Benchmarking, Analysis, and Optimization of Serverless Function Snapshots

Dmitrii Ustiugov*
University of Edinburgh
United Kingdom

Plamen Petrov
University of Edinburgh
United Kingdom

Marios Kogias†
Microsoft Research
United Kingdom

Edouard Bugnion
EPFL
Switzerland

Boris Grot
University of Edinburgh
United Kingdom

ABSTRACT

Serverless computing has seen rapid adoption due to its high scalability and flexible, pay-as-you-go billing model. In serverless, developers structure their services as a collection of functions, sporadically invoked by various events like clicks. High inter-arrival time variability of function invocations motivates the providers to start new function instances upon each invocation, leading to significant cold-start delays that degrade user experience. To reduce cold-start latency, the industry has turned to *snapshotting*, whereby an image of a fully-booted function is stored on disk, enabling a faster invocation compared to booting a function from scratch.

This work introduces vHive, an open-source framework for serverless experimentation with the goal of enabling researchers to study and innovate across the entire serverless stack. Using vHive, we characterize a state-of-the-art snapshot-based serverless infrastructure, based on industry-leading Containerd orchestration framework and Firecracker hypervisor technologies. We find that the execution time of a function started from a snapshot is 95% higher, on average, than when the same function is memory-resident. We show that the high latency is attributable to frequent page faults as the function's state is brought from disk into guest memory one page at a time. Our analysis further reveals that functions access the same stable working set of pages across different invocations of the same function. By leveraging this insight, we build REAP, a light-weight software mechanism for serverless hosts that records functions' stable working set of guest memory pages and proactively prefetches it from disk into memory. Compared to baseline snapshotting, REAP slashes the cold-start delays by 3.7×, on average.

*Corresponding author: dmitrii.ustiugov@ed.ac.uk.
†This work was done while the author was at EPFL, Switzerland.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446714

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Information systems** → **Computing platforms**; **Data centers**; • **Software and its engineering** → **n-tier architectures**.

KEYWORDS

cloud computing, datacenters, serverless, virtualization, snapshots

ACM Reference Format:
Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. 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), April 19–23, 2021, Virtual, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446714

1 INTRODUCTION

Serverless computing has emerged as the fastest growing cloud service and deployment model of the past few years, increasing its Compound Annual Growth Rate (CAGR) from 12% in 2017 to 21% in 2018 [18, 44]. In serverless, services are decomposed into collections of independent stateless functions that are invoked by events specified by the developer. The number of active functions at any given time is determined by the load on that specific function, and could range from zero to thousands of concurrently running instances. This scaling happens automatically, on-demand, and is handled by the cloud provider. Thus, the serverless model combines extreme elasticity with pay-as-you-go billing where customers are charged only for the time spent executing their requests – a marked departure from conventional virtual machines (VMs) hosted in the cloud, which are billed for their up-time regardless of usage.

To make the serverless model profitable, cloud vendors colocate *thousands* of independent function instances on a single physical server, thus achieving high server utilization. A high degree of colocation is possible because most functions are invoked relatively infrequently and execute for a very short amount of time. Indeed, a study at Microsoft Azure showed that 90% of functions are triggered less than once per minute and 90% of the functions execute for less than 10 seconds [53].

Because of their short execution time, booting a function (i.e., *cold start*) has overwhelmingly expensive latency, and can easily dominate the total execution time. Moreover, customers are not billed for the time a function boots, which de-incentivizes the cloud vendor from booting each function from scratch on-demand.

Customers also have an incentive to avoid cold starts because of their high impact on latency [48]. As a result, both cloud vendors and their customers prefer to keep function instances memory-resident (i.e., *warm*) [29, 45, 48]. However, keeping idle function instances alive wastefully occupies precious main memory, which accounts for 40% of a modern server's typical capital cost [5]. With serverless providers instantiating thousands of function on a single server [5, 7], the memory footprint of keeping all instances warm can reach into hundreds of GBs.

To avoid keeping thousands of functions warm while also eliding the high latency of cold-booting a function, the industry has embraced *snapshotting* as a promising solution. With this approach, once a function instance is fully booted, its complete state is captured and stored on disk. When a new invocation for that function arrives, the orchestrator can rapidly load a new function instance from the corresponding snapshot. Once loaded, the instance can immediately start processing the incoming invocation, thus eliminating the high latency of a cold boot.

To facilitate deeper understanding and experimentation with serverless computing, this work introduces *vHive*, an open-source framework for serverless experimentation, which enables systems researchers to innovate across the entire serverless stack.[1] Existing open-source systems and frameworks are ill-suited for researchers, being either incomplete, focusing only on one of the components, such as a hypervisor [28], or rely on insufficiently secure container isolation [8–10, 30, 36]. vHive integrates open-source production-grade components from the leading serverless providers, namely Amazon Firecracker [5], Containerd [21], Kubernetes [37], and Knative [12], that offer the latest virtualization, snapshotting, and cluster orchestration technologies along with a toolchain for functions deployment and benchmarking.

Using vHive, we study the cold-start latency of functions from the FunctionBench suite [32, 33], their memory footprint, and their spatio-temporal locality characteristics when the functions run inside Firecracker MicroVMs [5] as part of the industry-standard Containerd infrastructure [21, 60]. We focus on a state-of-the-art baseline where the function is restored from a snapshot on a local SSD, thus achieving the lowest possible cold-start latency with existing snapshotting technology [26, 58].

Based on our analysis, we make three key observations. First, restoring from a snapshot yields a much smaller memory footprint (8-99MB) for a given function than cold-booting the function from scratch (148-256 MB) – a reduction of 61-96%. The reason for the greatly reduced footprint is that only the pages that are actually used by the function are loaded into memory. In contrast, when a function boots from scratch, both the guest OS and the function's user code engage functionality that is never used during serving a function invocation (e.g., function initialization).

Our second observation is that the execution time of a function restored from a snapshot is dominated by serving page faults in the host OS as pages are lazily mapped into the guest memory. The host OS serves these page faults one by one, bringing the pages from the backing file on disk. We find that these file accesses impose a particularly high overhead because the guest accesses lack spatial locality, rendering host OS' disk read-ahead prefetching ineffective. Altogether, we find that servicing page faults on the critical path of function execution slows down function processing by 95%, on average, compared to executing a function from memory (i.e., "warm").

Our last observation is that a given function accesses largely the same set of guest-physical memory pages across multiple invocations of the function. For the studied functions, 97%, on average of the memory pages are the same across invocations.

Leveraging the observations above, we introduce Record-and-Prefetch (REAP) – a light-weight software mechanism for serverless hosts that exploits recurrence in the memory working set of functions to reduce cold-start latency. Upon the first invocation of a function, REAP records a trace of guest-physical pages and stores the copies of these pages in a small working set file. On each subsequent invocation, REAP uses the recorded trace to proactively prefetch the entire function working set with a single disk read and eagerly installs it into the guest's memory space. REAP is implemented entirely in userspace, using the existing Linux user-level page fault handling mechanism [39]. Our evaluation shows that REAP eliminates 97% of the pages faults, on average, and reduces the cold-start latency of serverless functions by an average of 3.7×.

We summarize our contributions as following:

- We release vHive, an open-source framework for serverless experimentation, combining production-grade components from the leading serverless providers to enable innovation in serverless systems across their deep and distributed software stack.
- Using vHive, we demonstrate that the state-of-the-art approach of starting a function from a snapshot results in low memory utilization but high start-up latency due to lazy page faults and poor locality in SSD accesses. We further observe that the set of pages accessed by a function across invocations recurs.
- We present REAP, a record-and-prefetch mechanism that eagerly installs the set of pages used by a function from a pre-recorded trace. REAP speeds up function cold start time by 3.7×, on average, without introducing memory overheads or memory sharing across function instances.
- We implement REAP entirely in userspace with minimal changes to the Firecracker hypervisor and no modifications to the kernel. REAP is independent of the underlying serverless infrastructure and can be trivially integrated with other serverless frameworks and hypervisors, e.g., Kata Containers [3] and gVisor [28].

2 SERVERLESS BACKGROUND

2.1 Workload Characteristics and Challenges

Serverless computing or Function as a Service (FaaS) is an increasingly popular paradigm for developing and deploying online services. In the serverless model, the application functionality is sliced into one or more stateless event-driven jobs (i.e., functions), executed by the FaaS provider. Functions are launched on-demand based on the specified event triggers, such as HTTP requests. All major cloud providers support serverless deployments; examples include Amazon Lambda [13] and Azure Functions [45].

A recent study of Azure Functions in production shows that serverless functions are short-running, invoked infrequently, and function invocations are difficult to predict [53]. Specifically, the Azure study shows that half of the functions complete within 1

[1] The code is available at https://github.com/ease-lab/vhive.

second while >90% of functions have runtime below 10 seconds. Another finding is that functions tend to have small memory footprints: >90% of functions allocate less than 300MB of virtual memory. Lastly, 90% of functions are invoked less frequently than once per minute, albeit >96% functions are invoked at least once per week.

Given these characteristics of functions, the providers seek to aggressively co-locate thousands of function instances that share physical hosts to increase utilization of the provider's server fleet [5]. For example, a stated goal for AWS Lambda is deploying 4-8 *thousand* instances on a single host [5, 7].

This high degree of colocation brings several challenges. First, serverless functions run untrusted code provided by untrusted cloud service developers that introduces a challenge for security. Second, serverless platforms aim to be general-purpose, supporting functions written in different programming languages for a standard Linux environment. As a result, most serverless providers use virtualization sandboxes that either run a full-blown guest OS [1, 3, 5, 14, 50] or emulate a Linux environment by intercepting and handling a sandboxed application's system calls in the hypervisor [28].

Another challenge for serverless deployments is that idle function instances occupy server memory. To avoid wasting memory capacity, most serverless providers tend to limit the lifetime of function instances to 8-20 minutes after the last invocation due to the sporadic nature of invocations, deallocating instances after a period of inactivity and starting new instances on demand. Hence, the first invocation after a period of inactivity results in a start-up latency that is commonly referred to as the serverless function *cold-start* delay. In the last few years, high cold-start latencies have become one of the central problems in serverless computing and one of the key metrics for evaluating serverless providers [54, 56].

2.2 Hypervisor Specialization for Cold Starts

As noted in the previous section, leading serverless vendors, including Amazon Lambda, Azure Functions, Google Cloud Functions, and Alibaba Cloud, choose virtual machines (VMs) as their sandbox technology in order to deliver security and isolation in a multi-tenant environment. Although historically virtualization is known to come with significant overheads [51], recent works in hypervisor specialization, including Firecracker [5] and Cloud Hypervisor [1], show that virtual machines can offer competitive performance as compared to native execution (e.g., Docker containers), even for the cold-start delays.

Firecracker is a recently introduced hypervisor with a minimal emulation layer, supporting just a single *virtio* network device type and a single block device type, and relying on the host OS for scheduling and resource management [5]. This light-weight design allows Firecracker to slash VM boot time to 125ms and reduces the hypervisor memory footprint to 3MB [5, 7]. However, we measure that booting a Firecracker VM within production-grade frameworks, such as Containerd [21] or OpenNebula [50], takes 700-1300ms since their booting process is more complex, e.g., it includes mounting an additional virtual block device that contains a containerized function image [52, 60]. Finally, the process inside the VM, which receives the function invocation in the form of an RPC, takes up to several seconds to bootstrap before it is able to invoke the user-provided function, which may have its own initialization phase [26]. Together these delays – which arise on the critical path of function invocation – significantly degrade the end-to-end execution time of a function.

2.3 VM Snapshots for Function Cold Starts

To reduce cold-start delays, researchers have proposed a number of VM *snapshotting* techniques [26, 28, 58]. Snapshotting captures the current state of a VM, including the state of the virtual machine monitor (VMM) and the guest-physical memory contents, and store it as files on disk. Using snapshots, the host orchestrator (e.g., Containerd [21]) can capture the state of a function instance that has been fully booted and is ready to receive and execute a function invocation. When a request for a function without a running instance but with an existing snapshot arrives, the orchestrator can quickly create a new function instance from the corresponding snapshot. Once loading finishes, this instance is ready to process the incoming request, thus eliminating the high cold-boot latency.

Snapshots are attractive because they require no main memory during the periods of a function's inactivity and reduce cold-start delays. The snapshots of function instances can be stored in local storage (e.g., SSD) or in a remote storage (e.g., disaggregated storage service).

The state-of-the-art academic work on function snapshotting, Catalyzer [26], showed that snapshot-based restoration in the context of gVisor [28] virtualization technology can be performed in 10s-100s of milliseconds.[2] To achieve such a short start-up time, Catalyzer minimizes the amount of processing on the critical path of loading a VM from a snapshot. First, Catalyzer stores the minimum amount of snapshot state that is necessary to resume VM execution de-serialized to facilitate VM loading. After that, Catalyzer maps the plain guest-physical memory file as a file-backed virtual memory region and resumes VM execution. Crucially, the guest-physical memory of the VM is not populated with memory contents, which reside on disk, when the user code of the function starts running. As a result, each access to a yet-untouched page raises a page fault. These page faults occur on the critical path of function execution and, as we show in §4, significantly increase the runtime cost of a function loaded from a snapshot.

Recently, Firecracker introduced their own open-source snapshotting mechanism that follows the same design principles as Catalyzer, which is proprietary. Similarly to Catalyzer, loading a Firecracker VM from a snapshot is done in two phases. First, the hypervisor process loads the state of the VMM and the emulated devices (that we further refer to as *loading VMM* for brevity) and then maps a plain guest-physical memory for lazy paging [58].

3 VHIVE: AN OPEN-SOURCE FRAMEWORK FOR SERVERLESS EXPERIMENTATION

To enable a deeper understanding of serverless computing platforms, this paper introduces *vHive*, an open-source framework for experimentation with serverless computing. As depicted in Fig. 1,

[2]Here we only consider Catalyzer's "cold-boot" design that does not share memory across instances. We discuss Catalyzer's warm-boot designs in §8.

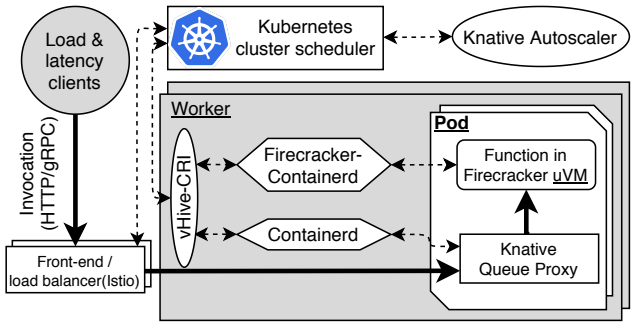


Figure 1: vHive architecture overview. Solid and dashed arrows show the data plane and the control plane, respectively.

vHive integrates production-grade components from the leading serverless providers, such as Amazon and Google.

3.1 Deploying and Programming with Functions in vHive

vHive adopts Knative [12], a serverless framework that runs on top of Kubernetes [37] and offers a programming and deployment model that is similar to AWS Lambda [13] and Azure functions [45]. To deploy an application in vHive, one can deploy application functions by supplying Knative with each function's Open Container Initiative (OCI) [61] image, e.g., a Docker image, along with a configuration file. This OCI image contains the function's handle code, which is executed by an HTTP or gRPC server upon an invocation. The configuration file contains the relevant environment variables and other parameters for function composition and function instances scaling. Using the configuration files, the application developers can compose their functions with any conventional "serverful" services with Kubernetes providing their URLs to the relevant functions. For example, functions that use large inputs or produce large outputs, like photos or videos, often have to save them in an object store or a database.

Upon a function's deployment, Knative provides a URL for triggering this function. Using these URLs, application developers can compose their functions, e.g., by specifying the URL of a callee function in the configuration file of the caller function. For each function, Knative configures the load-balancer service, sets up the network routes and dynamically scales the number of instances of the function in the system, according to changes in the function's invocation traffic.

3.2 vHive Infrastructure Components

Serverless infrastructure comprises of the front-end fleet of servers that expose the HTTP endpoints for function invocations, the worker fleet that executes the function code, and the cluster manager that is responsible for managing and scaling function instances across the workers [5, 23, 53]. These components are connected by an HTTP-level fabric, e.g., gRPC [2], that that enables management and resources monitoring [5].

A function invocation, in the form of an HTTP request or an RPC, first arrives at one of the front-end servers for request authentication and mapping to the corresponding function. In vHive, the Istio service [11] plays the roles of an HTTP endpoint and a load balancer for the deployed functions. If the function that received an invocation has at least one active instance, the front-end server simply routes the invocation request to an active instance for processing.

If there are no active function instances, the load balancer contacts the cluster manager to start a new instance of the function before the load balancer routes this invocation to a worker. vHive relies on Kubernetes cluster orchestrator to automate services deployment and management. Knative seamlessly extends Kubernetes, which was originally designed for conventional "serverful" services, to enable autoscaling of functions. For each function, Knative deploys an autoscaler service that monitors the invocation traffic to each function and makes decisions on scaling the number of functions instances in the cluster based on observed load.

At the autoscaler's decision, a chosen worker's control plane starts a new function instance as a pod, the scaling granule in Kubernetes, that contains a Knative Queue-Proxy (QP) and a MicroVM that runs the function code. The QP implements a software queue and a health monitor for the function instance, reporting the queue depth to the function's autoscaler, which is the basis for the scaling decisions. The function runs in a MicroVM to isolate the worker host from the untrusted developer-provided code. vHive follows the model of AWS Lambda, which deploys a single function inside a MicroVM that processes a single invocation at a time [5].

To implement the control plane, vHive introduces a vHive-CRI orchestrator service that integrates the two forks of Containerd – the stock version [21] and the Firecracker-specific version developed for MicroVMs [60] – for managing the lifecycle of containerized services (e.g., the QP) and MicroVMs. The vHive-CRI orchestrator receives Container-Runtime Interface (CRI) [20] requests from the Kubernetes control plane and processes them, making the appropriate calls to the corresponding Containerd services. Once the load balancer, which received the function invocation, the QP, and the function instance inside a MicroVM establish the appropriate HTTP-level connections, the data plane of the function is ready to process function invocations. When the function instance finishes processing the invocation, it responds to the load balancer, which forwards the response back to the invoking client.

vHive enables systems researchers to experiment with serverless deployments that are representative of production serverless clouds. vHive allows easy analyzing of the performance of an arbitrary serverless setting by offering access to Containerd and Kubernetes logs with high-precision timestamps or by collecting custom metrics. vHive also includes the client software to evaluate the response time of the deployed serverless functions in different scenarios, varying the mix of functions and the load. Finally, vHive lets the users experiment with several modes for cold function invocations, including loading from a snapshot or booting a new VM from a root filesystem.

4 SERVERLESS LATENCY AND MEMORY FOOTPRINT CHARACTERIZATION

In this section, we use vHive to analyze latency characteristics and memory access patterns of serverless functions, deployed in Firecracker MicroVM instances with snapshot support [58].

4.1 Evaluation Methodology

Similarly to prior work [26], we focus on the evaluation of a single worker server. The existing distributed serverless stack contributes little, e.g., less than 30ms as shown by AWS [5], to the overall end-to-end latency, as compared to many hundreds of milliseconds of the worker-related latency that we demonstrate below. Prior work measured the cold-start delay as the time between starting to load a VM from a snapshot to the time the instance executes the first instruction of the user-provided code of the function [26]. As the metric for our cold-start delay measurements, we choose the latency that includes not only the critical path of the VM restoration but the entire cold function invocation latency on a single worker. The measurements capture the latency from the moment a worker receives a function invocation request to the moment when the worker is ready to send the function's response back to the load balancer. This latency includes both the control-plane delays (including interactions with Containerd and Firecracker hypervisor) and data-plane time that is gRPC request processing and actual function execution.

Our experiments aim to closely model the workloads as in a modern serverless environment. First, we adopt a number of functions, listed in Table 1, from a representative serverless benchmark suite called FunctionBench [32, 33]. Second, to simulate the low invocation frequency of serverless functions in production [53], the host OS' page cache is flushed before each invocation of a cold function.

To evaluate the cold-start start delay in a serverless platform similar to AWS Lambda, we augment the vHive-CRI orchestrator to act as a MicroManager in AWS [5]. In this implementation, the vHive-CRI orchestrator not only implements the control plane but also acts as a data plane software router that forwards incoming function invocations to the appropriate function instance and waits for its response over a persistent gRPC connection. Note that in this setting, the worker infrastructure does not include the Queue-Proxy containers so that the data plane resembles per-function gRPC connections. Without a loss of generality, this work assumes the fastest possible storage for the snapshots that is a local SSD, which yields the lowest possible cold-start latency compared to a local HDD or disaggregated storage. §6.1 provides further details of the platform as well as the host and the guest setup.

To study the memory access patterns of serverless functions, we trace the guest memory addresses that a function instance accesses between the point when the vHive-CRI orchestrator starts to load a VM from a snapshot and the moment when the orchestrator receives a response from the function. As Firecracker lazily populates the guest memory, first memory access to each page from the hypervisor or the guest raises a page fault in the host that can be traced. We use Linux `userfaultfd` feature [39] that allows a userspace process to inspect the addresses and serve the page faults on behalf of the host OS.

4.2 Quantifying Cold-Start Delays

We start by evaluating the cold-start latency of each function under study and compare it to the invocation latency of the warm function instance. Recall that a warm instance is memory-resident and does not experience any cold-start delay when invoked. To obtain a detailed cold-start latency breakdown, we instrument the vHive-CRI orchestrator and invoke each function 10 times. To model a cold invocation, we flush the host OS page caches after each measurement.

Figure 2 shows the latency for the cold and warm invocations for each function. As expected, when a function instance remains warm (i.e., stays in memory), the instance delivers a very low invocation latency. By contrast, we find that a cold invocation from a snapshot takes one to two orders of magnitude longer than a warm invocation, which indicates that even with state-of-the-art snapshotting, cold-start delays are a major pain point for functions.

To investigate the performance difference, we examine the end-to-end cold invocation latency breakdown. First, the vHive-CRI orchestrator spawns a new Firecracker process and restores the virtual machine monitor (VMM) state as well as the state of the emulated network and block devices – the *Load VMM* latency component. After that, the orchestrator resumes the loaded function instance's virtual CPUs and restores the persistent gRPC connection to the gRPC server inside the VM. We name this latency component as *Connection restoration*. These two latency components are universal across all functions as they are part of the serverless infrastructure. Finally, we measure the actual function invocation processing time, referred to as *Function processing*.

The per-function latency breakdown is also plotted in Figure 2. We observe that the first two universal components, namely *Load VMM* and *Connection restoration*, take 156-317 ms. Meanwhile, the actual function processing takes much longer (95% longer on average) for cold invocations as compared to warm invocations of the same functions, reaching into 100s of milliseconds even for functions like `helloworld` and `pyaes` that take only a few milliseconds to execute when warm.

The state-of-the-art snapshotting techniques rely on lazy paging to eliminate the population of guest memory from the critical path of VM restoration (§2.2). A consequence of this design is that each page touched by a function must be faulted in at the first access, resulting in thousands of page faults during a single invocation of a function. Page faults are processed serially because the faulting thread is halted until the OS brings the memory page from disk and installs it into the virtual address space by setting up the memory mappings in the process page table. In this case, the performance of the guest significantly depends on the disk latency as the OS needs to bring the missing pages from the guest memory file.

We also study the contiguity of the faulted pages, with the results depicted in Fig. 3. We find that function instances tend to access pages that are located in non-adjacent locations inside the guest memory. This lack of spatial locality significantly increases disk access time, and thus page fault delays, because sparse accesses to disk cannot benefit from the host OS's run-ahead prefetching. Fig. 3 shows that the average length of the contiguous regions of the guest-physical memory is around 2-3 pages for all functions except `lr_training` that shows contiguity of up to 5 pages.

4.3 Function Memory Footprint & Working Set

The above analysis demonstrates the benefits of keeping functions warm, because cold function invocations significantly increase the end-to-end function invocation latency. In this subsection, we show

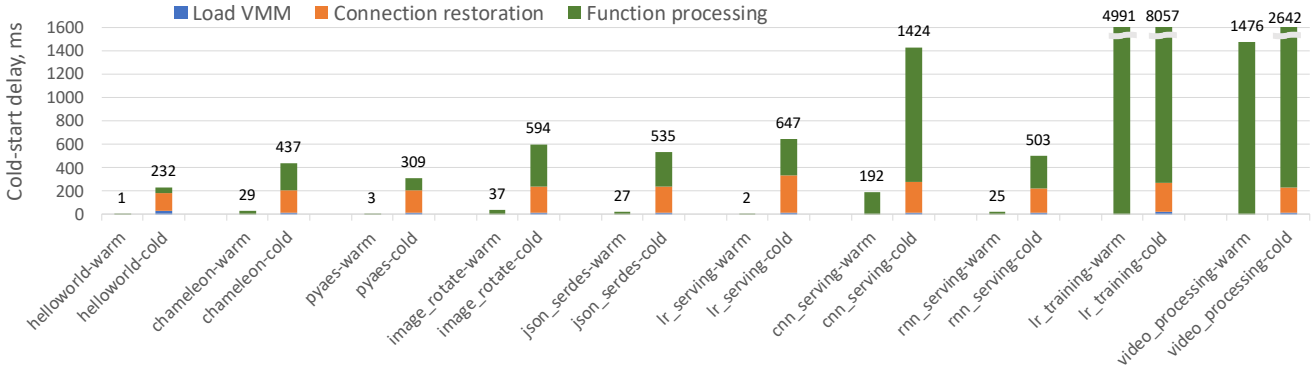


Figure 2: Cold-start latency breakdown for Firecracker's snapshot load mechanism, compared to the warm latency.

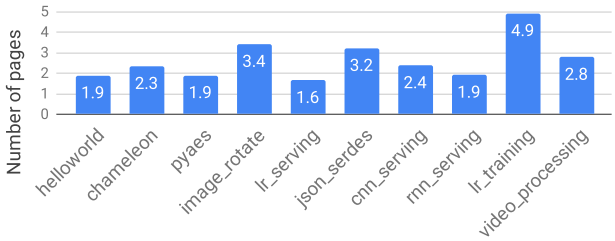


Figure 3: Guest memory pages contiguity.

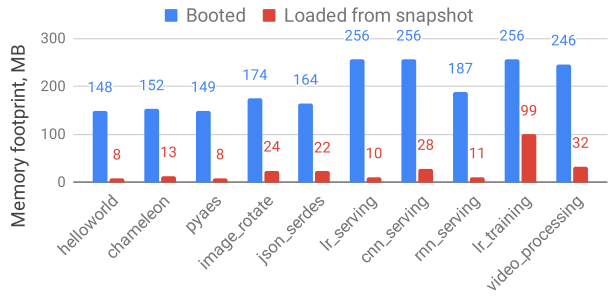


Figure 4: Memory footprint of function instances after one invocation.

that despite the advantages of warm functions, keeping many functions warm is wasteful in terms of memory capacity.

We first investigate the fraction of a function instance's memory footprint that is related to the actual function invocation. First, we measure the total footprint of a booted VM after the first function invocation is complete using the Linux `ps` command, since a VM appears as a regular process in the host OS. This footprint includes the hypervisor and the emulated layer overhead (around 3MB [5]), the memory pages that are accessed during the VM's boot process, function initialization, and the actual invocation processing. Second, for a VM that is loaded from a snapshot, we trace the pages, using Linux `userfaultfd` [39], that are accessed only during the invocation processing, i.e., from the moment the VM is loaded to the moment when the vHive-CRI orchestrator receives the response from the function. Unlike the first measurement, this footprint relates only to the invocation processing.

Figure 4 (the blue bars) shows the memory footprint of a single freshly-booted function instance. We observe that, for all functions, their memory footprint reaches 100-200MB. Thus, assuming that a serverless provider co-locates thousands of different functions instances on the same host and disallows memory sharing for security reasons (as is the case in practice [5]), the aggregate footprint of function instances will reach into hundreds of gigabytes.

Figure 4 also plots the footprint of the function instances loaded from a snapshot after the first invocation (red bars). We observe that, in this case, the functions' working sets span 8-99MB (24MB on average), which is 3-39%, and 9% on average of their memory footprint after booting. The reason why the memory footprint of a function booted from scratch is much higher than the one loaded from a snapshot is that starting an instance by booting requires many steps: booting a VM, starting up the Containerd's agents [52] as well as user-defined function initialization. This complex boot procedure engages much more logic (e.g., guest OS and userspace code) than just processing the actual function invocation, which naturally affects the former's memory footprint.

Despite the fact that, when loaded from a snapshot, the memory footprint of a function instance is relatively compact, the total memory footprint for thousands of such functions would still comprise tens of GBs. While potentially affordable memory-wise, we note that keeping this much state in memory is wasteful given the low invocation frequency of many functions (§2.1). Moreover, such a high memory commitment would preclude colocating memory-intensive workloads on nodes running serverless jobs, thus limiting a cloud operator's ability to take advantage of idle resources. We thus conclude that while functions loaded from a snapshot present an opportunity in terms of their small memory footprint, by itself, they are not a solution to the memory problem.

4.4 Guest Memory Pages Reuse

After establishing that the working sets of serverless functions booted from a snapshot are compact, we study how the working set of a given function changes across invocations. Our hypothesis is that the stateless nature of serverless functions results in a stable working set across invocations.

User and guest kernel code pages account for a large fraction of functions' footprint. This code belongs either to the underlying infrastructure or the actual function implementation. Providers

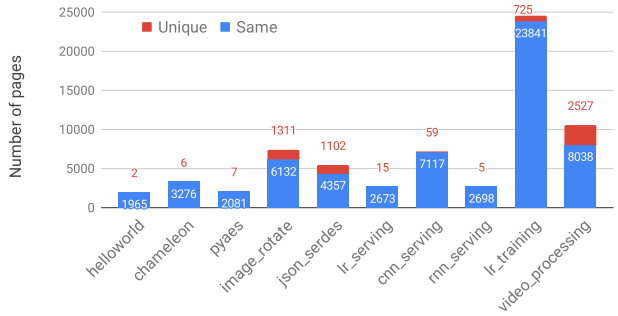


Figure 5: Number of pages that are unique or same across invocations with different inputs. The numbers above the bars correspond to uniquely accessed pages.

deploy additional control-plane services inside a function's sandbox and use general-purpose communication fabric (e.g., gRPC) to connect functions to the vHive-CRI orchestrator [5, 52]. The gRPC framework uses the standard TCP network protocol, similarly to AWS Lambda [5], that adds the guest OS's network stack to the instance footprint. Using the `helloworld` function, we estimate that this infrastructure overhead accounts for up to 8MB of a function's guest-memory footprint and is stable across function invocations.

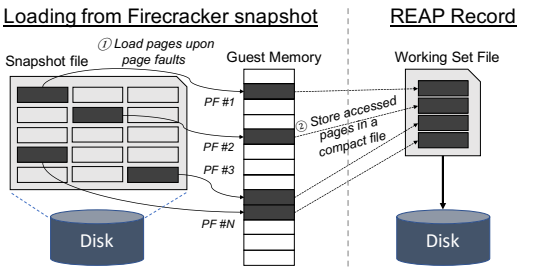
We observe that functions naturally use the same set of memory pages while processing different inputs. For example, when rotating different images or evaluating different customer reviews, functions use the same calls to the same libraries and rely on the same functionality inside the guest kernel, e.g., the networking stack. Moreover, the functions engage the same functionality that is a part of the provider's infrastructure, e.g., the Containerd's agents inside a VM [52]. Finally, we observe that even when a function's code performs a dynamic allocation, the guest OS buddy allocator is likely to make the same or similar allocation decisions. These decisions are based on the state of its internal structures (i.e., lists of free memory regions), which is the same across invocations being loaded from the same VM snapshot. Hence, the lack of concurrency and non-determinism inside the user code of the functions that we study results in a similar guest physical memory layout.

We validate our hypothesis about the working sets by studying the guest memory pages that are accessed when a function is invoked with different inputs. Fig. 5 demonstrates that the majority of pages accessed by all studied functions are the same across invocations with different inputs. For 7 out of 10 functions, more than 97% of the memory pages are identical across invocations. For `image_rotate`, `json_serdes`, `lr_training`, and `video_processing`, reuse is lower because these functions have large inputs (photos, JSON files, training datasets, and videos, respectively) that are 1-10MB in size. Nonetheless, even for these three functions, over 76% of memory pages are the same across invocations.

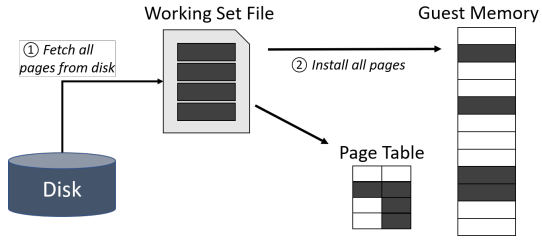
4.5 Summary

We have shown that invocation latencies of cold functions may exceed those of warm functions by one to two orders of magnitude, even when using state-of-the-art VM snapshots for rapid restoration of cold functions. We found that the primary reason for these elevated latencies is that the existing snapshotting mechanisms populate the guest memory on-demand, thus incurring thousands of page faults during function invocation. These page faults are served one-by-one by reading non-contiguous pages from a snapshot file on disk. The resulting disk accesses have little contiguity and induce significant delays in processing of these page faults, thus slowing down VM restoration from a snapshot.

We have further shown that function instances restored from a snapshot have compact working sets of guest memory pages, spanning just 24MB, on average. Moreover, these working sets are stable across different invocations of the same function; indeed, the function instances access predominantly the same memory pages even when invoked with different inputs.



(a) REAP record phase.



(b) REAP prefetch phase.

Figure 6: REAP's two-phase operation.

5 REAP: RECORD-AND-PREFETCH

The compact and stable working set of a function's guest memory pages, which instances of the given function access across invocations of the function, provides an excellent opportunity to slash cold-start delays by prefetching.

Based on this insight, we introduce Record-and-Prefetch (REAP), a light-weight software mechanism inside the vHive-CRI orchestrator to accelerate function invocation times in serverless infrastructures. REAP records a function's working set upon the first invocation of a function from a snapshot and replays the record to accelerate load times of subsequent cold invocations of the function by eliminating the majority of guest memory page faults. The rest of the section details the design of REAP.

5.1 REAP Design Overview

Given an existing function snapshot, REAP operates in two phases. During the *record* phase, REAP traces and inspects the page faults

that a function instance triggers when accessing pages in the guest memory, identifying the positions of these pages in the backing guest memory file (Fig. 6a). After a function invocation is complete, REAP creates two files, namely the *working set (WS) file)* that contains a copy of all accessed guest memory pages in a contiguous compact chunk and the *trace file* that contains the offsets of the original pages inside the guest memory file. The contiguous compact WS file can be rapidly brought into physical memory in a single read operation, which greatly reduces disk and system-level overheads in the snapshot baseline that requires many disparate accesses to pages scattered across the guest memory file on disk.

After the completion of the record phase, all future invocations of the function enjoy accelerated load times as REAP's *prefetch* phase eagerly populates the guest memory from the WS file before launching the function instance (Fig. 6b). Upon an arrival of a new invocation, REAP fetches the entire WS file from disk into a temporary buffer in the orchestrator's memory and eagerly installs the pages into the function instance's guest memory region. REAP also populates the page table of the instance in the host OS. As a result, when the instance is loaded, the function executes without triggering page faults to the stable memory working set. Page faults to uniquely-accessed pages in a given invocation are handled by REAP on demand.

5.2 Implementation

REAP adheres to the following design principles, which facilitate adoption and deployment in a cloud setting: i) REAP is agnostic to user codebase; ii) REAP is independent of the underlying serverless infrastructure; iii) REAP is implemented entirely in userspace without kernel modifications; iv) REAP works efficiently in a highly multi-tenant serverless environment.

We implement REAP as a part of the vHive-CRI orchestrator that controls the lifecycle of all function instances. For each function, the vHive-CRI orchestrator tracks active function instances and performs the necessary bookkeeping, including maintaining the snapshot files and working set records. To accommodate the highly-concurrent serverless environment with many function instances executing simultaneously, it is a fully parallel implementation with a dedicated *monitor* thread for each function instance. Each monitor thread records or prefetches the working set pages and also serves page faults that are raised by the corresponding instance. In our prototype, the monitor threads are implemented as lightweight goroutines, which are scheduled by the Go runtime.

To implement the monitor, we use the Linux `userfaultfd` feature that allows a userspace program to handle page faults on behalf of the OS. In Linux, a target process can register a virtual memory region in anonymous memory and request a user-fault file descriptor, which can be passed to a monitor running as a separate thread or process. The monitor polls for page fault events that the OS forwards to the user-fault file descriptor. Upon a page fault, the monitor installs the contents of the page that triggered the page fault. The monitor is free to retrieve page contents from any appropriate source, such as a file located on a local disk or from the network. Furthermore, the monitor can install any number of pages before waking up the target process. Thanks to these features, the monitor can support both local and remote snapshot storage, and can eagerly install the content of the entire WS file at once.

Upon each function invocation for which there is no warm instance available, the vHive-CRI orchestrator launches the monitor thread in one of two modes: record, if no WS file is available for this instance, or prefetch, if a corresponding WS file exists.

5.2.1 Record phase. The goal of the monitor during the record phase is to capture the memory working set for functions instantiated from snapshots. Before loading the VMM state from a snapshot, the hypervisor maps the guest memory file as an anonymous virtual memory region and requests a user-fault file descriptor from the host OS, passing this descriptor over a socket to the monitor thread of the vHive-CRI orchestrator. Then, the hypervisor restores the VMM and emulated devices' state and resume the virtual CPUs of the newly loaded function instance that can start processing the function invocation.

Every first access to a guest memory page raises a page fault that needs to be handled and recorded by the monitor. The monitor maps the guest memory file as a regular virtual memory region in the monitor's virtual address space and polls (using the `epoll` system call) for the host kernel to forward the page fault events, triggered by the instance. Upon receiving a page fault event, the monitor reads a control message from the user-fault file descriptor that contains the description of the page fault, including the address in the virtual address space of the target function instance. The monitor translates this virtual address into an offset that corresponds to the page location in the guest memory file. While serving the page faults, the monitor records the offsets of the working set pages in the trace file.

We augment the Firecracker hypervisor to inject the first page fault of each instance to the first byte of the instance's guest memory. Doing so allows file offsets for all of the following page faults to be derived by subtracting the virtual address of the first page fault. Using the file offset of the missing page, the monitor locates the page in the guest memory file and installs the page into the guest memory region of the instance by issuing an `ioctl` system call to the host kernel, which also updates the extended page tables of the target function instance. After the vHive-CRI orchestrator receives a response from the function, indicating that the function invocation processing has completed, the monitor copies the recorded working set pages, using the offsets recorded in the trace file, into a separate WS file (§4.3).

Note that the record phase increases the function invocation time as compared to the baseline snapshots due to userspace page fault handling. As such, REAP penalizes the first function invocation to benefit subsequent invocations. We quantify the recording overheads in §6.4.

5.2.2 Prefetch phase. For every subsequent function invocation, the vHive-CRI orchestrator spawns a dedicated monitor thread that uses the WS file to prefetch the working set memory pages from disk into a buffer in the monitor's memory with a single `read` system call. Then, the monitor eagerly and proactively installs the pages into the guest memory through a sequence of `ioctl` calls, following which it wakes up the target function instance with another `ioctl` call.

As in the record phase, the monitor maps the guest memory file during every subsequent cold function invocation. After installing all the working set pages from the WS file, the monitor keeps polling for page faults to pages that are missing from the stable working set and installs them on demand, as in the record phase. Since the WS file captures the majority of pages that a function instance accesses during an invocation, only a small number of page faults needs to be served by the monitor on demand.

5.2.3 Disk Bandwidth Considerations. REAP's efficiency depends entirely on the performance of the prefetch phase and, specifically, how fast the vHive-CRI orchestrator can retrieve the working set pages from disk. Although a single commodity SSD can deliver up to 1-3 GB/s of read bandwidth, SSD throughput varies considerably depending on disk access patterns. An SSD can deliver high bandwidth with one large multi-megabyte read request, or with >10 4KB requests issued concurrently. For example, on our platform (§6.1), with a standard Linux `fio` IO benchmark [42] that issues a single 4KB read request, the SSD can deliver only 32MB/s, whereas issuing 16 4KB requests can increase the SSD throughput to 360MB/s. While concurrent reads deliver much higher bandwidth than a single 4KB read, the achieved bandwidth is still considerably below the peak of 850MB/s of our Intel SATA3 SSD.

We find that REAP achieves close to the peak SSD read throughput (533-850MB/s) by fetching the WS file in a single >8MB read operation that bypasses the host OS' page cache (i.e., the WS file needs to be opened with the `O_DIRECT` flag).

5.3 Discussion

REAP adheres to the design principles set out in §5.2. We implement REAP entirely in userspace as a part of the vHive-CRI orchestrator. It is written in 4.5K Golang LoC, including tests and benchmarks, and is loosely integrated with the industry-standard Containerd framework [21, 52, 60] via gRPC. The implementation does not require any changes to host or guest OS kernel. We add less than 200 LoC to Firecracker's Rust codebase, not including two publicly available Rust crates that we used, to register a Firecracker VM's guest memory with `userfaultfd` and to delegate page faults handling to the vHive-CRI orchestrator. Finally, the orchestrator follows a purely parallel implementation by spanning a lightweight monitor thread (goroutine) per function instance.

6 EVALUATION

In this section, we describe the platform setup, including the host and the guest setups, and present REAP evaluation results. In our experiments, we follow the methodology that is described in §4.1, unless specified otherwise.

6.1 Evaluation Platform

We conduct our experiments on a 2×24-core Intel Xeon E5-2680 v3, 256GB RAM, Intel 200GB SATA3 SSD, running Ubuntu 18.04 Linux with kernel 4.15. We fix the CPU frequency at 2.5GHz to enable predictable and reproducible latency measurements. We disallow memory sharing among all function instances and disable swapping to disk, as suggested by AWS Lambda production guidelines [5, 59].

Table 1: Serverless functions adopted from FunctionBench.

Name	Description
helloworld	Minimal function
chameleon	HTML table rendering
pyaes	Text encryption with an AES block-cipher
image_rotate	JPEG image rotation
json_serdes	JSON serialization and de-serialization
lr_serving	Review analysis, serving (logistic regr., Scikit)
cnn_serving	Image classification (CNN, TensorFlow)
rnn_serving	Names sequence generation (RNN, PyTorch)
lr_training	Review analysis, training (logistic regr., Scikit)
video_processing	Applies gray-scale effect (OpenCV)

We use a collection of nine Python-based functions (Table 1) from the representative FunctionBench [32, 33] suite.[3] We also evaluate a simple `helloworld` function. The root filesystems for all functions are generated automatically by Containerd, using Linux device mapper functionality as used by Docker [25], from Linux Alpine OCI (Docker) images.[4] Functions with large inputs (namely `image_rotate`, `json_serdes`, `lr_training`, `video_processing`) retrieve their inputs from an S3 server [46] deployed on the same host.

We optimize virtual machines for minimum cold-start delays, similar to a production serverless setup, as in [5, 31]. The VMs run a guest OS kernel 4.14 without modules. Each VM instance has a single vCPU. We boot VM instances with 256MB guest memory, which is the minimum amount to boot all the functions in our study.

6.2 Understanding REAP Optimizations

We start by evaluating the cold-start latency of the `helloworld` function, whose short user-level execution time is useful for understanding serverless framework overheads. In addition to evaluating the baseline Firecracker snapshots and REAP as presented in §5, we also study two additional design points that help justify REAP's design decisions. Specifically, we consider the following configurations.

Vanilla snapshots: This is the baseline configuration, which restores the VMM and the emulation layer in 50ms, then spends 182ms processing the function invocation (Fig. 7) that takes just 1ms for for a warm instance (Fig. 2). The large processing delay is directly attributed to the handling of page faults in the critical path of function execution. As `helloworld`'s working set is around 8MB, one can infer that vanilla snapshotting is only able to utilize 43MB/s of SSD bandwidth, i.e., <5% of the peak bandwidth on our platform.

Parallel Page Fault handling: This design (labeled "Parallel PFs" in Fig. 7) parallelizes page fault processing. It does so by using the trace file specifying the offsets of the pages comprising the stable working set, and deploys goroutines to bring in the associated pages, in parallel, from the guest memory file. For this and all the following configurations, we make 16 hardware threads available

[3] We omitted the microbenchmarks, MapReduce and Feature Generation because they require a distributed coordinator, and Video Face Detection that is not open source.
[4] The only exception is `video_processing` that uses a Debian image due to a problem with OpenCV installation on Alpine Linux.

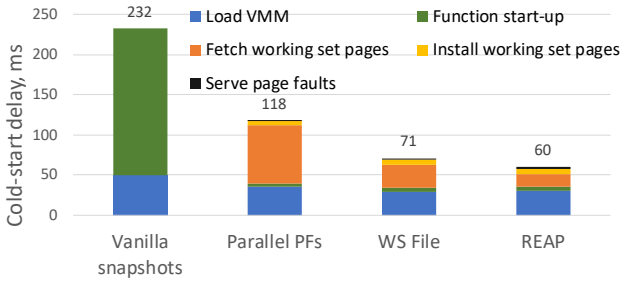


Figure 7: REAP optimization steps.

to vHive-CRI goroutines, managed by the Go runtime. Note that this configuration does *not* use the WS file.

We observe that parallelizing page faults reduces function invocation time by 1.9× (to 118ms) by overlapping I/O processing and exploiting SSD's internal parallelism. Repeating the same calculation as for the baseline, we identify that the orchestrator uses only 130MB/s of SSD bandwidth, which is 15% of the maximum. This design point underlines that achieving high read bandwidth from the SSD is key to efficient page fault processing, and that lowering software overheads by itself is insufficient.

WS file: This design leverages the WS file (Sec 5.1), which enables fetching the entire stable memory working set of a function with a single IO read operation. The difference between this design point and REAP is that the former reads into the OS page cache (which is the default behavior in Linux), whereas REAP bypasses the page cache (§5.2.3). From the figure, one can see that fetching the pages from the WS file can be performed in 29ms, 3.1× faster than through parallel page-sized reads ("Parallel PFs" bar in Fig 7). This design point utilizes 275MB/s of SSD bandwidth.

REAP: The last bar shows the performance of the actual REAP design, as described in Sec 5.2.3, that fetches the working set pages from the WS file and bypasses the OS page cache. As the figure shows, retrieving the working set pages is accelerated by 2× (to 15ms) over the "WS File" design point that does not bypass the page cache. This highlights that while it's essential to optimize for disk bandwidth, software overheads also cannot be ignored. In this final configuration, REAP achieves 533MB/s of SSD bandwidth, which is within 37% of the 850MB/s peak of our SSD.

6.3 REAP on FunctionBench

Fig. 8 compares the cold-start delays of all the functions that we study with the baseline Firecracker snapshots and REAP prefetching. With REAP, all functions' invocations become 1.04-9.7× faster (3.7× on average). The fraction of time for restoring the connection from the orchestrator to the function's gRPC server shrinks by 45×, on average to a mere 4-7ms thanks to the stable working set for this core functionality that is prefetched by REAP.

Although we find that REAP efficiently accelerates the actual function processing, functions with a large number of pages missing from the recorded working set benefit less from REAP. The function processing latency is reduced by 4.6×, on average, for all functions except `video_processing`. During the REAP record phase, the `video_processing` function takes a video fragment of a different aspect ratio than in the prefetch phase that, as we suspect, changes the way OpenCV performs dynamic memory allocation (e.g., uses buffers of different sizes), resulting in a different guest physical memory layout and, hence, different working sets. The orchestrator has to serve the missing pages one-by-one as page faults arise. However, the end-to-end cold-start delay for `video_processing` is nonetheless reduced as the longer function processing time is offset by faster re-connection to the function. We highlight, however, that functions with large inputs or control-flow that differs substantially across invocations may benefit less from REAP.

We repeat the same experiment in the presence of the invocation traffic to 20 warm, i.e., memory resident, functions and observe that the obtained data is within 5% of Fig. 8 results. Also, we measure the efficacy of REAP on the same server but store the snapshots on a 2TB Western Digital WD2000F9YZ SATA3 7200 RPM HDD, instead of the SSD, and observed a 5.4× speed-up (not shown in the figure), on average, with REAP over baseline snapshotting.

6.4 REAP Record Phase

REAP incurs a one-time overhead for recording the trace and the WS files. Upon the first invocation of a function, this one-time overhead increases the end-to-end function invocation time by 15-87% (28% on average). Since most functions that we study have small dynamic inputs, they exhibit relatively small overheads of 12-34%, with `image_rotate` being an outlier with a performance degradation of 87%.

Because of the high speedups provided by REAP on all subsequent invocations of a function, and because the vast majority of functions execute multiple times [53], we conclude that REAP's one-time record overhead is easily amortized.

6.5 REAP Scalability

We demonstrate that REAP orchestrator retains its efficiency in the face of higher load. Specifically, we measure the average time that an instance takes to load from a snapshot and serve one function invocation when multiple independent functions arrive concurrently. We use the the `helloworld` function and consider up to 64 concurrent independent function arrivals. Fig. 9 shows the result of the study, comparing REAP to the baseline snapshots.

Concurrently loading function instances should be able to take advantage of the multi-core CPU and abundant SSD bandwidth (48 logical cores and 850MB/s peak measured SSD bandwidth in our platform). Thus, we expect that as the degree of concurrency increases, the average per-instance latency will not significantly increase thanks to the available parallelism. Indeed, REAP's cold invocation latency stays relatively low, increasing from 70ms to 185ms when the number of concurrent function arrivals goes from 1 to 8. By contrast, the baseline's per-instance invocation time shows a near-linear growth with the number of concurrently-arriving functions. We measure that the SSD throughput that the baseline instances are able to collectively extract is limited to mere 32MB/s for a single instance and 81MB/s for 64 concurrent instances.[5] Compared to the baseline, REAP is able to achieve 118-493MB/s, which explains its lower latency *and* better scalability. Starting

[5]We compute the SSD throughput per instance as the working set size divided by the average loading time.

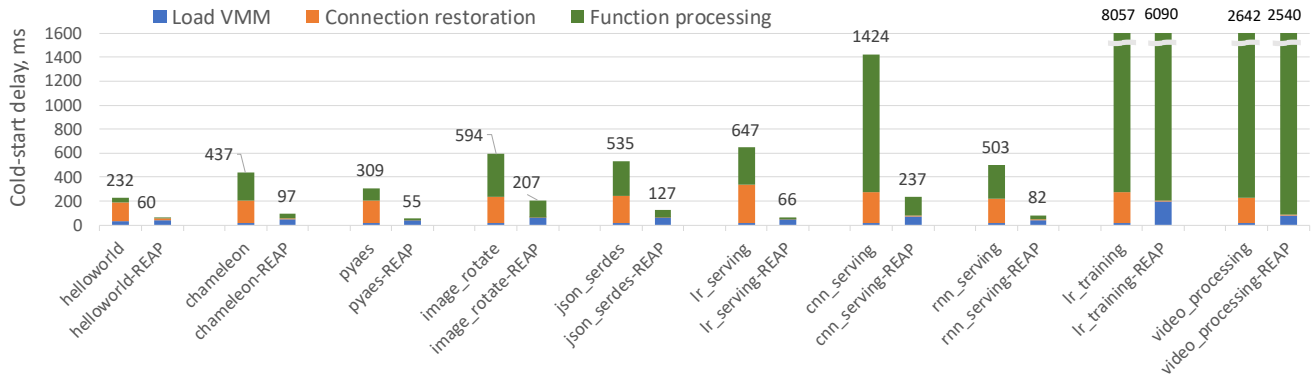


Figure 8: Cold-start delay with baseline snapshots (left bars) and REAP (right bars).

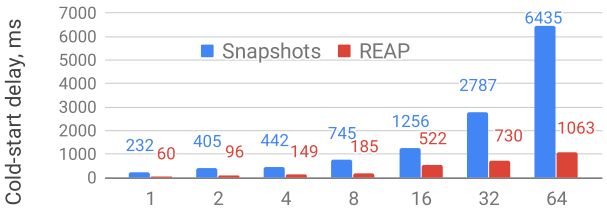


Figure 9: Average instance cold-start delay while sweeping the number of the concurrently loading instances.

from the concurrency degree of 16, REAP becomes disk-bandwidth bound and its scalability is diminished.

7 DISCUSSION

7.1 REAP's Efficiency and Mispredictions

REAP's efficiency depends on how quickly the orchestrator can retrieve the guest memory pages from the snapshot storage and the percentage of the retrieved but unused pages. If the snapshots are located in a remote storage service (e.g., S3 or EBS), the retrieval speed depends on the amount of data to be moved and the latency and bandwidth of the network between the host and the service as well as the latency and bandwidth of the service's internal disks.

REAP reduces both the network and the disk bottlenecks by proactively moving a minimal amount of state. However, REAP may fetch a modest number of pages that are not accessed during processing of some invocations. Our analysis shows that the fraction of mispredicted pages during a cold invocation is close to the "Unique" pages metric, shown in Fig. 5, which is 3-39%. These mispredictions have no impact on system correctness. The cost of these mispredictions is a modest increase in SSD bandwidth usage, proportionate to the fraction of the mispredicted pages.

7.2 Applicability to Real-World Functions

Although REAP is applicable to the vast majority of functions, some functions may not benefit from REAP. For these functions, the additional working set and trace files may not be justified. Prior work shows that 90% of Azure functions are invoked less than once per minute, making these functions the primary target for REAP [53]. Functions that are invoked very rarely (e.g., 3.5% of functions are invoked less frequently than once per week) or more frequently than once per minute (and thus remaining warm) are unlikely to benefit from snapshot-based solutions. Also, REAP is ill-suited for the functions where the first invocation is not representative of future invocations although we do not observe such behavior in our studies. In this pathological case, the orchestrator can easily detect low working set pages usage and either repeat REAP's record phase or fall back to vanilla Firecracker snapshots for future invocations. For detection, the orchestrator could monitor the number of page faults that occur after the working set pages are installed, comparing this number to the number of pages in the working set.

7.3 Security Concerns

Similar to other snapshot techniques, spawning virtual machine clones from the same VM snapshot with REAP has implications for overall platform's security. In a naive snapshotting implementation, these VM clones may have an identical state for random number generators (i.e., poor entropy) and the same guest physical memory layout. The former problem may be addressed at the system level with hardware support for random number generation albeit the user-level random number generation libraries may remain vulnerable [27, 57]. The latter problem may lead to compromised ASLR, allowing the attacker to obtain the information about the guest memory layout. One mitigation strategy could be periodic re-generation of a snapshot (as well as the working set file and the trace file, for REAP). Alternatively, similarly to prior work on after-fork memory layout randomization [40], the orchestrator can dynamically re-randomize the guest memory placement while loading the VM's working set from the snapshot in the record phase of REAP. This would require modifying the guest page tables, with the hypervisor support, according to the new guest memory layout.

8 RELATED WORK

8.1 Open-Source Serverless Platforms

Researchers release a number of benchmarks for serverless platforms. vHive adopts dockerized benchmarks from FunctionBench

that provides a diverse set of Python benchmarks [32, 33]. ServerlessBench contains a number of multi-function benchmarks, focusing on function composition patterns [65]. Researchers and practitioners release a range of systems that combine the FaaS programming model and autoscaling [8–10, 30, 36]. Most of these platforms, however, rely on Docker or language sandboxes for isolating the untrusted user-provided function code that is often considered insufficiently secure in public cloud deployments [16, 53]. Kata Containers [3] and gVisor [28] provide virtualized runtimes that are CRI-compliant but do not provide a toolchain for functions deployment and end-to-end evaluation and do not support snapshotting. Compared to these systems, vHive is a open-source serverless experimentation platform – representative of the production serverless platforms, like AWS Lambda [13] – that uses latest virtualization, snapshotting, and cluster orchestration technologies combined with a framework for functions deployment and benchmarking.

8.2 Virtual Machine Snapshots

Originally, VM snapshots have been introduced for live migration before serverless computing emerged [19, 47]. The Potemkin project propose flash VM cloning to accelerate VM instantiation with copy-on-write memory sharing [63]. Snowflock extends the idea of VM cloning to instantiating VMs across entire clusters, relying on lazy guest memory loading to avoid large transfers of guest memory contents across network [38]. To minimize the time spent in serving the series of lazy page faults during guest memory loading, the researchers explore a variety of working set prediction and prefetching techniques [35, 66–68]. These techniques rely on profiling of the memory accesses *after* the moment a checkpoint was taken and inspecting the locality characteristics of the guest OS' virtual address space. Compared to these techniques, our work shows that serverless functions do not require complex working set estimation algorithms: it is sufficient to capture the pages that are accessed from the moment the vHive-CRI orchestrator forwards the invocation request to the function until the orchestrators receives the response from that function. Moreover, we find that extensive profiling may significantly bloat the captured working set, hence slowing down loading of future function instances, due to the guest OS activity that is not related to function processing.

8.3 Serverless Cold-Start Optimizations

Researchers have identified the problem of slow VM boot times, proposing solutions across the software stack to address it. Firecracker [5] and Cloud Hypervisor [1] use a specialized VMM that includes only the necessary functionality to run serverless workloads, while still running functions inside a full-blown, albeit minimal, Linux guest OS. Dune [15] implements process-level virtualization. Unikernels [17, 34, 41, 43] leverage programming language techniques to aggressively perform deadcode elimination and create function-specific VM images, but sacrifice generality. Finally, language sandboxes, e.g. Cloudflare Workers [22] and Faasm [55], avoid the hardware virtualization costs and offer language level isolation through techniques such as V8 isolates [62] and WebAssembly [4]. Such approaches reduce the start-up costs, but limit the function implementation language choices while providing weaker isolation guarantees than VMs. REAP targets serverless workloads but remains agnostic to the hypervisor and the software that runs inside the VM.

Caching is another approach to reduce start-up latency. Several proposals investigate the idea of keeping pre-warmed, pre-initialized execution environments in memory and ready to process requests. Zygote [24] was introduced to accelerate the instantiation of Java applications by forking pre-initialized processes. The zygote idea has been used for serverless platforms in SOCK [49], while SAND [6] allows the reuse of pre-initialized sandboxes across function invocations. These proposals, though, trade-off low memory utilization for better function invocation latencies. REAP is able to deliver low invocation latencies without occupying extra memory resources when function instances are idle.

Prior work uses VM snapshots for cold-start latency reduction, although snapshots have been initially introduced for live migration and VM cloning before serverless computing emerged [19, 38, 47]. Both Firecracker [5, 58] and gVisor with its checkpoint-restore functionality [28] support snapshoting. The state-of-the-art snapshotting solution, called Catalyzer, improves on gVisor's VM offering three design options for fast VM restoration [26]. Besides the "cold-boot" optimization discussed in §2.3, Catalyzer also proposes "warm-boot" and "sfork" optimizations that provide further performance improvements but require memory sharing across different VMs. In a production serverless deployment, memory sharing is considered insecure and is generally disallowed [5, 59].

Replayable execution aims to minimize the memory footprint and skip the lengthy code generation of the language-based sandboxes by taking a snapshot after thousands of function invocations, exploiting a similar observation as this work – that functions use a small number of memory pages when processing a function invocation [64]. However, when loading a new instance, their design relies on lazy paging similar to other snapshotting techniques [26, 58]. In contrast, our work shows that the working set of the guest memory pages of virtualization-based sandboxes can be captured during the very first invocation, and that all future invocations can be accelerated by prefetching the stable memory working set into the guest memory.

9 CONCLUSION

Optimizing cold-start delays is key to improving serverless clients experience while maintaining serverless computing affordable. Our analysis identifies that the root cause for high cold-start delays is that the state-of-the-art solutions populate the guest memory on demand when restoring a function instance from a snapshot. This results in thousands of page faults, which must be served serially and significantly slow down a function invocation. We further find that functions exhibit a small working set of the guest memory pages that remains stable across different function invocations. Based on these insights, we present the REAP orchestrator that records a function's working set of pages, upon the first invocation of the function, and speeds up all further invocations of the same function by eagerly prefetching the working set of the function into the guest memory of a newly loaded function instance.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers and the paper's shepherd, Professor James Bornholt, as well as Professors Antonio Barbalace and James Larus, the members of the EASE lab at the University of Edinburgh, the members of the DCSL and the VLSC lab at EPFL for the fruitful discussions and for their valuable feedback on this work, Theodor Amariucai and Shyam Jesalpura for helping with the release of the vHive software. The authors would like to specially thank Firecracker and Containerd developers at Amazon Web Services, particularly Adrian Catangiu and Kazuyoshi Kato, for their guidance in setting up the Firecracker and Containerd components. This research was supported by the Arm Center of Excellence at the University of Edinburgh, Microsoft Swiss JRC TTL-MSR project at EPFL, and an IBM PhD fellowship.

REFERENCES

[1] [n.d.]. Cloud Hypervisor. Available at https://github.com/cloud-hypervisor.
[2] [n.d.]. gRPC: A High-Performance, Open Source Universal RPC Framework. Available at https://grpc.io.
[3] [n.d.]. Kata Containers. Available at https://katacontainers.io.
[4] [n.d.]. WebAssembly. Available at https://webassembly.org.
[5] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*. 419–434.
[6] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. 923–935.
[7] Amazon. [n.d.]. A Demo Running 4000 Firecracker MicroVMs. Available at https://github.com/firecracker-microvm/firecracker-demo.
[8] Apache. [n.d.]. OpenWhisk. Available at https://openwhisk.apache.org/.
[9] The Fission Authors. [n.d.]. Fission: Open Source, Kubernetes-Native Serverless Framework. Available at https://fission.io.
[10] The Fn Project Authors. [n.d.]. Fn Project. Available at https://fnproject.io.
[11] The Istio Authors. [n.d.]. Istio. Available at https://istio.io.
[12] The Knative Authors. [n.d.]. Knative. Available at https://knative.dev.
[13] AWS re:Invent. 2019. A Serverless Journey: AWS Lambda Under the Hood.
[14] Baidu. [n.d.]. The Application of Kata Containers in Baidu AI Cloud. Available at https://katacontainers.io/collateral/ApplicationOfKataContainersInBaiduAICloud.pdf.
[15] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*. 335–348.
[16] Ricardo Bianchini. [n.d.]. Serverless in Seattle: Toward Making Serverless the Future of the Cloud. Available at https://acmsocc.github.io/2020/keynotes.html.
[17] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the 2020 EuroSys Conference*. 32:1–32:15.
[18] CBINSIGHTS. [n.d.]. Why Serverless Computing Is The Fastest-Growing Cloud Services Segment. Available at https://www.cbinsights.com/research/serverless-cloud-computing.
[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*.
[20] Cloud Native Computing Foundation. [n.d.]. CRI-O: Lightweight Container Runtime for Kubernetes. Available at https://cri-o.io.
[21] Containerd. [n.d.]. An Industry-Standard Container Runtime with an Emphasis on Simplicity, Robustness and Portability. Available at https://containerd.io.
[22] CouldFlare. [n.d.]. CloudFlare Workers. Available at https://workers.cloudflare.com/.
[23] Daniel Krook. [n.d.]. Five Minute Intro to Open Source Serverless Development with OpenWhisk. Available at https://medium.com/openwhisk/five-minute-intro-to-open-source-serverless-development-with-openwhisk-328b0ebfa160.
[24] Android Developers. [n.d.]. Overview of Memory Management. Available at https://developer.android.com/topic/performance/memory-overview.
[25] Docker. [n.d.]. Use the Device Mapper Storage Driver. Available at https://docs.docker.com/storage/storagedriver/device-mapper-driver.
[26] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 467–481.
[27] Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael M. Swift. 2014. Not-So-Random Numbers in Virtualized Linux and the Whirlwind RNG. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*. 559–574.
[28] Google. [n.d.]. gVisor. Available at https://gvisor.dev.
[29] Google Cloud. [n.d.]. Configuring Warmup Requests to Improve Performance. Available at https://cloud.google.com/appengine/docs/standard/python/configuring-warmup-requests.
[30] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.
[31] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*. 158–164.
[32] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD)*. 502–504.
[33] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*. 477.
[34] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*. 61–72.
[35] Thomas Knauth and Christof Fetzer. 2014. DreamServer: Truly On-Demand Cloud Services. In *Proceedings of the 7th ACM International Systems and Storage Conference (SYSTOR)*. 9:1–9:11.
[36] Kubeless. [n.d.]. Kubeless: The Kubernetes Native Serverless Framework. Available at https://kubeless.io.
[37] Kubernetes. [n.d.]. Production-Grade Container Orchestration. Available at https://kubernetes.io.
[38] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 2009 EuroSys Conference*. 1–12.
[39] Linux programmer's manual. [n.d.]. Userfaultfd. Available at https://man7.org/linux/man-pages/man2/userfaultfd.2.html.
[40] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*.
[41] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*. 461–472.
[42] Linux man page. [n.d.]. fio. Available at https://linux.die.net/man/1/fio.
[43] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. 218–233.
[44] Market Reports World. 2019. Serverless Architecture Market by End-Users and Geography - Global Forecast 2019-2023. Available at https://www.marketreportsworld.com/serverless-architecture-market-13684687.
[45] Microsoft. 2019. Azure Functions. Available at https://azure.microsoft.com/en-gb/services/functions.
[46] MinIO. [n.d.]. Kubernetes Native, High Performance Object Storage. Available at https://min.io.
[47] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. 2005. Fast Transparent Migration for Virtual Machines. In *USENIX Annual Technical Conference*. 391–394.
[48] Goncalo Neves. [n.d.]. Keeping Functions Warm – How To Fix AWS Lambda Cold Start Issues. Available at https://serverless.com/blog/keep-your-lambdas-warm.
[49] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. 57–70.
[50] OpenNebula. [n.d.]. OpenNebula + Firecracker: Building the Future of On-Premises Serverless Computing. Available at https://opennebula.io/opennebula-firecracker-building-the-future-of-on-premises-serverless-computing.
[51] Allison Randal. 2020. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. *ACM Comput. Surv.* 53, 1 (2020), 5:1–5:31.
[52] Samuel Karp. [n.d.]. Deep Dive into Firecracker-Containerd. Available at https://speakerdeck.com/samuelkarp/deep-dive-into-firecracker-containerd-

re-invent-2019-con408.
[53] Mohammad Shahrad, Rodrigo Fonseca, Iñigo 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 *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 205–218.
[54] Mikhail Shilkov. [n.d.]. Serverless: Cold Start War. Available at https://mikhail.io/2018/08/serverless-cold-start-war.
[55] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 419–433.
[56] Bernd Strehl. [n.d.]. Lambda Serverless Benchmark. Available at https://serverless-benchmark.com.
[57] The Firecracker Authors. [n.d.]. Entropy for Clones. Available at https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/random-for-clones.md.
[58] The Firecracker Authors. [n.d.]. Firecracker Snapshotting. Available at https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/snapshot-support.md.
[59] The Firecracker Authors. [n.d.]. Production Host Setup Recommendations. Available at https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md.
[60] The Firecracker-Containerd Authors. [n.d.]. Firecracker-Containerd. Available at https://github.com/firecracker-microvm/firecracker-containerd.
[61] The Linux Foundation Projects. [n.d.]. Open Container Initiative. Available at https://opencontainers.org.
[62] V8. [n.d.]. Isolate Class Reference. Available at https://v8docs.nodesource.com/node-0.8/d5/dda/classv8_1_1_isolate.html.
[63] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2005. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. 148–162.
[64] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the 2019 EuroSys Conference*. 39:1–39:16.
[65] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with ServerlessBench. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*. 30–44.
[66] Irene Zhang, Tyler Denniston, Yury Baskakov, and Alex Garthwaite. 2013. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*. 1–12.
[67] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. 2011. Fast Restore of Checkpointed Memory using Working Set Estimation. In *Proceedings of the 7th International Conference on Virtual Execution Environments (VEE)*. 87–98.
[68] Jun Zhu, Zhefu Jiang, and Zhen Xiao. 2011. Twinkle: A Fast Resource Provisioning Mechanism for Internet Services. In *Proceedings of the 2011 IEEE Conference on Computer Communications (INFOCOM)*. 802–810.