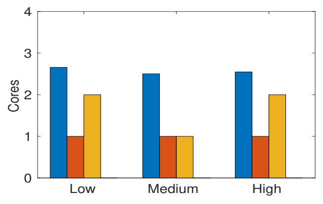

(b) Serverless cores used

Figure 17: Results of colocating serverless with Moses (LV trace) and Sphinx (QPS of 1) running on two 4-core VMs.

5.4 Significance of ServerMore's resource regulation

We now perform an ablation study to understand the importance of the resource regulation decisions made in the design of ServerMore. We colocate Moses (running the LV trace) with serverless functions and selectively disable a specific resource regulation. Figure 18(a) shows the P99 degradation for Moses when we disable LLC regulation for ServerMore. We see that the P99 degradation is consistently above the threshold, highlighting the importance of both partitioning the cache and checking for LLC-sensitive functions when admitting serverless load. Figure 18(b) shows the P99 degradation when we disable memory bandwidth regulation for ServerMore. This time, the threshold is violated for the more memory-bandwidth-intensive *Medium* and *High* function mixes, highlighting the necessity of memory bandwidth regulation.

6 ACKNOWLEDGMENT

We would like to thank our shepherd Sundar Dev, and other anonymous reviewers for their valuable feedback. We would also like to thank CloudLab [9] for providing the infrastructure to conduct the experiments. This work was supported by NSF grants CNS-1750109 and CNS-1717588.

REFERENCES

[1] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 735–751. https://www.usenix.org/conference/osdi20/presentation/ambati

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766

[3] AWS. 2021. AWS Samples. https://github.com/aws-samples.

[4] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[5] Jacob Cohen. 1988. *Statistical power analysis for the behavioral sciences*. Routledge.

[6] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2020. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. *CoRR* abs/2012.14132 (2020). arXiv:2012.14132 https://arxiv.org/abs/2012.14132

[7] 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* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[8] Docker. 2021. docker stats | Docker Documentaion. https://docs.docker.com/engine/reference/commandline/stats/.

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[10] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* 47, 4 (March 2012), 37–48. https://doi.org/10.1145/2248487.2150982

[11] 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

[12] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[13] Google Cloud Platform GCP. 2021. PerfKit Benchmarker. https://github.com/GoogleCloudPlatform/PerfKitBenchmarker.

[14] Andrey Goder, Alexey Spiridonov, and Yin Wang. 2015. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 459–471. https://www.usenix.org/conference/atc15/technical-session/presentation/goder

[15] Google Cloud Platform. 2021. Cloud Run. https://cloud.google.com/run.

[16] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 280–295. https://doi.org/10.1145/3423211.3425683

[17] Intel. 2016. Introduction to Cache Allocation Technology. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html.

[18] Intel. 2019. Introduction to Memory Bandwidth Allocation. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html.

[19] Intel. 2021. Intel RDT software package. https://github.com/intel/intel-cmt-cat.

[20] Intel. 2021. Intel Resource Director Technology Framework. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[21] 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

[22] 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* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 272–285. https://doi.org/10.1145/3357223.3362734

[23] Kostis Kaffes, Dragos Sbirlea, Yiyan Lin, David Lo, and Christos Kozyrakis. 2020. Leveraging Application Classes to Save Power in Highly-Utilized Data Centers. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Virtual Event, USA, 134–149.

[24] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581261

[25] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 477. https://doi.org/10.1145/3357223.3365439

[26] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 427–444.

[27] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*.

Association for Computing Machinery, New York, NY, USA, 450–462. https://doi.org/10.1145/2749469.2749475

[28] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 19–25.

[29] mediawiki.org. 2021. memcached. http://www.mediawiki.org/wiki/Memcached.

[30] Joydeep Mukherjee and Diwakar Krishnamurthy. 2020. PRIMA: Subscriber-Driven Interference Mitigation for Cloud Services. *IEEE Transactions on Network and Service Management* 17, 2 (2020), 958–971.

[31] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien. 2019. Real-Time Serverless: Enabling Application Performance Guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*. Davis, CA, USA, 1–6.

[32] NLANR. [n.d.]. National Laboratory for Applied Network Research. Anonymized access logs. ftp://ftp.ircache.net/Traces/.

[33] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. 2013. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of 2013 USENIX Annual Technical Conference (ATC '13)*. San Jose, CA, USA, 219–230.

[34] 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

[35] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[36] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[37] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (April 2021), 76–84. https://doi.org/10.1145/3406011

[38] Amazon Web Services. 2020. AWS Lambda now supports 10 GB of memory and 6 vCPU cores for Lambda functions. https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/.

[39] Amazon Web Services. 2021. Using Lambda Insights in Amazon Cloud Watch. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-insights.html.

[40] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[41] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless Linear Algebra. In *SoCC '20* (Virtual Event, USA). Association for Computing Machinery, New York, NY, USA, 281–295. https://doi.org/10.1145/3419111.3421287

[42] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2019. Archipelago: A Scalable Low-Latency Serverless Platform. *CoRR* abs/1911.09849 (2019). arXiv:1911.09849 http://arxiv.org/abs/1911.09849

[43] Amoghvarsha Suresh and Anshul Gandhi. 2019. FnSched: An Efficient Scheduler for Serverless Functions. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3366623.3368136

[44] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. 2020. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 1–10. https://doi.org/10.1109/ACSOS49614.2020.00020

[45] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Virtual Event, USA, 311–327.

[46] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *SoCC '20* (Virtual Event, USA). Association for Computing Machinery, New York, NY, USA, 311–327. https://doi.org/10.1145/3419111.3421306

[47] The Apache Software Foundation 2021. *Apache OpenWhisk*. The Apache Software Foundation. https://openwhisk.apache.org/

[48] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[49] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[50] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3447786.3456225

[51] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. 2018. DCat: Dynamic Cache Management for Efficient, Performance-Sensitive Infrastructure-as-a-Service. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 14, 13 pages. https://doi.org/10.1145/3190508.3190555

[52] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. Prague, Czech Republic, 379–391.