HORSE: Ultra-low latency workloads on FaaS platforms

Djob Barbe Thystere Mvondo Djob, Francois Taiani, Yérom-David Bromberg

▶ **To cite this version:**

Djob Barbe Thystere Mvondo Djob, Francois Taiani, Yérom-David Bromberg. HORSE: Ultra-low latency workloads on FaaS platforms. Middleware '24: 25th International Middleware Conference, ACM Association For Computing Machinery, Dec 2024, Hong Kong, Hong Kong SAR China. pp.445-453, 10.1145/3652892.3700784 . hal-04894549

HAL Id: hal-04894549
https://hal.science/hal-04894549v1

Submitted on 17 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Horse: Ultra-low latency workloads on FaaS platforms

Djob Mvondo
bmvondod@irisa.fr
Univ Rennes, CNRS, INRIA, IRISA
Rennes, France

Francois Taiani
francois.taiani@irisa.fr
Univ Rennes, CNRS, INRIA, IRISA
Rennes, France

Yerom-David Bromberg
david.bromberg@irisa.fr
Univ Rennes, CNRS, INRIA, IRISA
Rennes, France

ABSTRACT

We investigate if FaaS platforms can handle ultra-low latency workloads that run as low as less than $1\mu s$ and show that even for a warm start, the initialization time takes up to 99, 99% of the total execution time. This is due to the resume process of warm sandboxes that takes more time as the number of the sandbox's allocated virtual CPUs (vCPUs) increases. We uncover that two operations use up to 93, 1% of the resume time. The first is the insertion of the paused sandbox's vCPUs to a CPU-sorted run queue. The second is the update of a lock-protected variable, which represents the vCPUs' load on each CPU. This variable is used for frequency scaling.

We introduce Horse, for hot resume. Horse presents two simple approaches. The first is parallel precomputed sorted merge ($\mathcal{P}^2\mathcal{SM}$), a parallel algorithm that leverages pre-computed data to have a parallel sorted merge of two sorted lists in $O(1)$. The second is to coalesce the updates on the lock-protected variable used for frequency scaling. We implement Horse in Xen and Firecracker, two mainstream virtualization platforms. Our evaluation with real-world FaaS traces shows that Horse achieves up to 7, 16× resume time improvement and reduces sandbox initialization overhead by up to 142, 84× with no impact on functions.

ACM Reference Format:
Djob Mvondo, Francois Taiani, and Yerom-David Bromberg. 2025. **Horse: Ultra-low latency workloads on FaaS platforms. In . ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn**

1 INTRODUCTION

The quest for ultra-low latency services. Several cloud workloads run at very low latency in the order of dozens of μs [3, 5, 23, 24, 28, 30, 54, 55, 66] with workloads running under 1μs [31]. Examples of such services include NFVs [2, 76, 85], HPC tasks [37, 41, 61, 65], machine learning (ML) inference tasks [14], finance microservices [63, 83], and distributed in-memory key-value stores with small objects [24, 32, 33]. We denote such workloads as ultra-low latency (hereafter denoted as uLL). Dealing with uLL workloads has led to many efforts targeting multiple layers of the data center infrastructure to address such services correctly. Some of these works propose novel intra-server scheduling and network stack optimizations that enable uLL workloads to run as soon as they are triggered [27, 45, 47, 48, 57, 58, 62]. Other research works explore frequency scaling mechanisms to ensure that both energy usage and CPU frequencies are optimal for uLL workloads [6, 7, 17, 36, 43, 72, 84]. Lastly, some research works explore server interconnect network congestion and hardware configurations to enable nanosecond scale inter-server communications [16, 29, 31, 34, 35, 42, 87]. However, no previous work has considered what happens if the uLL workload runs in a sandbox environment such as a (micro-) virtual machine (VM). Concretely, if we consider that the data center network and the server scheduling policies are optimal enough to provide a nanosecond scale trigger, we ask the following question: *can a uLL workload meet its low latency requirements if triggered in a sandbox ?* This paper explores this question, considering that uLL workloads run atop a Function as a Service (FaaS) platform.

FaaS: Warm starts are not enough. In recent years, FaaS has become a popular Cloud computing model [22]. With FaaS, tenants ship functions in a specific format and configure events that will trigger the function's execution in a sandbox environment in the providers' Cloud. It allows users to focus on their business logic and leave the resource provisioning alongside management to the Cloud providers. Several major public and private Cloud have their FaaS platforms, such as Amazon Lambda [69], Azure Functions [19], Alibaba Function Compute [18], or Meta XFaaS [67]. A *cold start* occurs when a function must wait for a sandbox to be initialized. Because sandboxes can be microVMs (as within Amazon Lambda with Firecracker [1]) or containers hosted in virtual machines (as with Alibaba Function Compute [79]), they typically require a few milliseconds to boot, which is detrimental for uLL workloads. Several research works have proposed solutions to improve cold starts that leverage snapshots [8, 10, 78], memory deduplication [68], caching approaches for pipelines [15, 50, 64, 73, 80], and fork approaches [25, 82, 86]. However, for the best of the latter approaches, a cold start remains around $1ms$, leading to an overhead of up to 99, 99% for uLL workloads (Section 2).

A function benefits from *a warm start* when an initialized sandbox is already available to host it. This happens for one of the following two reasons. The first reason is that FaaS platforms implement a keep-alive strategy, which consists of keeping a sandbox active for a fixed time after the function that was running ends its execution [70, 71, 74, 79, 81]. The second reason is that users can subscribe to an option that ensures to always have a sandbox ready for their function. This option is proposed by Azure Premium Functions [11], Amazon Lambda Provisioned Concurrency [39], or Alibaba Provisioned Mode [49]. This is the fastest option for workloads that have very low latency requirements. To prevent useless CPU contention between idle warm sandboxes (waiting for a function) and running sandboxes (running a function) [51], these warm sandboxes are paused [9]. Unfortunately, resuming a sandbox can take up to 1, 1μs, which amounts to up to 61% overhead for uLL workloads (Section 2). Thus, existing designs for the reactivity of FaaS platforms induce severe overhead for uLL workloads.

Our approach: Horse. We propose Horse, which stands for hot resume, a fast path for FaaS platform to meet the stringent requirements of uLL workloads by improving the resume process of a warm sandbox. We analyze the resume process of sandboxes and

Conference'17, July 2017, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

uncover that up to 93, 1% is passed in two operations, which varies depending on the number of virtual CPUs (vCPUs) allocated to the sandbox (Section 3). The first operation is the sorted merge of each vCPU into a CPU-sorted run queue. The sorting parameter depends on the virtualization system and the scheduler algorithm used. The second operation is the update (after each latter sorted merge) of a global variable, which tracks the load of the resuming sandbox vCPUs on their corresponding run queue. This variable is leveraged for DVFS and thread load balancing on cores.

Horse cuts down by up to 69% the time required for the aforementioned operations by introducing two simple mechanisms. The first mechanism, denoted $\mathcal{P}^2\mathcal{SM}$ for parallel precomputed sorted merge, is an algorithm designed to perform a sorted merge of sorted lists. $\mathcal{P}^2\mathcal{SM}$ requires additional data structures, which allows a sorted merge in $O(1)$ (Section 4.1). We use $\mathcal{P}^2\mathcal{SM}$ to improve the sorted merge of each vCPU in a sorted run queue. Concretely, instead of performing a sorted merge of each vCPU sequentially, we merge the vCPU of a paused sandbox in a sorted way and when it is resumed, we apply $\mathcal{P}^2\mathcal{SM}$ to merge the sandbox sorted vCPUs list into the target run queue in $O(1)$. Additionally, Horse reserves a set of run queues for resuming uLL workloads sandboxes where each task has a maximum time slice of $1\mu s$ (Section 4.1.3).

The second mechanism is to coalesce all updates that will be achieved on a global variable to update it in one operation (Section 4.2). Concretely, instead of updating the DVFS-related global variable for each resumed sandbox vCPU inserted in a run queue, we perform one update that will represent all the changes that would have been applied. Horse mechanisms do not rely on specific CPU operations nor hardware accelerators, meaning that they can be applied by mainstream servers found in Cloud architectures.

Horse results. We implement Horse in Xen [13] and Firecracker [1], two mainstream virtualization systems that enable creating and managing sandboxes such as VMs and microVMs. Firecracker is also the virtualization system that powers AWS Lambda infrastructure. We implement Horse as a fast-path resume that can be used for resuming sandboxes that need to match uLL workload requirements. We evaluate our prototypes with micro-benchmarks and production FaaS traces from Azure serverless Cloud [12], leading to three key results. Firstly, compared to the existing resume algorithm, Horse improves warm sandboxes resume time by up to 7, 16× and sandbox initialization overhead for uLL workloads by up to 142, 84×. Second, the overhead in terms of CPU and memory is less than 1%. Lastly, when triggering uLL workloads and longer running workloads, there is a negligible overhead on longer running workloads that reaches 0.00107% on their 99th latency percentiles.

2 ANALYSIS OF FAAS PLATFORMS REGARDING ULL WORKLOADS

We consider the data center network stack fast enough to ensure the nanosecond-scale trigger of functions. Thus, the bottleneck shifts towards the initialization and readiness of the sandbox used by the FaaS platform. We perform the following evaluation to assess how this bottleneck affects uLL workload in FaaS platforms.

Evaluation scenario, platform, and server description. We trigger the execution of a uLL workload and measure the time required for the uLL workload to start its execution and the uLL workload execution time. Since we do not consider network communications overhead, we trigger the uLL workload on the same server node where it will run. We consider three modes for the FaaS platform during our evaluation. The first mode, denoted *cold*, is the scenario where creating a new sandbox is required before running the uLL workload. The second, denoted *warm*, refers to the scenario where an existing sandbox is reused to run the uLL workload. The last, denoted *restore*, refers to the scenario where a snapshot of the target sandbox is restored instead of creating a sandbox from scratch.

We use Firecracker $v1.3.3$ as our FaaS platform; thus, our sandbox is a microVM. We allocate 1 vCPU and 512 MB memory to the triggered microVM. We use FaaSnap [8] for the *restore* mode since it is one of the latest works that propose rapid snapshotting for sandboxes in the context of FaaS and targets microVMs. We use a Cloudlab r650 server [26] that has 2 CPUs Intel Xeon Platinium 8360Y with 36 cores at 2.40 GHz (hyperthreading disabled) running Ubuntu Server 22.04.2 LTS running a top of the Linux kernel v6.2.9.

Workload description. For each mode, we consider three categories of uLL workloads related to their execution times. We consider uLL workloads whose execution time is $\leq 20\mu s$ (Category 1), $\leq 1\mu s$ (Category 2), and hundreds of *ns* (Category 3). We chose these values as they englobe the different uLL workloads considered in previous related work that tackles low-latency workloads in datacenters [3, 5, 23, 24, 28, 30, 31, 54, 55, 66]. For Category 1, we implement a stateless firewall that takes a request header as input and determines whether the request should go through by querying a static allow list. For Category 2, we implement a NAT that changes a request header based on pre-registered routing rules. The two functions (firewall and NAT) are common use cases for NFVs. For Category 3, given an array composed of 3000 integers, they retrieve the indexes of all the elements in the array that are larger than an integer parameter passed during the workload trigger. Such operations are used during image transformation operations. We chose Node.JS as our language runtime for implementation since it is one of the most used languages on FaaS platforms [22].

Results. Table 1 presents for each uLL workload and FaaS platform scenario, the time required for making a sandbox ready to run the uLL function and the time the uLL workload takes to finish its processing. Additionally, Figure 1 presents the time required to get a sandbox ready as a percentage of the overall pipeline up to the end of the uLL workload execution. The *cold* and *restore* scenarios take up to 99% just in the sandbox preparation phase. Even for the *warm* scenario, getting the sandbox ready takes as low as 6% and up to 61% of the total execution pipeline.

The lowest latency is achieved by the warm start, which can be enforced as in public Cloud with their premium options such as Azure Premium Functions [11], Amazon Lambda Provisioned Concurrency [39], or Alibaba Provisioned Mode [49]. In the next section, we decompose the hot start mechanism to understand the major bottlenecks for uLL workloads.

3 ANALYSIS OF ULL WORKLOADS ON FAAS PLATFORMS

A warm start corresponds to the scenario when a FaaS platform reuses an initialized sandbox to host the requested function. To reduce the keep-alive tax, hot sandboxes are paused. Pausing hot

	Category 1			Category 2			Category 3		
	Cold	*Restore*	***Warm***	*Cold*	*Restore*	***Warm***	*Cold*	*Restore*	***Warm***
Initialization (μs)	$1{,}5\times10^6$	1300	**1,1**	$1{,}5\times10^6$	1300	**1,1**	$1{,}5\times10^6$	1300	**1,1**
Average Execution (μs)		17			1,5			0,7	
Init. Per. (%)	99,99	98,7	**6,07**	99,99	99,98	**42,3**	99,99	99,94	**61,1**

Table 1: Sandbox readiness latency and function execution time for each uLL workload and FaaS platform scenario. Even with the best scenario, *warm*, sandbox initialization takes up to $61,1\%$ of the total trigger process.

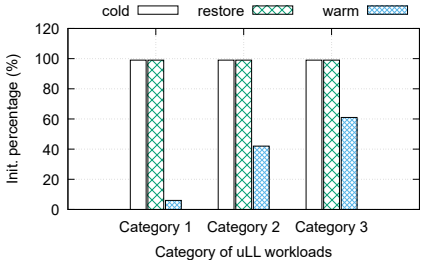

Figure 1: Percentage of time taken by the sandbox initialization for each scenario: cold, restore, and warm starts for 3 categories Category 1 ($\leq 20\mu s$), Category 2 ($\leq 1\mu s$), and Category 3 (100s of ns).

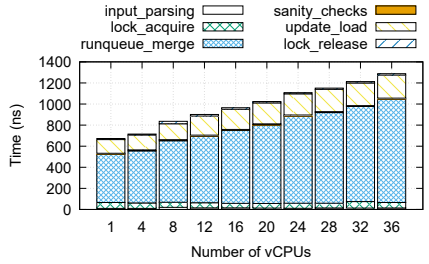

Figure 2: Breakdown of the resume process of a sandbox while varying the number of vCPUs allocated to the sandbox.

sandboxes prevents CPU contention between the hot sandboxes and the running sandboxes (a sandbox running a function) [9, 51]. The consequence of pausing a sandbox is that its virtual CPUs (vCPUs) are removed from the CPUs run queues. The hot sandbox is then resumed when a corresponding function can run inside.

3.1 Overview of a sandbox resume process

In this section, we unroll a paused sandbox's resume process before analyzing each step's cost. ① Firstly, the input parameters associated with the resume command are parsed and passed to the virtualization system if the parameters are correctly parsed. ② Upon reception of the input parameters, the virtualization system acquires a lock to prevent a parallel resume of another paused sandbox. ③ If the lock is successfully acquired, it then performs sanity checks are performed, such as checking if the target sandbox is in the pause state. ④ Once the sanity checks are successfully passed, for each vCPU associated with the target sandbox, the virtualization system finds a run queue to add the vCPU. Each run queue is sorted, and the attribute considered for the sort depends on the scheduling policy used. For example, with the `credit2` scheduler [60] in Xen, the run queues will be sorted based on their credit to have the process with the least remaining credit first in a run queue. Thus, the virtualization system performs a sorted merge of each vCPU to the target run queue. ⑤ For each vCPU, when added to a run queue, the virtualization system updates the run queue's load, a measure of processing performed by the tasks in that run queue that the virtualization system governor uses for frequency scaling. Several algorithms can be used to track the load of run queues, such as the per-entity load tracking (PELT) introduced in 2011 by Paul Turner [21, 77] (depending on the governor used) and is used in Xen and Linux KVM. Independently of the algorithm used, when placing a paused vCPU in a run queue, the update is always in the form $L(x) = \alpha x + \beta$, where x is the previous load and α, β are constants. ⑥ Upon inserting each vCPU into a run queue and the corresponding run queue load is updated, the virtualization system releases the lock and changes the state of the sandbox to running.

3.2 Resume cost breakdown

To understand the cost of each operation, we break down the resume of a paused sandbox to investigate the bottleneck of hot starts for uLL workloads. To achieve that, we manually trigger the pause and resume of the sandbox used in Section 2 and record the time taken by each during the resume process with Firecracker. Additionally, we vary the number of allocated vCPUs to the sandbox from 1 to 36. With up to 36 vCPUs, we cover and exceed all the configuration options FaaS Cloud providers provide. We do not vary the number of memory allocated to the sandbox, as the resume process does not involve the memory of the sandbox.

Figure 2 presents the breakdown of the resume process of a sandbox as we vary the number of allocated vCPUs. The first observation is that two operations amount from $87,5\%$ to $93,1\%$ of the resume process. These two operations correspond to the sorted merge of each vCPU into a run queue (step ④) and the load update of the corresponding run queue (step ⑤). Secondly, these operations contributions increase with the number of the sandbox vCPUs as the virtualization system iterates over the number of vCPUs of the target sandbox. We obtain similar observations when using the Xen virtualization system (we change the XenStore to an in-memory shared space to reduce userspace costs as proposed by LightVM [44]). Thus, we must reduce these two operations to make warm starts suitable for uLL workloads.

4 HORSE: HOT RESUME

Horse, which stands for Hot Resume, aims at providing a fast path for the warm start resume such that it meets uLL latency requirements. Horse introduces two techniques that target the longest steps during a sandbox resume (steps ④ and ⑤ in Section

Algorithm 1: $\mathcal{P}^2\mathcal{SM}$ merge

```
Data: A, B, posA, arrayB
Result: B as the sorted merged linked-list
threads ← Threads(len(posA), posA.keys) ; /* Allocate as
  many threads as len(posA) */
for (thread,key) in threads do
    tmp ← arrayB[key].next;
    arrayB[key].next ← posA[key].list_head;
    posA[key].list_tail.next ← tmp
```

3.1). Horse introduces $\mathcal{P}^2\mathcal{SM}$, a $O(1)$ algorithm for a sorted merge of the resuming sandbox into corresponding run queues and a coalescing mechanism to update the load on run queues faster than per run queue. The following sections discuss each technique and how we implement it in a virtualization system.

4.1 $\mathcal{P}^2\mathcal{SM}$: parallel precomputed sorted merge

$\mathcal{P}^2\mathcal{SM}$ is an algorithm that aims at merging a sorted linked list B to another linked list B in a sorted way by leveraging precomputed information on the lists to achieve a $O(1)$ time complexity. It targets step ④ of the resume process of a sandbox (Section 3.1). $\mathcal{P}^2\mathcal{SM}$ is a two-fold algorithm with a pre-computation and a merge phase.

4.1.1 Pre-computation phase. **Description.** To merge A in B, $\mathcal{P}^2\mathcal{SM}$ requires two additional data structures. The first is an array where the entry at each index is the address of the element of the linked list B at the same position as the index. We denote this array, *arrayB*. The second is a hashmap where the key is an integer, and the value is a linked list. Each key provides the position of the linked list (a subset of A) concerning B. We denote this data structure, *posA*. Merging these two data structures can occur when required.

Datastructures update complexity. Updating *arrayB* requires adding or removing an element from the array depending on the update on B. The update is performed in $O(1)$ since when an update is performed on B, the corresponding entry is *arrayB* is added or removed. Adding an element to A will require the computation of the new element's relative position regarding B and inserting the information to *posA*. In the worst case, computing the position takes $O(n)$ where n is the number of elements in B. Inserting an element in *posA* is done in $O(1)$ since it is either a linked list insertion or the creation of a one-element linked list. Thus, an insert update of *posA* takes $O(n)$. Deleting an element in *posA* requires the items of its corresponding linked list (associated with its key) to delete it. In the worst case, all elements of A will be in one linked list with the deleted element at the last position leading to a $O(m)$ delete complexity, where m is the number of elements in A.

4.1.2 Merge phase. **Description.** Algorithm 1 shows the merge phase of A in B, with the information obtained through the pre-computation phase. The merge process requires as many threads as the number of keys of *posA*. Each thread, $t_{i,i=1,m}$, works with a *posA* key value, key_{t_i} which represents an index. Each thread, $t_{i,i=1,m}$, performs two steps that affect B. The first step consists of changing the next pointer of the element in B at position key_{t_i} (obtained from *arrayB*) such that it points at the head of the linked list referenced by key_{t_i} in *posA*. The second step consists of changing the next pointer of the latter list to point to the previous next pointer of the element in B at position key_{t_i}. Due to the pre-computed elements, no mutual exclusion is required on B between each thread.

Complexity analysis. Each thread performs at most two operations, consisting of changing the value of the next pointer. Thus, the merge is performed in a constant time $O(1)$, independently of the size of linked lists to merge.

4.1.3 Implementing $\mathcal{P}^2\mathcal{SM}$ for sandbox's resume. We update the pause and resume path to implement $\mathcal{P}^2\mathcal{SM}$. To avoid iterating over the vCPUs of the paused sandbox, we create a data structure, `merge_vcpus` that is a sorted merge of all sandbox's vCPUs sorted with the same parameter used by the active scheduling algorithm. When the sandbox resumes, we can apply $\mathcal{P}^2\mathcal{SM}$ to perform a sorted merge between `merge_vcpus` and a run queue. However, the current resume process selects a run queue for each vCPU. Thus, applying $\mathcal{P}^2\mathcal{SM}$ would mean maintaining the two data structures (*arrayB* and *posA* - Section 4.1.1) required by $\mathcal{P}^2\mathcal{SM}$ for all run queues, which would be computationally expensive.

To cope with this problem, we reserve one run queue for running the uLL sandboxes. We denote this run queue `ull_runqueue`. Consequently, for each paused sandbox, we continuously update the data structures necessary for $\mathcal{P}^2\mathcal{SM}$ related to the sandbox `merge_vcpus` and `ull_runqueue`. The updates are performed each time `ull_runqueue` is updated. Thus, when a paused sandbox is resumed, we apply $\mathcal{P}^2\mathcal{SM}$ with `ull_runqueue` and the paused sandbox `merge_vcpus`. Each task on the `ull_runqueue` has a maximum timeslice of $1\mu s$. Since this run queue is reserved for running uLL sandboxes, $1\mu s$ provides every workload with enough CPU time to terminate its execution as soon as possible.

In the case of a high frequency of uLL workload triggers, we can increase the number of `ull_runqueue`. In this case, the target run queue for an uLL sandbox is chosen when pausing the sandbox. The choice of the associated run queue considers the number of paused sandboxes already associated with each `ull_runqueue` to perform load balancing.

Merge threads are given the highest priority to preempt any task on the run queue where it is scheduled. Since merge threads perform a single operation, they terminate rapidly; thus, their impact on potential preempted tasks is minimal.

4.2 Coaelscing load updates

To reduce step ⑤ of the resume process (Section 3), we propose a simple mechanism that aims at reducing the processing time of this step. With $\mathcal{P}^2\mathcal{SM}$, all the vCPUs of the resuming sandbox are placed on one run queue (Section 4.1.3). Consequently, the current resume process will iterate over each vCPU and update the load of the same run queue. The key idea with our coalescing approach is to find an analytic function that applies changes to the load that reflects the changes normally performed.

4.2.1 Design. Let's consider a function f that updates a value x with the formula: $f(x) = \alpha x + \beta$. Applying f, n times ($n \geq 1$) times results correspond to: $\alpha^n x + \beta \sum_{i=0}^{n-1} \alpha^i$. Furthermore, $\beta \sum_{i=0}^{k} \alpha^i$ is a well known geometric series whose value is $\beta \frac{(1-\alpha^k)}{(1-\alpha)}$, which simplifies the computation of Equation 1. Thus, instead of applying the function $f(x)$ n times, we coalesce and compute $\alpha^{\mathbf{n}}\mathbf{x} + \beta \frac{(1-\alpha^{\mathbf{n}-1})}{(1-\alpha)}$.

4.2.2 Implementing load update coalesce for sandbox resume. We modify the sandbox pause and resume path to implement the load update coalesce. When a sandbox is paused, using its number of vCPUs, we compute $\alpha^{\mathbf{n}}$ and $\beta\frac{(1-\alpha^{\mathbf{n}-1})}{(1-\alpha)}$ and save these two values as an attribute of the sandbox. When the sandbox is resumed, for the target run queue load update, instead of iterating over the sandbox vCPUs and updating the run queue load, we apply the coalesce formula and use the precomputed parameters for the formula to speed the new load computation.

5 EVALUATIONS

Prototype implementation details. We implement a prototype of Horse within the Firecracker $v1.3.3$ (by modifying the Linux KVM pause/resume) and Xen hypervisor 4.17. Our changes amount to 76 and 11 LOCs to implement $\mathcal{P}^2\mathcal{SM}$ and load coalescing in Firecracker, respectively. With Xen, our changes amount to 72 and 12 LOcs for $\mathcal{P}^2\mathcal{SM}$ and load coalescing, respectively. Most changes for $\mathcal{P}^2\mathcal{SM}$ concern the additional data structures definitions and their update when the `ull_runqueue` is updated (Section 4.1.3).
Evaluation goal. We evaluate our Horse prototypes to answer the following questions: (Q_1) What is the contribution of each solution brought by Horse on the resume time of a sandbox? (Q_2) What is the impact of Horse techniques regarding the resume of a sandbox? (Q_3) What is the overhead of using Horse on a virtualization system? (Q_4) What is the impact of Horse regarding uLL workloads on FaaS platforms? (Q_5) What is the impact of Horse when colocating uLL workloads and longer-running functions?
Testbed and experimental procedure. To answer the aforementioned questions, we run several experiments on a Cloud lab server with 2 CPUs Intel Xeon Platinium 8360Y with 36 cores at 2.40 GHz (144 threads with hyperthreading). The server possesses 128 GB memory, 240 GB of SSD storage, and a 10 GB network ethernet card. Unless otherwise specified, we run each experiment 10×, which is enough for us to achieve 95% confidence interval ≤ 3% for each experiment. Due to similar observations and space constraints, we only report the results for the Firecracker virtualization system.

5.1 $\mathcal{P}^2\mathcal{SM}$ for sandbox resumes

This section aims to answer (Q_1) and (Q_2).
Experimental procedure. In a server, we trigger the resume of a previously paused sandbox. The sandbox runs an Ubuntu server $v22.04.2$ atop of the Linux kernel $v6.2.9$. For each run, we vary the number of allocated vCPUs from 1 to 36 and measure the time necessary for the VM to be resumed. We consider the VM to be resumed when the virtualization system exits from the resume call. We run this experiment in four different setups: *vanil* (the vanilla setup), *ppsm* (the $\mathcal{P}^2\mathcal{SM}$ solution), *coal* (the load update coalescing), and Horse ($\mathcal{P}^2\mathcal{SM}$ and the load update coalescing).
Results. Figure 3 reports the results of the resume times for the four different setups. Compared to *vanil*: *coal* improves the resume time from 16% to 20%, *ppsm* improves the resume time from 55% to 69%. Thus, Horse improves the resume time by up to 85%, thus up to 7, 16×. Additionally, the resume time does not vary with the number of vCPUs of the resuming sandbox and achieves a constant $O(1)$ resume time ≈ $150ns$.

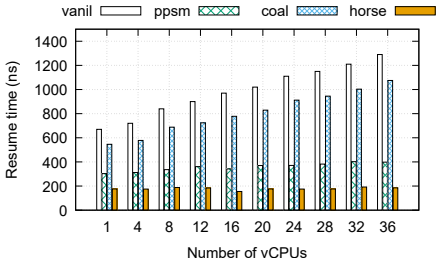

Figure 3: Resume time of a sandbox with the vanilla approach, $\mathcal{P}^2\mathcal{SM}$, load update coalescing, and Horse ($\mathcal{P}^2\mathcal{SM}$ + coalescing). Compared to vanilla, load update coalescing improves the resume time by up to 20%, $\mathcal{P}^2\mathcal{SM}$ improves the resume time by up to 70%.

5.2 Overhead of Horse

The goal of this section is to answer (Q_3). Horse's overhead occurs when uLL sandboxes are paused due to the update of the $\mathcal{P}^2\mathcal{SM}$ related data structures when the `ull_runqueue` changes and when resuming uLL sandboxes. We perform the following experiment.
Experimental procedure. On a server running 10 1-vCPU sandboxes (each running a CPU-intensive application with sysbench [38]), we successively create 10 sandboxes, pause them for 5 seconds, and then trigger their resume. These newly created sandboxes host uLL workloads that start 5 seconds after the sandbox initialization. Each sandbox is allocated 512MB of memory, and the host frequency governor is set to performance mode (all cores run at the highest frequency). We repeat the experiment while increasing the allocated vCPUs for the created sandboxes from 1 to 36. We aim to evaluate the overhead of Horse during uLL workloads sandboxes pause and resume compared to the vanilla approach. We use one `ull_runqueue` for all the uLL workloads in this experiment. For each run, we record the CPU and memory usage each 500ms. We perform the experiments on the Xen and Firecracker.
Results. Regarding memory usage, when uLL sandboxes are paused, the global memory usage increases by up to 528 KB with Horse. This is the memory footprint of the data structures used by $\mathcal{P}^2\mathcal{SM}$ for all the 10 paused uLL sandboxes. The low footprint can also be explained by the fact that $\mathcal{P}^2\mathcal{SM}$ data structures mostly reference other data structures. Concretely, in our setup, compared to the memory used on the server by all the running sandboxes, ≈ 5 GB, the memory overhead of Horse is about 0.11%. Even with several `ull_runqueue`, the overhead remains the same since each uLL sandbox is tied to one `ull_runqueue` when paused.

Regarding CPU usage, we make three main observations. Firstly, we observe a slight increase in CPU usage when pausing uLL sandboxes of up to 0.3%. This increase is due to the different computations performed for each paused uLL sandbox to speed the resumes, mainly regarding coalescing. Secondly, there is no significant increase in CPU usage due to updating the data structures necessary for $\mathcal{P}^2\mathcal{SM}$ independently of the size of uLL sandboxes. Lastly, resuming the uLL sandboxes with Horse incurs up to 2, 7% increase regarding the CPU usage. Despite the parallel algorithm of $\mathcal{P}^2\mathcal{SM}$, each thread performs a light operation, which induces a very light overall increase in CPU usage. Furthermore, since the resumed

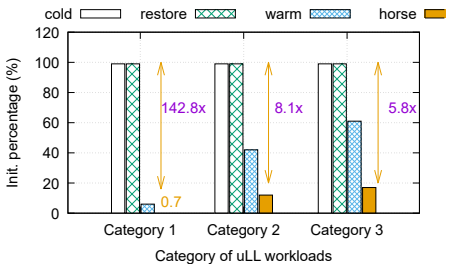

Figure 4: Sandox initialization percentage in the trigger process of 3 uLL workloads for four scenarios. Horse outclasses *warm* by up to $8,95\times$, *restore* by up to $142,7\times$, and *cold* by up to $142,84\times$.

sandboxes are uLL sandboxes, the workload rapidly ends even after resuming, which explains the lack of major CPU usage increase.

5.3 Horse on uLL workloads

The goal of this section is to answer (Q_4).

Experimental procedure. We trigger the execution of a uLL workload atop the Firecracker and Xen virtualization system. For the uLL workloads, we consider the three categories of uLL workloads similar to Section 2. For each uLL workload, We consider three FaaS trigger strategies: *cold*, *restore*, *warm*, and Horse. For each run, we collect the sandbox initialization and the workload execution time.

Results. Figure 4 reports the sandbox initialization percentage for the trigger of the 3 uLL workloads for all the FaaS scenarios for Firecracker. We observe that Horse achieves the lowest sandbox initialization percentage for all uLL workloads. Horse outclasses *warm* by up to $8,95\times$, *restore* by up to $142,7\times$, and *cold* by up to $142,84\times$. Concretely, with Horse, the sandbox initialization percentage varies between $0,77\%$ and $17,64\%$. These results show that Horse improves FaaS capability to accept uLL workloads.

5.4 Horse on longer running functions

The goal of this section is to answer (Q_5). Several studies reveal that a non-negligible fraction of serverless functions has an execution time longer than 1*s* [71, 79]. Consequently, we must measure the impact of Horse when uLL workloads are colocated with these longer-running functions and determine if the latter can be negatively affected by Horse. We perform the following experiment.

Experimental procedure. We trigger a function with arrival times derived from a 30s chunk of the Azure Cloud serverless real-world traces [12]. The function is the thumbnail generator from the SEBS benchmark suite [20], which generates thumbnails from images stored on an Amazon S3 bucket. The sandboxes are allocated 1GB of memory and 2 vCPUs. In parallel, for each 1s, we trigger 10 uLL workloads by resuming previously paused sandboxes. Our experiment is designed to prevent measurement noise from CPU contention due to resource scarcity so that both the uLL workloads and the thumbnail function instances theoretically have enough available cores. We repeat the experiment by varying the number of vCPUs of the uLL workloads sandboxes from 1 to 36 and record each thumbnail function latency during the experiment to compute the mean, 95th, and 99th percentile latencies. We run the experiment with the vanilla setting and Horse on Firecracker.

Results. Independently of the size of uLL sandboxes resumed, we observe no difference between the mean and 95th percentile latencies between vanilla and Horse. This is the consequence of isolating uLL sandboxes on a run queue different from the other functions, thus preventing contention between both function categories. However, regarding the 99th percentiles, Horse incurs an overhead of up to $0,00107\%$ (uLL sandbox vCPUs = 36), which is $\approx 30\mu s$. This is the extreme case where a thread used for resuming a uLL sandbox with $\mathcal{P}^2\mathcal{SM}$ preempts a longer-running function. However, the overhead is not significant since $\mathcal{P}^2\mathcal{SM}$ is fast enough.

6 RELATED WORK

Several research works try to improve initialization times for sandboxes in FaaS. These works either leverage snapshots to later perform optimized restore operations [8, 10, 78], memory deduplication to reduce the memory footprint of sandboxes to initialize [68], caching approaches to speed critical components loading into main memory [15, 50, 56, 64, 73] and fork based approaches [25, 82, 86]. Other research works propose an optimized message bus between functions to speed execution functions' execution [4, 46]. Horse does not target long-running functions but instead uLL.

Horse isolates uLL workloads on runqueues where longer running functions are excluded. This reduces the overhead of implementing $\mathcal{P}^2\mathcal{SM}$ and optimizes the load update coalescing approach. Other research works use similar strategies for scheduling, DVFS for latency-sensitive and throughput-oriented workloads [52, 59, 75] and even on NUMA [40, 53]. The most recent, Demeter [75], divides cores into hot, warm, and idle. Hot cores run at peak frequency and are meant for latency-sensitive workloads, while warm cores are for throughput-oriented.

7 CONCLUSION

We present Horse, a fast resume path for paused sandboxes that host uLL workloads. Horse introduces two mechanisms: $\mathcal{P}^2\mathcal{SM}$ and load update coalescing that aim to speed the paused sandbox vCPUs' merge operation to a run queue and update the load variable used for DVFS. Our evaluation shows that Horse improves warm sandboxes resume time by up to $7,16\times$ and sandbox initialization overhead reduction for uLL workloads by up to $142,84\times$.

8 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported by Rennes Métropole (Program "Allocation d'installation scientifique", AIS, Rennes, France), along with the French ANR project sGOV (ANR-23-CE25-0007-01).

REFERENCES

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

[2] Ankur Aggarwal and Annicka Garbers. Replace your hardware firewalls with cloudflare one. https://blog.cloudflare.com/replace-your-hardware-firewalls-with-cloudflare-one.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[6] Hrishikesh Amur, Ripal Nathuji, Mrinmoy Ghosh, Karsten Schwan, and Hsien-Hsin S Lee. Idlepower: Application-aware management of processor idle states. In *Proceedings of the Workshop on Managed Many-Core Systems, MMCS*, volume 8, 2008.

[7] Georgia Antoniou, Haris Volos, Davide B Bartolini, Tom Rollet, Yiannakis Sazeides, and Jawad Haj Yahya. Agilepkgc: An agile system idle state architecture for energy proportional datacenter servers. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 851–867. IEEE, 2022.

[8] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Apache. Apache: Open source serverless cloud platform. https://openwhisk.apache.org/.

[10] AWS. Improving startup performance with lambda snapstart. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html.

[11] Azure. Azure functions premium plan. https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal#eliminate-cold-starts.

[12] Azure. Azure public dataset. https://github.com/Azure/AzurePublicDataset.

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

[14] Alex Bocharov. Every request, every microsecond: scalable machine learning at cloudflare. https://blog.cloudflare.com/scalable-machine-learning-at-cloudflare.

[15] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association.

[16] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 141–159, Boston, MA, April 2023. USENIX Association.

[17] Chih-Hsun Chou, Laxmi N. Bhuyan, and Daniel Wong. μdpm: Dynamic power management for the microsecond era. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 120–132, 2019.

[18] Alibaba Cloud. Function compute: A secure and stable, elastically scaled, o&m-free, pay-as-you-go, serverless computing platform. https://www.alibabacloud.com/product/function-compute.

[19] Azure Cloud. Azure functions: Execute event-driven serverless code functions with an end-to-end development experience. https://azure.microsoft.com/en-us/products/functions/.

[20] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.

[21] Jonathan Corbet. Per-entity load tracking. https://lwn.net/Articles/531853/.

[22] Datadog. The state of serverless. https://www.datadoghq.com/state-of-serverless/.

[23] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, feb 2013.

[24] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[25] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[27] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[28] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 249–264, USA, 2016. USENIX Association.

[29] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1249–1266, Renton, WA, April 2022. USENIX Association.

[30] Eashan Gupta, Prateesh Goyal, Ilias Marinos, Chenxingyu Zhao, Radhika Mittal, and Ranveer Chandra. Dbo: Fairness for cloud-hosted financial exchanges. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 550–563, New York, NY, USA, 2023. Association for Computing Machinery.

[31] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

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

[33] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, aug 2014.

[34] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[36] Ki-Dong Kang, Gyeongseo Park, Hyosang Kim, Mohammad Alian, Nam Sung Kim, and Daehoon Kim. Nmap: Power management based on network packet processing mode transition for latency-critical workloads. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 143–154, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J. Ramanujam. A massively parallel distributed n-body application implemented with hpx. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 57–64, 2016.

[38] Alexey Kopytov. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/*, 2004.

[39] AWS Lambda. Configuring provisioned concurrency. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html.

[40] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on {NUMA} systems: Asymmetry matters. In *2015 USENIX annual technical conference (USENIX ATC 15)*, pages 277–289, 2015.

[41] Yilong Li, Seo Jin Park, and John Ousterhout. MilliSort and MilliQuery: Large-Scale Data-Intensive computing in milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 593–611. USENIX Association, April 2021.

[42] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186. USENIX Association, November 2020.

[43] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 301–312. IEEE Press, 2014.

[44] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, Renton, WA, April 2022. USENIX Association.

[46] Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, 2017.

[47] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 185–198, 2019.

[48] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 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, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[49] Moyand and Xuyida (Alibaba Cloud). Configure provisioned instances and auto scaling rules. https://www.alibabacloud.com/help/en/fc/configure-provisioned-instances-and-auto-scaling-rules.

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

[51] Djob Mvondo, Antonio Barbalace, Alain Tchana, and Gilles Muller. Tell me when you are sleepy and what may wake you up! In *Proceedings of the ACM Symposium on Cloud Computing*, pages 562–569, 2021.

[52] Djob Mvondo, Boris Teabe, Alain Tchana, Daniel Hagimont, and Noel De Palma. Closer: A new design principle for the privileged virtual machine os. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 49–60, 2019.

[53] Djob Mvondo, Boris Teabe, Alain Tchana, Daniel Hagimont, and Noel De Palma. Memory flipping: a threat to numa virtual machines in the cloud. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 325–333, 2019.

[54] Ali Najafi and Michael Wei. Graham: Synchronizing clocks by leveraging local clock properties. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 453–466, Renton, WA, April 2022. USENIX Association.

[55] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[56] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[57] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[58] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4), nov 2015.

[59] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 342–355, New York, NY, USA, 2015. Association for Computing Machinery.

[60] The Xen Project. Credit2 scheduler. https://wiki.xenproject.org/wiki/Credit2_Scheduler.

[61] Bartłomiej Przybylski, Maciej Pawlik, Paweł Żuk, Bartłomiej Lagosz, Maciej Malawski, and Krzysztof Rzadca. Using unused: Non-invasive dynamic faas infrastructure with hpc-whisk. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2022.

[62] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, October 2018. USENIX Association.

[63] Risk.net. Benchmarking deep neural networks for low-latency trading and rapid backtesting. https://www.risk.net/insight/technology-and-data/7956168/benchmarking-deep-neural-networks-for-low-latency-trading-and-rapid-backtesting.

[64] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 122–137, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Mashup: making serverless computing useful for hpc workflows via hybrid execution. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 46–60, New York, NY, USA, 2022. Association for Computing Machinery.

[66] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA, May 2011. USENIX Association.

[67] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 231–246, New York, NY, USA, 2023. Association for Computing Machinery.

[68] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Amazon Web Services. Aws lambda: Run code without thinking about servers or clusters. https://aws.amazon.com/lambda/.

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

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

[72] Erfan Sharafzadeh, Seyed Alireza Sanaee Kohroudi, Esmail Asyabi, and Mohsen Sharifi. Yawn: A cpu idle-state governor for datacenter applications. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 91–98, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. Sponge: Fast reactive scaling for stream processing with serverless frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 301–314, 2023.

[74] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020.

[75] Wenda Tang, Yutao Ke, Senbo Fu, Hongliang Jiang, Junjie Wu, Qian Peng, and Feng Gao. Demeter: Qos-aware cpu scheduling to reduce power consumption of multiple black-box workloads. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 31–46, New York, NY, USA, 2022. Association for Computing Machinery.

[76] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in network function virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 283–297, Renton, WA, April 2018. USENIX Association.

[77] Paul Turner. Sched: Entity load-tracking re-work. https://lkml.org/lkml/2012/2/1/763.

[78] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[79] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[80] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. {InfiniCache}: Exploiting ephemeral serverless functions to build a {Cost-Effective} memory cache. In *18th USENIX conference on file and storage technologies (FAST 20)*, pages 267–281, 2020.

[81] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.

[82] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.

[83] Felix Winterstein. Ultra-low latency xgboost with xelera silva. https://www.xelera.io/post/ultra-low-latency-xgboost-with-xelera-silva.

[84] Jawad Haj Yahya, Haris Volos, Davide B. Bartolini, Georgia Antoniou, Jeremie S. Kim, Zhe Wang, Kleovoulos Kalaitzidis, Tom Rollet, Zhirui Chen, Ye Geng, Onur Mutlu, and Yiannakis Sazeides. Agilewatts: An energy-efficient cpu core idle-state architecture for latency-sensitive server applications. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 835–850, 2022.

[85] Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. Halo: Accelerating flow classification for scalable packet processing in nfv. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 601–614, 2019.

[86] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. Beehive: Sub-second elasticity for web services with semi-faas execution. In *Proceedings of the 28th ACM International Conference on Architectural Support*

for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, page 74–87, New York, NY, USA, 2023. Association for Computing Machinery.

[87] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.