# Benchmarking, Analysis, and Optimization of Serverless Function Snapshots
## Extended Abstract

Dmitrii Ustiugov[†], Plamen Petrov[†], Marios Kogias[*‡], Edouard Bugnion[+], and Boris Grot[†]

[†]University of Edinburgh, [‡]Microsoft Research, [+]EPFL

## 1. Motivation

In recent years, serverless computing has emerged as a rapidly-growing cloud service and deployment model, increasing from 12% adoption in 2017 to 21% adoption in 2018 [6, 15]. 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 strictly determined by the load on that specific function, and could range from zero to thousands of instances running concurrently. 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 whereby customers are charged only for the time spent executing their requests – a marked departure from conventional virtual machines (VMs) hosted in the cloud that are billed for their uptime 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. The reason why such a high degree of colocation is possible is that 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 the functions execute for less than 10 seconds [18].

Because of their short execution time, booting a function (i.e., *cold start*) is an overwhelmingly expensive operation latency-wise, 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. Similarly, customers also have an incentive to avoid cold starts because of their high impact on latency [17]. As a result, both cloud vendors and their customers prefer to keep function instances memory-resident (i.e., *warm*) [10, 16, 17]. However, keeping idle function instances alive wastefully occupies precious main memory, which accounts for 40% of a modern server's typical capital cost [1]. With serverless providers instantiating thousands of function on a single server [1, 2], the memory footprint of keeping all instances warm can reach into hundreds of GBs.

## 2. Limitations of the State of the Art

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 full architectural 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.

Snapshots are attractive because they provide zero main memory consumption during the periods of a function's inactivity and enable short 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, called Catalyzer [8], showed that snapshot-based restoration in the context of gVisor [9] virtualization technology can be performed in 10s-100s of milliseconds. 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 loads the minimum amount of state necessary to resume VM execution. Then, it maps the plain guest-physical memory file as a file-backed virtual memory region. Crucially, the guest-physical memory of the VM is not populated with memory contents, which still 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 that must be served by the host OS. These page faults occur serially on the critical path of function execution and significantly increase the runtime of a function loaded from a snapshot.

## 3. Key Insights

To understand serverless system operation, we introduce vHive, an open-source framework for serverless experimentation across the whole serverless stack.[1] Using vHive, we study cold-start latency of functions from the FunctionBench suite [11, 12], as well as their memory footprint and spatio-temporal locality characteristics. Functions run inside Firecracker MicroVMs [1] as part of the industry-standard Containerd infrastructure [3, 7]. In our experiments, we evaluate a state-of-the-art baseline where the function is restored from a snapshot on a local SSD [4, 8]. We simulate cold function invocations by flushing the page cache of the host OS before each measurement.

[*]The work was done when the author was at EPFL.

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

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 when restoring from a snapshot 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., reading files and loading libraries).

Our second observation is that the execution time of a function restored from a snapshot is dominated by serving of 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 guest-memory 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, even when the function is invoked with different inputs (e.g, different images to be rotated).

## 4. Main Artifacts

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 deep distributed serverless stack. vHive integrates open-source production-grade components from the leading serverless providers, namely Amazon Firecracker hypervisor [1], Containerd [7], Kubernetes [13], and Knative [5], and a toolchain for functions deployment and analysis.

Our analysis, summarized in §3, shows that a function's working set of guest memory pages is compact and stable across different invocations of the same function. Leveraging these insights, we introduce Record-and-Prefetch (REAP) – an orchestrator 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 compact trace of guest-physical pages that comprise the working set of a function, 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 working set into main memory (by reading the contents of the working set file with a single disk read) and eagerly installs it into the guest's memory space.

To implement REAP in vHive, we add minimal changes to the Firecracker hypervisor to register the guest-physical memory region for handling page faults in userspace, using the stock Linux mechanism [14]. This allows the orchestrator to handle a function instance's page faults on behalf of the host OS. Our implementation of the REAP orchestrator is independent of the underlying serverless infrastructure and requires no changes to the OS kernel.

Our evaluation demonstrates that REAP is able to eliminate up to 97% of the pages faults, as compared to baseline snapshots, by taking full advantage of SSD bandwidth and by carefully avoiding software overheads in the kernel. As a result, REAP slashes cold-start latency of serverless functions by an average of 3.7×.

## 5. Key Results and Contributions

- We release vHive, an open-source framework for serverless experimentation, comprising production-grade components from the leading serverless providers to enable innovation in serverless systems across their deep and distributed software stack.
- 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 the REAP orchestrator, which uses a record-and-prefetch mechanism to eagerly install 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 any memory overheads or memory sharing across function instances.

## 6. Why ASPLOS

Our work enables cross-stack innovation in serverless systems by releasing vHive, an open-source framework for serverless experimentation, comprising the leading industrial components. Using vHive, we show that the state-of-the-art approach of restoring a serverless function from a snapshot involves inefficiencies both in system software and in hardware (disk bandwidth). We design REAP that achieves fast function start-up times by fully utilizing SSD bandwidth and overcoming software inefficiencies of baseline snapshotting.

## 7. Citation for Most Influential Paper Award

For unlocking innovation across deep and distributed serverless stack with vHive, an open-source framework for serverless experimentation, and showcasing the framework's utility with a detailed analysis of the state-of-the-art snapshot-based serverless systems that led to a simple yet highly effective snapshot load optimization.

## 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 *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[2] Amazon. A demo running 4000 Firecracker MicroVMs. Available at https://github.com/firecracker-microvm/firecracker-demo.

[3] Amazon. Firecracker-containerd. Available at https://github.com/firecracker-microvm/firecracker-containerd.

[4] Amazon. Firecracker snapshotting. Available at https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/snapshot-support.md.

[5] The Knative Authors. Knative. Available at https://knative.dev.

[6] CBINSIGHTS. Why serverless computing is the fastest-growing cloud services segment. Available at https://www.cbinsights.com/research/serverless-cloud-computing.

[7] Containerd. An industry-standard container runtime with an emphasis on simplicity, robustness and portability. Available at https://containerd.io.

[8] 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 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020.

[9] Google. gVisor. Available at https://gvisor.dev.

[10] Google Cloud. Configuring warmup requests to improve performance. Available at https://cloud.google.com/appengine/docs/standard/python/configuring-warmup-requests.

[11] Jeongchul Kim and Kyungyong Lee. FunctionBench: A suite of workloads for serverless cloud function service. In *Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD)*, 2019.

[12] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless FaaS. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*, 2019.

[13] Kubernetes. Production-grade container orchestration. Available at https://kubernetes.io.

[14] Linux programmer's manual. Userfaultfd. Available at https://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[15] Market Reports World. Serverless Architecture Market by End-Users and Geography - Global Forecast 2019-2023, 2019. Available at https://www.marketreportsworld.com/serverless-architecture-market-13684687.

[16] Microsoft. Azure functions, 2019. Available at https://azure.microsoft.com/en-gb/services/functions.

[17] Goncalo Neves. Keeping functions warm – how to fix AWS Lambda cold start issues. Available at https://serverless.com/blog/keep-your-lambdas-warm.

[18] Mohammad Shahrad, Rodrigo Fonseca, Iñigo 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 *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.