

FaaSnap: FaaS Made Fast Using Snapshot-based VMs

Lixiang Ao
UC San Diego
liao@cs.ucsd.edu

George Porter
UC San Diego
gmporter@cs.ucsd.edu

Geoffrey M. Voelker
UC San Diego
voelker@cs.ucsd.edu

Abstract

FaaSnap is a VM snapshot-based platform that uses a set of complementary optimizations to improve function cold-start performance for Function-as-a-Service (FaaS) applications. Compact loading set files take better advantage of prefetching. Per-region memory mapping tailors page fault handling depending on the contents of different guest VM memory regions. Hierarchical overlapping memory-mapped regions simplify the mapping process. Concurrent paging allows the guest VM to start execution immediately, rather than pausing until the working set is loaded. Altogether, FaaSnap significantly reduces guest VM page fault handling time on the critical path and improves overall function loading performance. Experiments on serverless benchmarks show that it reduces end-to-end function execution by up to 3.5x compared to state-of-the-art, and on average is only 3.5% slower than snapshots cached in memory. Moreover, we show that FaaSnap is resilient to changes of working set and remains efficient under bursty workloads and when snapshots are located in remote storage.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Software and its engineering → Virtual machines**.

Keywords: cloud computing, serverless, FaaS, virtualization, snapshots, cold starts, caching

ACM Reference Format:
Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS Made Fast Using Snapshot-based VMs. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3492321.3524270

1 Introduction

As an emerging cloud computing offering, Function-as-a-Service, or FaaS, has gained popularity for applications including IoT and API backends [19, 35], machine learning [5], and data analytics [3, 12, 13, 27]. Its simple "function" interface, minimal management, considerable scalability, and fine-grained pay-as-you-go billing model are attractive features to users. Users only need to focus on the application code itself while the burdens of managing servers and networks, provisioning capacity, and reclaiming resources are shifted to the cloud provider.

Since function invocations are often short compared to long-running servers in virtual machine instances, the performance overhead of creating the execution environments for functions can be a significant component of overall execution time. In particular cold starts, during which the cloud provider prepares the isolation sandboxes (either VMs or containers) and runtime environments required by the FaaS applications, have received significant attention. Both academia and industry have explored cold start mitigations using techniques like lightweight sandboxing [1, 22, 30], sharing of resources among instances [2, 10, 32], or cloning pre-initialized environments in memory [4, 24, 34].

Virtual machine snapshots are a recent method developed to mitigate cold starts by restoring the guest VM to an existing warm initialized state to avoid the time-consuming steps of cold start like initializing runtimes and loading libraries [11]. Existing VM snapshot methods apply lazy loading of guest memory to avoid loading the whole guest memory when starting guest VMs. The memory pages are loaded from disk on-demand when accessed by the guest. However, the guest page accesses exhibit low spacial locality, leading to many major page faults and scattered disk reads that add significant overhead and slow down function execution [33].

Focusing on the active memory working set at the time when a snapshot is taken is a promising direction for improving performance. By prefetching the working set when restoring a snapshot for a function invocation, slow major page faults and disk reads can be avoided during execution. Zhang et al. [37] explore this approach in the context of traditional virtual machine environments, scanning the access bits of page table entries to determine the recent working set of a guest VM at checkpoint time to reduce page faults after restoring. REAP [33] is the state-of-the-art for using working sets to accelerate snapshot restoring for FaaS. REAP assumes memory access is stable across function invocations and prefetches a compact representation of the working set of previous invocations when serving new ones. However, as we show, this assumption does not hold when the input data differs significantly from previous invocations, leading to page accesses outside of the working set that slow down



This work is licensed under a Creative Commons Attribution International 4.0 License.

EuroSys '22, April 5–8, 2022, RENNES, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3524270

the invocation. In addition, the guest VM has to wait until the entire working set has been restored to start, a problem especially troublesome for functions with large working sets.

Analyzing the snapshot behavior of FaaS functions, we present several notable observations on guest VM snapshots for FaaS. First, there is substantial differences in performance between major page faults and minor page faults, and the host OS page cache can be used to reduce the cost of major page faults. Second, due to the variance of function inputs and execution flows, the working set can change dramatically. Pages used by previous invocations should be treated as a reference for future invocations, and restoring a snapshot should efficiently handle invocations whose working set substantially differs. Third, a semantic gap between host pages and guest pages leads to unnecessary disk reads that slow down functions.

Based on these observations we propose FaaSnap, an efficient VM snapshot loading method that integrates several new techniques to reduce the cost of restoring guest VM snapshots and thereby improve overall FaaS invocation performance. Concurrent paging and working set groups avoid blocking function invocations while loading the working set, and opportunistically prevent slow disk reads by taking advantage of the host OS page cache. Host page recording relaxes the working set criteria to better tolerate variance in pages used by future invocations. Per-region mapping bridges the semantic gap between the host and the guest by handling memory pages differently based on their types. Finally, FaaSnap uses loading sets, a more compact working set definition, and an efficient layout of the loading set file that improves working set prefetching by removing unnecessary pages and consolidating disk reads.

We implement FaaSnap based on Firecracker, a lightweight VM tailored for serverless workloads. FaaSnap improves function execution by up to 3.5× compared to REAP snapshots. It is on average only 3.5% slower than snapshots cached in memory across a wide range of FaaS functions. FaaSnap is resilient to changes of working set across invocations, and remains efficient under bursty workloads and when snapshots are located in remote storage.

FaaSnap is open-source software and is accessible at https://github.com/ucsdsysnet/faasnap.

2 Background and Related Work

Function-as-a-Service, or FaaS, is a form of serverless computing. FaaS provides a high-level "function" abstraction—request handler code—so that users do not need to explicitly manage resources like servers and networks. Because of its popularity, it is sometimes simply referred to as serverless computing and, in this paper, we use the terms FaaS and serverless interchangeably. AWS Lambda, Azure Functions, and Google Cloud Functions are examples of popular FaaS commercial offerings. Several open-source projects are also gaining popularity, including OpenWhisk [26], OpenFaaS [25], and Knative [17].

Compared to traditional cloud offerings like IaaS, FaaS provides more convenient computing resources. Users simply provide the function code for handling requests, and the service ensures that the code will be invoked when a request arrives. FaaS applications depend on underlying computing and networking resources including isolation sandboxes, virtual networks, operating systems and runtimes, which are all managed and provided by the cloud provider.

2.1 Cold start problem

If the environment for a FaaS application does not exist when a function is invoked, the FaaS platform needs to initialize the environment for invoking the function, a process called cold start. The initialization steps include creating the virtual network, booting the isolation sandbox (usually a VM or a container), installing the function code, initializing the runtime, etc. The initialization steps take from several seconds up to minutes. A cold start is especially costly for FaaS since more than 50% of all invocations are less than 1 second, and 75% are less than 3 seconds [29]. The long cold start process can negatively impact both the performance of FaaS applications and system capacity. Therefore, many platforms keep the environment alive after an invocation finishes using an optimization called warm start. The warm state of the environment, including the runtime, loaded libraries, and accessed files, are kept in memory or in the page cache. In subsequent invocations, the environment and warm state are reused to accelerate function invocation.

However, there is a cost for keeping warm resources alive. Too many idle environments consume memory, reducing overall system capacity and throughput for invoking functions. Moreover, experience from large providers, such as the real-world traces from Azure [29], reveals that only a small portion of functions are invoked frequently. Less than half of the functions are invoked every hour, and less than 10% are invoked every minute. Cloud providers only keep the environment alive for a short period of time after an invocation is finished to minimize wasting resources; AWS Lambda keeps functions warm for 15-60 minutes [14]. As a result, invocations of hot functions are likely to have warm starts, while less frequent functions are prone to cold starts.

The cold start overheads can be generally divided into two parts, booting and initializing state. Booting a VM takes at least several seconds. Initializing memory state includes starting runtime, installing function code, loading libraries, etc., which can take seconds to minutes.

2.2 Booting time mitigation

Containers [7, 20, 31] are a more lightweight isolation mechanism than virtual machines. Containers use kernel abstractions like cgroups and namespaces to provide isolation among

instances. The overhead of starting a container can be the same order of magnitude as starting a process.

However, in multi-tenant clouds like AWS and Azure, VMs are used as the sandboxing mechanism. VMs have a simpler, more stable guest-host interface than container's syscall interface and is considered more secure [22]. The downside is that VMs are traditionally designed for general-purpose guests. Full-fledged heavyweight VMs lead to long booting times for cold starts.

Unikernels have been explored to accelerate VM booting time [21]. Unikernels trim the guest kernel by customizing it for the application and eliminate the kernel-application boundary. LightVM [22] reduces the booting time to less than 10 ms by using a unikernel and a Xen VMM specialized for serverless computing. Faasm [30] uses lightweight language-based isolation to avoid the cost of virtualization.

Firecracker [1] is an open-source virtual machine monitor from Amazon and is used by AWS Lambda and other cloud services as a security sandbox. It uses a lightweight design that is tailored to serverless computing. The device emulation is minimized and BIOS is excluded to improve performance and reduce resource consumption. Firecracker can boot an unmodified Linux kernel in 125 ms.

2.3 Preparing memory state

In FaaS, not only does the isolation sandbox need to be created, the initialization of the OS, runtime, and libraries also takes significant time. Du et al. [9] report that the majority of cold start time in Google's gVisor is spent on the initialization of runtimes.

Copying existing memory state that has already been initialized is a method to skip the initialization step. Potemkin [34] proposed flash VM cloning and copy-on-write delta virtualization to quickly make copies of existing VMs. Snowflock [18] enables the cloning of VMs to remote hosts, lazily transferring guest pages to remote hosts when handling guest page faults. SOCK [24] is a container-based isolation system that creates copies of existing processes that have already initialized runtimes and libraries to skip the initialization step. SEUSS [4] takes in-memory snapshots of a unikernel guest and removes redundant execution paths, including initializing runtime and importing libraries, by serving invocations from the memory snapshots.

These systems require an existing VM or container for the function to exist in memory. This requirement is not always feasible in FaaS due to its memory resource consumption, especially for less frequently invoked functions.

2.4 Snapshots

Snapshot and restore is another method for avoiding booting and initialization overheads. The VM or container in-memory state can be saved to a file, and later the state can be restored from the file, skipping the initialization steps.

gVisor [15], a sandboxed container runtime by Google, supports checkpoint and restore. The latency of restoring a gVisor container can be a few hundreds of milliseconds due to the loading of guest memory and recovering guest kernel state. Catalyzer [9] uses optimizations including lazy memory loading and restoring some kernel state out of the critical path. The memory pages are read on-demand, reducing the initial wait time.

Firecracker recently introduced a snapshot and restore feature [11]. A Firecracker snapshot can be restored in a few milliseconds, a drastic reduction from typical cold start latencies. A Firecracker snapshot includes a snapshot file that stores the state of the VM like virtual devices and CPU registers as well as a memory file, which is the copy of the entire guest physical memory.

When Firecracker restores a snapshot, it loads and restores the VM state and maps the guest memory file to the VMM memory region that is provided to KVM (the virtual machine monitor in the host system) as the guest memory. Similar to Catalyzer, the guest memory pages are then loaded by the host on-demand when guest page faults happen.

While this approach to snapshot and restore improves performance, the cold start problem is still not entirely solved. Although the VM state can be restored and guest memory initialized in just milliseconds, to do any useful work, the guest needs to access at least a few thousand memory pages. Lazy restore only loads pages when the guest VM accesses them and creates page faults. As motivated by REAP [33], and our experiments also show in Section 3, a simple `hello-world` function takes more than 200 ms to execute using Firecracker snapshots, compared to a warm VM which finishes within 4 ms. Guest page faults are slow because the pages need to be read from disk, and the reads are small and scattered.

2.5 Working sets

Zhang et al. [37] proposed accelerating the lazy restore of VMs by eagerly prefetching working set pages. The working set is estimated by detecting which pages have been accessed recently before a snapshot. Their approach continuously scans the access bits in the page table to obtain the working set. When creating a snapshot, the working set pages are stored in a separate file. In the restore step, the working set pages are loaded sequentially and copied to the guest memory before the guest VM starts. This optimization decreases the number of future guest VM page faults. Halite [36] merges pages that are accessed together into locality groups. During restoring of the VM, when any page in the group is accessed, the whole group is prefetched.

REAP [33] is the state-of-the-art in optimizing serverless performance using working sets. Their observation is that serverless functions tend to access a stable set of pages across invocations. The idea is to prefetch pages accessed from previous invocations when an invocation starts. REAP uses `userfaultfd`, a kernel feature that allows user-level

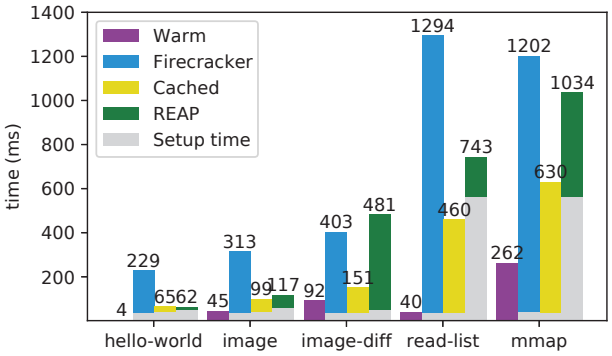


Figure 1. Time breakdown of function invocations. Primary color bars are function invocations. Gray bars are for VM setup, including starting the VMM, connecting virtual devices, restoring VM CPU state, etc.

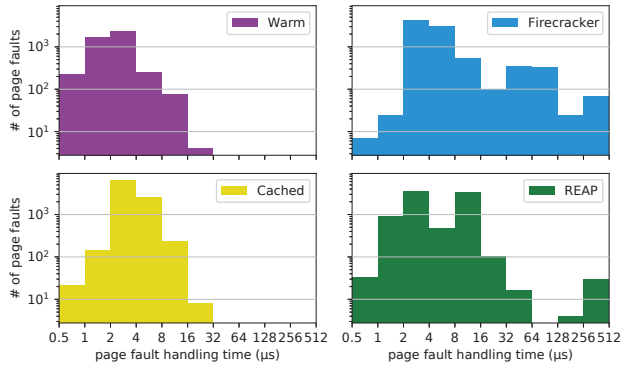


Figure 2. Distributions of page fault handling time for `image-diff` under different settings. Both axes are log-scale.

programs to handle page faults. It records the pages accessed in the first invocation into a working set file. In subsequent invocations, the working set pages are prefetched and installed in the guest memory, reducing the number of later page faults and disk reads. The working set pages are saved to a compact working set file and can be fetched in a single batch read, avoiding the cost of scattered page reads.

REAP works well for some workloads. However, as we show in Section 3, its performance is sensitive to changes in the working set. Invocation performance decreases when the working set differs significantly from the previous invocations because of change of input data, or when large amounts of anonymous pages are allocated in the guest. Both cases are common in real-world serverless functions.

3 Snapshot Analysis

To understand the overheads and challenges in snapshot-based function invocations, we measure several aspects of snapshot restoring of Firecracker VMs and REAP. We conduct our experiments using our FaaSnap platform, which we describe in more detail in Section 4.1.

3.1 Measurements

We measure the invocation of the following functions: a trivial `hello-world` function that replies with a "hello" string; a `read-list` function that reads every page of a large (512 MB) existing Python list; an `mmap` function that memory maps a large (512 MB) anonymous memory region and writes to every page of the region; an `image` function from FunctionBench [16], a comprehensive FaaS benchmark, that processes a JPEG image. `image-diff` is the same as `image` except it uses different inputs across invocations. In real-world deployments, inputs are most likely different across invocations.

We measure each function under four settings. Warm executes a function using a warm VM cached in memory that served a previous invocation. Firecracker executes a function by restoring a VM from a standard Firecracker snapshot memory file that was recorded and saved after a VM served a previous invocation. Cached executes a function similarly to Firecracker, but the snapshot memory file is loaded into the page cache so that there is no disk read overhead. While not practical in real-world deployments, it is a useful reference point for comparing snapshotting systems. We integrated REAP into our platform as an optional setting. REAP executes a function using a snapshot memory file together with a working set file created from a previous invocation, and it loads the entire working set file into memory immediately before executing the function.

For the time breakdown tests, we measure the time of the setup and execution steps for all functions. For additional insight, we also measure the time the guest VM spent handling page faults (`kvm_mmu_page_fault` kernel function) specifically for the `image-diff` function. We use bpftrace [28], a tracing tool based on eBPF, for the page fault measurements.

The host is an AWS EC2 `c5d.metal` instance with an Ubuntu Linux kernel version of 5.4.0. The disk is an NVMe SSD with measured maximum throughput of 1589 MB/s and 285,000 IOPS. The guest VM is configured with 2 GB of memory and 1 VCPU. Guest VMs use Debian Linux with a 4.14 kernel.

3.2 Time breakdowns

The time breakdown results are shown in Figure 1. The gray bars show the time taken to set up the VM, including starting the VMM, restoring virtual devices and CPU states, and for REAP, loading working sets. The primary color bars show the time to invoke the functions. As expected, Warm outperforms all other settings. The `hello-world` function completes in 4 ms, much faster than all other settings. Warm is so fast because it does not have the overheads of restoring and setting up the VM, and it already has most of the guest VM state in physical memory.

Among the snapshot-based systems, Firecracker is the slowest. Firecracker uses OS on-demand paging, and a page is only read when accessed by the guest or prefetched when nearby pages are accessed. Small reads on the disk are slow (relative to memory) even on high performance NVMe SSDs. Cached has the best performance for all the functions except `hello-world`. Its contrast with Firecracker highlights the importance of avoiding costly disk reads in page fault handling. The function invocation times of Cached for the `image` and `image-diff` functions are close to that of Warm. For the `read-list` and `mmap` functions, however, Cached is significantly slower because, although Cached avoids all the major page faults that read from disk, minor page faults are still needed to install the page table entries for the pages in the host OS page cache.

REAP performs well for the `hello-world` and `image` functions, where the function is supplied with the same input data as the previous invocation. The invocation time is similar to that of Cached and Warm. However, in `image-diff` where the input data differs for the second invocation, its performance degrades. REAP relies on the VM using a stable set of pages across invocations. When the pages used are significantly different, it handles the pages that are not in the working set file at user level, reducing performance. The `read-list` and `mmap` functions have a large working set. As a result, the setup step takes a long time to load and install the working set. Once installed, though, invocation becomes fast. The `read-list` and `mmap` invocation steps are faster for REAP than Cached because the pages are installed into the host page table by REAP's `userfaultfd` handler.

The `mmap` function allocates anonymous memory in the guest. However, because the whole guest memory is mapped to the host memory file, the host does not know the guest is allocating anonymous memory. As a result, allocating guest memory causes disk reads on the host, which is much slower than allocating from anonymous memory on the host.

3.3 Page fault behavior

The distribution of page fault handling times is shown in Figure 2. We test the `image-diff` function invoked on the four systems. The x ticks represent times spent handling a page fault, and each bar between ticks counts the number of page faults whose time falls into that interval. Note that both axes are in log scale.

Warm has around 4,000 page faults, while all the snapshot-based systems have around 9,000. Warm VMs have many of their pages already loaded into physical memory, and the page faults are caused by accessing new pages not touched in the first invocation. Since warm VMs are booted from VM images, and the guest memory region is mapped to host anonymous memory, the warm page faults are quickly handled using anonymous memory, which is faster than file-backed mappings that go through the page cache layer. The average time is 2.5 microseconds, and more than 90% of the warm page faults take less than 4 microseconds. The total time of handling all the page faults is 12 ms.

Cached handles more than 90% of the page faults in less than 8 microseconds, and the average time is 3.7 microseconds. All the Cached page faults are minor page faults that are served by the page cache. The time to handle page faults for Cached takes slightly longer than that of Warm because it has to access the page cache layer. The total page fault handling time is 35 ms.

Firecracker is the slowest among the four systems, with an average page fault time of 13.3 microseconds. Nearly 9% of the page faults take more than 32 microseconds, which are slow major page faults that read from disk. When handling a page fault from disk, the `readahead` mechanism in the host kernel fetches pages near the faulting page into the page cache to reduce future disk reads. The page faults shorter than 32 microseconds are mostly minor page faults served from the page cache, and they show a distribution similar to that of Cached. The total page fault handling time is 120 ms.

REAP page faults have an interesting distribution. Pages from the working set file are installed into the host page table by `userfaultfd` at the beginning of the invocation. Page faults on these pages are processed in less than 4 microseconds since the host page table entries already exist. Page faults outside of the working set causes the userspace `userfaultfd` process to read from the original memory file. Depending on whether the page in the memory file has been prefetched into the page cache, the handling can be relatively fast (8–64 microseconds) or slow (>128 microseconds). Userspace `userfaultfd` adds an overhead of several microseconds to each page fault outside of the working set. The average page fault time is 6.7 microseconds, and the total page fault handling time is 56 ms. Although REAP handles pages faults faster than Firecracker, its execution time is longer than Firecracker. With REAP, the guest cannot immediately resume after a page fault is handled, causing context switches that slow down guest execution.

3.4 Summary

We make several observations from these experiments: (1) The performance difference between Firecracker and Cached, and the time difference between handling minor and major page faults, highlight the benefits of caching pages in memory. The OS page cache can play an important role in accelerating VM page faults; (2) The working set of a function can change dramatically due to changes in input data or function execution flow. Therefore the page accesses in previous invocations should be treated as a reference, and restoring a snapshot should efficiently handle invocations whose working set substantially differs; (3) The anonymous page allocation in the guest memory is translated to unnecessary file-backed page fault on the host because of the semantic gap between the guest and the host.

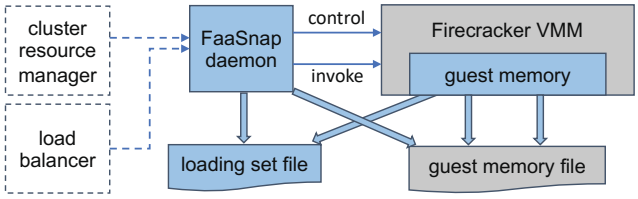


Figure 3. High level system architecture. Blue elements are FaaSnap components, and gray elements are existing Firecracker components. Dashed components are expected to interact with the FaaSnap daemon in real-world FaaS deployments, but are not needed in our use of FaaSnap.

4 FaaSnap

Based on the observations from Section 3, we propose FaaSnap, a snapshot loading mechanism that handles real-world FaaS snapshot paging more efficiently. We introduce several techniques and optimizations to improve various aspects of snapshot loading. We first describe the system design, and then detail each of the optimization techniques in turn.

4.1 System design

Figure 3 shows a system diagram of FaaSnap. The blue elements are FaaSnap components and the gray elements are existing Firecracker components. The dashed components are expected to interact with the FaaSnap daemon in real-world FaaS deployments, but are not needed in our experimental use of FaaSnap.

The FaaSnap daemon is the core system component of FaaSnap. It communicates with the Firecracker VMM and manages related resources, and it forwards invocation requests to the VMs. It is similar to the "MicroManager" component in Firecracker deployments in AWS [1]. The FaaSnap daemon manages local VM images, guest kernels, snapshot memory and working set files, active VMs, and network resources like namespaces and virtual network devices. All of the techniques described in this section are implemented in the FaaSnap daemon except for the provisioning of the guest memory. FaaSnap also exposes an API to allow remote clients to control resources and send invocation requests. In real-world deployed FaaS systems, the remote clients would be load balancers that route invocation requests and cluster-level resource managers that control the VM lifecycles.

We modify the Firecracker VMM for the provisioning of FaaSnap guest memory.

4.2 Concurrent paging

As shown in Section 3, default on-demand paging is costly because many slow VM page faults that need disk reads occur during an invocation. REAP, on the other hand, prefetches all of the previously accessed pages to avoid page faults during an invocation. The downside of this approach is that it results in a long initial loading step that blocks the invocation process. The problem is even more pronounced when the working set is large (Figure 1).

Instead of blocking the VM while waiting for the prefetch to complete, the FaaSnap daemon starts the VM immediately after setup, similar to the original Firecracker implementation. Once the daemon receives an invocation request, it starts a loader thread to prefetch the pages from the working set recorded in earlier invocations. FaaSnap starts the loader as a thread in the daemon instead of a thread in the Firecracker VMM so that it can start prefetching immediately when the daemon receives the invocation request, and does not need to wait for the VMM to start executing.

Thus FaaSnap supports concurrent page faults from both the VM and the FaaSnap loader. If a page is first accessed by the loader, the page fault handler will read and install the page into the page cache. When the page is later accessed by the VM, the page will be served from the page cache, resulting in a faster minor page fault instead of a blocking major page fault. Pages first accessed by the VM cause a blocking major page fault. Although it cannot guarantee that all the working set pages are first read by the FaaSnap loader, we show in Section 6 that concurrent paging reduces a significant portion of major VM page faults while removing the need for a long initial read step that blocks VM execution.

4.3 Working set group

To move disk reads out of the critical path of VM page fault handling, the daemon loader needs to prefetch pages before the VM accesses them. Ideally, the loader should access the pages in an order similar to that of the VM so that the loader has a higher chance of prefetching a page before the VM. A straightforward idea is to record the order of page accesses in the first invocation, and let the loader access the pages in the same exact order. However, loading the pages using the previous access order exhibits poor locality, leaving the Linux `readahead` mechanism ineffective in its prefetching. For the `image-diff` function execution, for instance, reading by access order takes the loader 100 ms longer than sequential address order, which slows down populating the page cache with prefetched pages.

Instead, FaaSnap uses an approximate order for loading. It divides the working set pages into several working set groups by their access order: e.g., the first N accessed pages are assigned group 1, the next N accessed pages are assigned group 2, etc. The loader reads groups in increasing group number, and reads pages within a group sequentially. In this way, the loader is more likely to access a page earlier than the guest while preserving disk access locality when loading. From our experiments, we find N = 1024 works well across the function benchmarks and use this value in our evaluations. Note that the working set group is different from Halite [36], where locality groups are used to predict clustering of pages instead of ordering of accesses.

4.4 Host page recording

To determine the working set of the guest VM, in previous work Zhang et al. [37] scanned the access bits of the page table entries and REAP used `userfaultfd` to record the address of every faulting guest page. In both methods, the working set obtained is limited to the faulting *guest pages*. However, the host kernel `readahead` mechanism fetches extra pages on each page fault that are not tracked with `userfaultfd` or access bits. While tracking only the faulting pages is a natural design decision, we find that, when handling invocations with different inputs, it improves performance if the working set includes not only the faulting guest pages but also the *host pages* cached by `readahead`. The reason is the pages touched by `readahead` can be accessed in future invocations when function inputs are different and the working set changes. In other words, `readahead` can "predict" some future guest memory accesses even if the pages are not touched in previous invocations.

Instead, FaaSnap uses the `mincore` syscall to construct the working set file. `mincore` scans the *present bits* in the page table entries to determine if pages in a memory range are present in memory. In our case, it detects if guest pages are in the host page cache. By calling `mincore` repeatedly, FaaSnap records the new pages loaded since the last `mincore` call. It assigns a working set group number to pages using the order they appear in the `mincore` scans. As an added benefit, `mincore` has lower overhead than `userfaultfd` for recording working set pages since it does not need to invoke a user-level process to handle and record a page fault. By including present pages instead of just accessed pages in the working set, FaaSnap is more tolerant of changes in the working set.

4.5 Per-region memory mapping

The Firecracker snapshot implementation maps the entire guest memory to the guest memory file. As a result, all guest page faults, including anonymous page faults, are translated into more costly file-backed page faults on the host, which degrades performance as shown in Section 3.

In the guest kernel, an anonymous page is initially attached to a read-only all-zero page. Any write to the anonymous page traps to the guest's copy-on-write page fault handler, which copies the zero page into a newly allocated guest physical page. This page copy traps into the file-backed page fault handler in the host kernel, which issues a disk read request. However, the read is unnecessary since the page is being overwritten with zeros.

The nature of the problem is a semantic gap between the host and the guest. The host kernel does not know the cause of the guest page fault. If the host kernel knows the guest is trying to access a newly allocated page, it can happily serve the page in the anonymous region. One solution to the problem is to use a paravirtualized kernel to explicitly provide the host kernel the cause of the page fault so that the host can handle it accordingly.

We choose a simpler approach. Instead of memory-mapping the entire guest memory file, FaaSnap only maps the pages that are non-zero to the guest memory file. Zero pages are instead `mmap`'d to anonymous host memory. Although guest zero pages are not necessarily anonymous pages in the guest, using anonymous memory in the host still guarantees that the pages are initialized to zero, thus providing correct semantics. When an invocation is finished, FaaSnap scans the guest memory file, merging consecutive zero pages into *zero regions* and non-zero pages into *non-zero regions*. A region is also assigned a group number, which is the lowest group number of any page in the region. During an invocation, our modifications to the Firecracker VMM `mmaps` zero regions to anonymous memory, and non-zero regions to the memory file. In this way, a page fault on the guest zero page will be handled by host anonymous memory instead of triggering a slow disk read.

Another optimization is the handling of freed pages. If the guest kernel frees a guest physical page, its contents no longer matter and it will be overwritten to zero when allocated again. However, Linux does not actively clear the contents of a freed page. The host has no way to know if a page has been freed and no longer needed. We modify the guest kernel so that it always *sanitizes* freed pages, writing zeroes to the page. As a result, FaaSnap excludes the freed pages from the set of non-zero pages, and future accesses to the freed pages will be fast anonymous page faults.

4.6 Loading set

As discussed above, the FaaSnap daemon uses `mincore` to determine the working set of a function invocation during the record phase. Since the working set often includes zero regions, which FaaSnap maps to anonymous memory, the loader does not need to prefetch the zero regions during the restore. We define the *loading set* as the working set pages excluding the zero pages. The group numbers of the loading set regions are derived from the working set. As a result, the loader only needs to prefetch the loading set regions.

The VMM needs to `mmap` every loading set region when setting up the VM for an invocation. Even for a simple `hello-world` function, there can be more than 1000 loading set regions, and the overhead to create large numbers of mappings is not negligible. However, we find that many loading set regions adjacent in the guest address space are only separated by a few non-loading set pages (i.e., either zero pages or non-working set pages). FaaSnap merges these adjacent regions by including the pages in between them. This relaxation greatly reduces the number of regions that need to be separately mapped, while only adding a small amount of additional data read. The distance threshold for merging two regions is empirically set to 32 pages, a value that reduces the number of regions to small enough while

Type	Non-zero	Working set	Mapping
Loading set	Y	Y	loading set file
Cold set	Y	N	memory file
Released set	N	Y	anonymous
Unused set	N	N	

Table 1. Types of pages and their mapping in FaaSnap.

not adding too many unneeded pages. For `hello-world`, merging regions reduces the number of regions to less than 100, while total amount of data increases by only 5%.

4.7 Loading set file

Our experience is that, across a variety of functions, the loading set pages tend to be scattered throughout the guest physical address space. Scattered reads, though, usually lead to lower disk performance. Similar to REAP, FaaSnap stores the loading set into a compact file that only contains the loading set pages so that the loader reads it more efficiently.

In contrast to REAP, though, FaaSnap sorts the loading set regions first by their group numbers, then by their addresses. The file offsets and sizes of the regions are cached in the FaaSnap daemon. When a function is invoked, the FaaSnap daemon tells the VMM to `mmap` the loading set regions using the recorded file offsets and lengths. The loader then reads the loading set file in sequential order, caching the pages that are scattered in the guest address space. When the guest VM accesses those pages, they result in a minor page fault that installs the page in the guest. Furthermore, as described in Section 4.2, FaaSnap supports concurrent paging so that reading the loading set file does not block function execution.

4.8 Summary

Table 1 summarizes the four types of pages in the guest VM memory. The loading set pages (i.e., non-zero pages in the working set) are mapped to the loading set file using offsets recorded when the loading set file was created and used by the loader in the daemon during subsequent function invocations. The cold set are non-zero pages not accessed during the first invocation. These pages are usually more than 100 MB in size, and most of them are pages used in the guest booting process. The cold set are mapped to the original memory file using the same offset as in the guest memory. They are less likely to be accessed during the next invocation, so they are not included in the loading set file, but they still need to be mapped to the original memory file to ensure memory integrity in case they are accessed. The released set are zero pages touched in the first invocation, and are primarily pages freed by the guest kernel. The unused set are zero pages that are never touched. Both the released and unused sets can be safely mapped to anonymous memory.

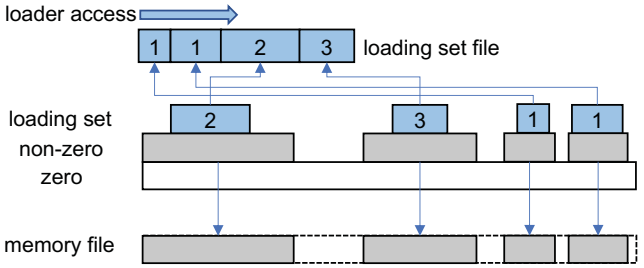


Figure 4. VMM guest memory mappings and backing files. Mappings are created from the bottom layer to the top layer using the overlapping semantics of the kernel, where upper layers override the pages of lower layers. Zero regions are mapped to anonymous pages (white bar). The cold set (non-zero regions not in the working set) is mapped to the memory file (gray bar). The loading set is mapped to the loading set file. The numbers in the loading set denote group numbers, lower group numbers are stored in earlier locations in the loading set file. The loading set file is read in sequential order.

One way to map these regions is to make non-overlapping `mmap` calls for each individual region. However, we can reduce the number of `mmap` calls by mapping smaller regions on top of existing ones in a hierarchy. First, an anonymous region for the entire guest address space is mapped. Then non-zero regions are mapped to the same offset in the memory file. Finally, the loading set regions are mapped to pre-recorded offsets in the loading set file. Figure 4 shows the VMM memory structure and the corresponding mapped files.

FaaSnap combines all the techniques described above, and Figure 5 shows a flow chart for the steps. In the first invocation, or record phase, the VM is started from restoring a "clean" snapshot. FaaSnap obtains the working set groups using repeated `mincore` syscalls to the memory file. After the invocation, a new snapshot is created to store the warm state. FaaSnap then scans the new memory file to find non-zero pages. The loading set is the intersection between the working set and non-zero pages. Adjacent loading set regions are merged to reduce the number of regions. The loading set is then stored into a compact loading set file in the order of group numbers and the region offsets are recorded.

In a subsequent invocation, or test phase, FaaSnap will use the new memory file and loading set file. Concurrent paging uses the host OS page cache to reduce guest major page faults. Per-region memory mapping allows different sets to be handled separately to improve performance. Overlapping `mmap` calls are used to simplify the mapping process. With all the optimizations combined, FaaSnap significantly reduces the guest VM's page fault handling time on the critical path.

5 Implementation

The majority of FaaSnap is implemented in the daemon with minor parts implemented in the Firecracker VMM for the

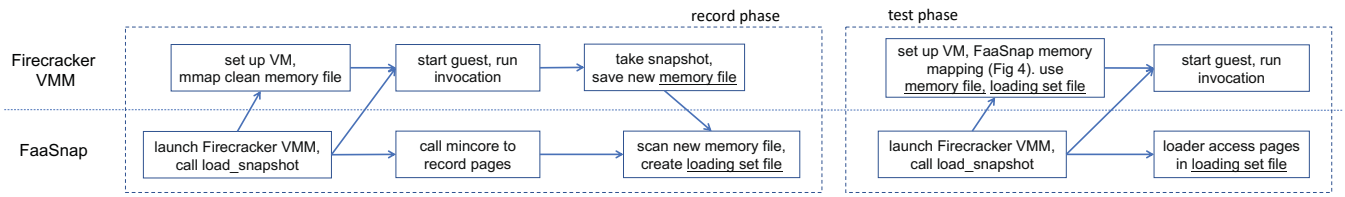


Figure 5. Flow chart of FaaSnap snapshot and restore.

per-region mapping technique. FaaSnap consists of ~2,500 lines of Go code (not including REAP integration) and ~200 lines of Rust code in the Firecracker VMM.

Code for functions is installed as Python files in the guest VM. We built a Flask-based server that runs in the guest and waits for HTTP invocation requests and invokes function code. The FaaSnap daemon supports operations like creating functions using installed images and kernels, booting VMs for a function, invoking functions on the booted VM, taking snapshots of a VM, restoring snapshots, etc. FaaS applications rely on external storage to store state, including input, output, and intermediate data, that persists beyond the lifetime of a function invocation. We run an in-memory Redis data store on the host for external storage for functions.

The daemon starts and manages the Firecracker VMM upon receiving user requests through an API. It communicates with Firecracker using HTTP via Firecracker's Unix sockets. Once the guest VM is started, it uses the guest IP address to connect to the Flask server (running in the guest) for invoking functions.

In the record phase, the daemon calls `mincore` on the mapped memory repeatedly to check for newly accessed pages to implement host page recording. Since the daemon only needs to record the 1024 recently accessed pages in a group, it waits for the guest to allocate enough new pages before calling `mincore`. The daemon polls `procfs` for the resident set size (RSS) of the guest. Once the RSS has more than 1024 new pages, it calls `mincore` to record them.

We extend the API call between the daemon and Firecracker VMM with additional arguments that specify the locations of non-zero regions and loading set regions. The daemon first allocates an anonymous region. It then uses the `MAP_FIXED` flag to place the overlapping non-zero and loading set regions onto the exact offsets of the anonymous region. The daemon then provides the whole memory region to KVM to use as the guest memory by issuing an `ioctl` call.

We modify the `free_pages_prepare` function in the guest kernel to sanitize freed pages. Sanitizing pages imposes overhead for the guest kernel (around 10% of execution time). Since sanitizing freed pages is only necessary during the record phase, we disable page sanitizing in the test phase. At the end of the record phase and before creating the snapshot, the FaaSnap daemon sends an HTTP request to the guest daemon, which signals the guest kernel to disable page sanitizing via the `procfs` interface.

	Description	Input A	Input B	Working Set A	Working Set B
Hello-world	a minimal function	n/a	n/a	11.8 MB	11.8 MB
Read-list	read an 512 MB Python list	n/a	n/a	526 MB	526 MB
Mmap	allocate anonymous memory	512 MB	512 MB	536 MB	536 MB
Image	rotate a JPEG image	101 KB JPEG	103 KB JPEG	20.6 MB	32.6 MB
Json	deserialize and serialize json	13 KB json	148 KB json	12.7 MB	14.4 MB
Pyaes	AES encryption	string of 20k	string of 22k	12.6 MB	13.2 MB
Chameleon	render HTML table	table size 30k	table size 40k	22.9 MB	25.1 MB
Matmul	matrix multiplication	matrix size 2000	matrix size 2200	113 MB	133 MB
FFmpeg	apply grayscale filter	1-sec 480p video, 338KB	1-sec 480p video, 381KB	179 MB	178 MB
Compression	file compression	13 KB file	148 KB file	15.3 MB	15.8 MB
Recognition	PyTorch ResNet image recognition	ResNet-50 cnn, 101 KB JPEG	ResNet-50 cnn, 103 KB JPEG	230 MB	234 MB
PageRank	igraph PageRank	graph size 90k	graph size 100k	104 MB	114 MB

Table 2. Functions used in the evaluation. Different inputs are used in the record and test phases to get realistic results.

The REAP developers generously support their system as an open-source project, and we integrated REAP as an optional mode for evaluation purposes. When receiving an invocation, the FaaSnap daemon registers the snapshot with REAP and activates REAP in a goroutine, which waits to receive the `userfaultfd` file descriptor through which it handles guest VM page faults on behalf of the kernel.

6 Evaluation

We evaluate the performance of FaaSnap, including function execution time, input size sensitivity, execution breakdown, the contributions of different optimizations to performance, and performance under bursty workloads and remote disks.

6.1 Methodology

Table 2 lists the functions we use for evaluation. The first three are synthetic, and the rest are from FunctionBench [16], SeBS [8], and Sprocket [3]. The functions cover a wide range of applications including web requests, multimedia, scientific computing, machine learning, and graph processing. To reflect the expected scenario where the same function will have different inputs in different invocations, we prepare two sets of inputs. For functions with variable inputs, input A is smaller than input B. There are two phases in a test, a

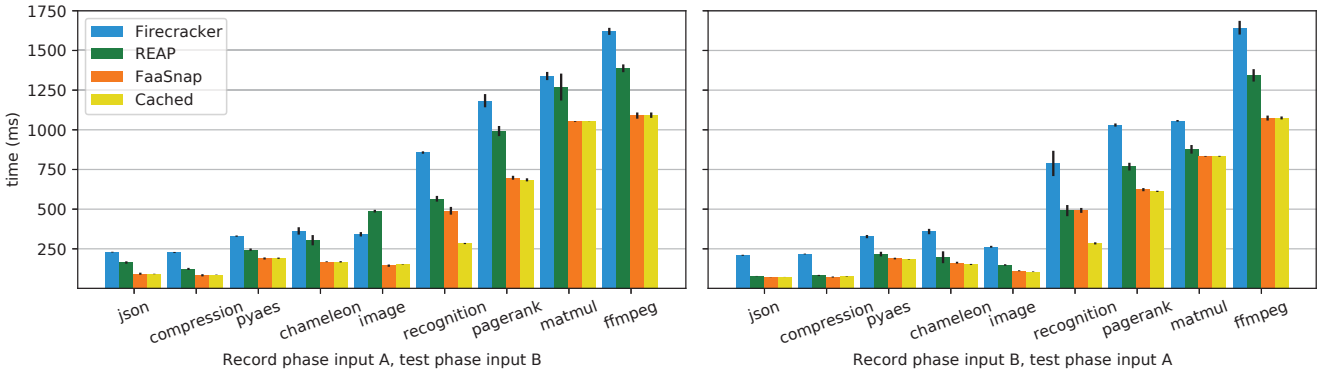


Figure 6. Execution time of the benchmark functions. Standard deviations are shown in error bars. Cached is used as a reference. FaaSnap shows similar results to Cached: most of the VM page faults are minor page faults, and the slow disk reads are taken out of the VM page fault critical path.

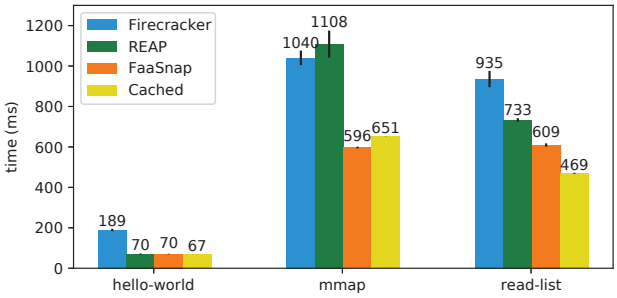


Figure 7. Execution time of the three synthetic functions.

record phase and a test phase. The first invocation happens in the record phase, whose warm state in the guest memory is stored in the snapshot or the working set/loading set file. The test phase is the actual test. We use input A in the record phase, input B in the test phase, and vice versa to evaluate the expected scenario where the input size grows or shrinks.

We drop the page cache of all the relevant files, including the snapshot memory file and the working set file, before each test to ensure we measure performance when the pages are actually read from disk.

Our measurement platform is an AWS `c5d.metal` instance with a 96 vCPU Intel Xeon Platinum 8275CL CPU running at 3.00 GHz, 192 GB of memory, and 25 Gbps network bandwidth. It runs Ubuntu Linux with a 5.4.0 kernel. Each guest VM has 2GB of memory and 2 vCPUs, a typical configuration in real-world FaaS systems like AWS Lambda. The guest uses Debian Linux with a 4.14 kernel. The disk is an NVMe SSD with measured maximum read throughput of 1589 MB/s and IOPS of 285,000.

6.2 Execution time

We first evaluate overall function execution time for different snapshot methods, which includes both guest VM setup and function invocation time. We measure the time to execute the functions listed in Table 2 using Firecracker, Cached, REAP, and FaaSnap snapshots. Cached snapshots preload the snapshot memory file into the page cache before execution. While not practical, we use it as a reference for other systems. We run each test five times and show the average and standard deviation.

Figure 6 shows average execution time and standard deviation for the benchmark functions. The left subfigure uses input A in the record phase and input B in test phase, and the right subfigure reverses the inputs. Figure 7 shows the results of the three synthetic functions. These functions have the same input (or no input) for the record and test phases, and therefore are shown separately.

FaaSnap has the shortest execution time for all the functions compared to Firecracker and REAP snapshots. On average, it improves upon Firecracker by 2.0× and it improves upon REAP by 1.4×. Note that for the benchmark functions, FaaSnap's speedup over REAP is higher when the test phase uses larger input B (1.55x) than when the test phase uses smaller input A (1.16x). The reason is that larger inputs in the test phase trigger more page faults outside of REAP's working set file (which was created with a smaller input), and these page faults are handled with more overhead at user level via REAP's use of `userfaultfd`. FaaSnap's per-region memory mapping, loading set, and host page recording techniques help it handle the workloads not captured by the working sets more efficiently. FaaSnap's concurrent paging also prevents the initial long blocking of functions with large working sets.

Moreover, the performance of FaaSnap is close to Cached snapshots for most functions. FaaSnap loads most of the loading set pages to the host OS page cache before they are accessed by the guest VM. Performance with FaaSnap can sometimes even be faster than Cached because the per-region memory mapping technique allows page faults for zero pages

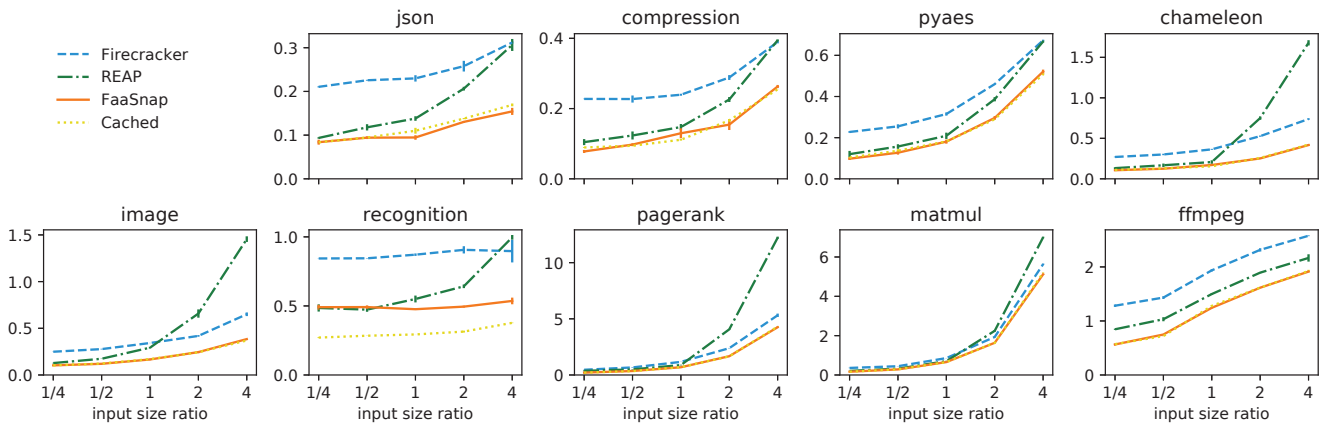


Figure 8. Execution time under varying input size ratios. All y-axes are in seconds. FaaSnap performs similarly to Cached for most functions, indicated by the orange solid line overlapping well with the yellow dotted line. REAP performance degrades when input size ratios are larger, shown by the steeper lines than others when the ratio > 1.

to be handled in anonymous memory, which is faster than faulting from the page cache (Section 3). Cached snapshots outperform FaaSnap in the `read-list` and `recognition` functions because of their access patterns. These functions read existing pages aggressively, and the FaaSnap loader cannot always keep pace with the page faults created by the guest. On average FaaSnap snapshots are only 3.5% slower than Cached snapshots: FaaSnap provides performance using an SSD nearly equal to that of the in-memory page cache.

6.3 Input size sensitivity

To further evaluate the sensitivity of the snapshot methods to input variation, we perform another series of experiments. For each of the functions in Figure 6, we use inputs of the same sizes as input A in the record phase, and then vary the sizes of the inputs in the test phase (whose contents are also entirely different). In particular, we use inputs in the test phase whose sizes are 1/4× to 4× the size of the input in the record phase. We run each test three times and report averages and standard deviations.

Figure 8 shows the results of this experiment. Each graph shows the results for one of the nine functions that take variable inputs. Each curve in the graph corresponds to a different snapshot technique. Each point on a curve shows the execution time of a function for a particular ratio of input sizes in the test and record phases. In the graph for `ffmpeg`, for example, the point on the blue dashed line at $x = 4$ shows the execution time of `ffmpeg` using Firecracker when the size of the input video for the test phase is 4× the size of the input video used in the record phase to take the snapshot.

These results show that FaaSnap snapshots provide performance benefits across the range of input size ratios. As with the results in Figure 6, FaaSnap outperforms Firecracker and REAP for all of the functions, and performs similarly to Cached except for `recognition`. In contrast, for many functions REAP execution time significantly increases when the input size is larger than that of the record phase, especially for `chameleon`, `image`, and `pagerank`. At these larger input sizes, REAP performs worse than Firecracker for many of the functions.

FaaSnap, on the other hand, handles changing input sizes well and its benefits are resilient to changes in working set. Relative to Firecracker, the benefit of FaaSnap snapshots is roughly constant across differences in input sizes, effectively providing the performance of having the pages that benefit the test phase prefetched into the cache. Note that the impact of this benefit on overall execution time does decrease for larger input size ratios. As function execution time becomes dominated by the input itself, working set optimizations including FaaSnap are going to provide diminishing returns. For these situations, the goal of a system taking advantage of working sets is to help when it can but otherwise avoid degrading function execution time, which FaaSnap achieves but unfortunately REAP does not.

6.4 Performance analysis

To provide more insight into the performance differences between FaaSnap and REAP, we examine the execution breakdown of the `ffmpeg` and `image` functions in more detail. They show different behaviors under the two systems, and are representative of other functions. We collect metrics including total execution time, working set fetch time, working set fetch size, guest page fault size, and page fault waiting time. The page fault waiting time includes both the page fault service time (`kvm_mmu_page_fault`) and the time KVM waits for the guest CPU to be ready to run (`kvm_vcpu_block`). We collect the numbers using bpftrace [28] and `perf` and report them in Table 3.

	Total time	Fetch time	Fetch size	Guest pagefault size	Page fault waiting time
REAP, ffmpeg	1408 ms	257 ms	201 M	20 M	780 ms
FaaSnap, ffmpeg	1070 ms	107 ms	146 M	32 M	866 ms
REAP, image	480 ms	51 ms	22 M	31 M	342 ms
FaaSnap, image	136 ms	55 ms	88 M	7.2 M	109 ms

Table 3. Performance analysis.

While FaaSnap outperforms REAP when executing both functions, it does so for different reasons. For `ffmpeg`, the benefit with FaaSnap mostly comes from a shorter fetch time. REAP's fetching process includes not only synchronously reading the working set, but also installing the pages via `userfaultfd`, which slows down execution. FaaSnap starts the function concurrently with fetching, avoiding the initial fetching delay.

For `image`, FaaSnap fetching is slower than REAP. `image` with FaaSnap has a larger relative working set size, which is caused by the sparse access pattern of `image` resulting in more pages being recorded in its loading set under FaaSnap than in the working set under REAP. Despite faster working set fetching, though, REAP is much slower when running the function because of a much longer page fault waiting time: when serving each page fault, KVM blocks to wait for the guest CPU to be ready, resulting in extra context switches that increase waiting time. In contrast, FaaSnap has far fewer page faults and handles them in the kernel. This benefit, together with concurrent paging, makes `image` perform 3.5× faster on FaaSnap.

6.5 Optimization steps

To understand how the different optimizations contribute to the overall performance improvements of FaaSnap, we selectively measure incremental contributions of the optimizations. Starting with Firecracker as the baseline, we also measure FaaSnap using just concurrent paging (Section 4.2), then the combined optimizations that together support per-region mapping (Sections 4.2 to 4.5), and finally all FaaSnap optimizations combined. We focus on the `image` benchmark and measure the execution times as well as the number of page faults, total page fault handling time, and the number of disk read requests caused by VM page faults. We collect the data using the bpftrace [28] tool.

Figure 9 shows the results. The daemon loader uses concurrent paging to prefetch pages concurrently with the execution of the guest VM, which reduces the number of major page faults, total page fault time, and number of block read requests from the guest VM. Per-region mapping, however, has *more* major page faults and *fewer* block requests and a lower page fault handling time. These seemly conflicting numbers are the result of different paging orders. In concurrent paging, the FaaSnap loader reads the working set pages in the address space order, which does not correspond to the exact guest VM page access order. When the guest VM has a major page fault, it usually causes a disk read. Therefore, the average time of serving a page fault with just concurrent paging is high.

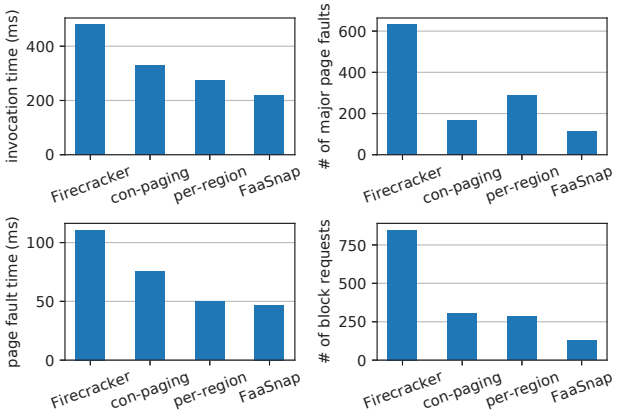


Figure 9. Optimization steps and their effects.

In comparison, with per-region mapping the daemon loader prefetches the pages in approximately the same order as the guest VM because of its use of working set groups. When a guest VM major page fault happens, it is more likely that the faulting page is being installed to the page cache from the disk by the daemon loader and does not cause another disk read. Therefore, the VM page fault handling time is shorter in per-region mapping, making the VM execution faster. Per-region mapping creates more major page faults since it is able to progress faster, but its major page faults are less "harmful" than major page faults in concurrent paging.

FaaSnap further reduces the loader's read time using the loading set and loading set file optimizations. The loader can prefetch most of the pages before the guest accesses them, leading to the fewest number of major page faults, fewest number of block read requests, shortest page fault time, and shortest invocation time.

6.6 Bursty workloads

Burst-parallelism is an increasingly common invocation pattern in serverless computing [29]. Real-world events like IoT events and data analytics frameworks can create a large number of parallel invocations in a short time window, and it is important for platforms to be able to efficiently handle such workload patterns. We evaluate two kinds of bursty workloads, the burst of VMs from the same snapshot and from different snapshots: same snapshot represents bursty workloads from the same application while different snapshots represent those from different applications.

We evaluate bursty workloads using Firecracker, REAP, and FaaSnap. For the same snapshot, Firecracker and FaaSnap take advantage of the host OS page cache to avoid redundant disk reads when servicing page faults from the guest

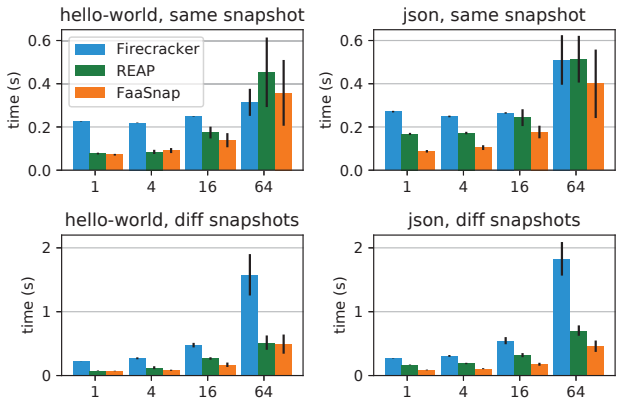


Figure 10. Performance with bursty workloads. Tests are performed using two functions, and using the same snapshot or different snapshots for each instance.

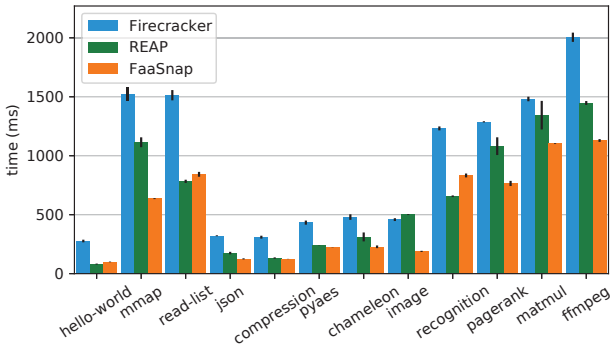


Figure 11. Performance using remote storage for snapshots and related files.

VMs. REAP bypasses the page cache to maximize read bandwidth. The FaaSnap loader uses a lock to ensure the loading set is accessed exactly once to avoid redundant accesses.

We measure workloads running 1–64 invocations of the `hello-world` and `json` functions at the same time. Figure 10 shows their average execution times and standard deviations. When using the same snapshot, both REAP and FaaSnap outperform Firecracker when parallelism is less than 64. Under higher parallelism, Firecracker benefits from the page cache when multiple guests access the same set of pages. The guests are in effect loading the cache for each other. FaaSnap is faster than REAP in all tests since REAP bypasses the page cache, missing the caching opportunity. When parallelism reaches 64, the CPU becomes the bottleneck and all settings take longer to execute and have higher variance.

When using different snapshots, Firecracker performance degrades quickly because the disk overheads from on-demand reading of all of the different snapshots increase quickly. Both REAP and FaaSnap run much faster than Firecracker with more efficient disk reads. REAP performs similarly to the case when it uses the same snapshot because it does not take advantage of the page cache. FaaSnap outperforms REAP, especially for `json` whose working set has more variance. Overall FaaSnap handles parallel guest page accesses more efficiently than Firecracker and REAP, better supporting bursty workloads.

6.7 Remote storage

In disaggregated storage environments, machines do not have local disks and attach remote block storage. To evaluate performance in such cases, we measure the invocation time while snapshots and related files are stored on remote block storage. We use an AWS Elastic Block Store (EBS) `io2` volume with 64K maximum IOPS and 1 GB/s maximum throughput. We measure the execution time of all the functions in Table 2 using Firecracker, REAP, and FaaSnap snapshots. We conduct the test three times and report the mean and standard deviations of execution time.

Figure 11 shows the results. Baseline Firecracker snapshot performance using remote EBS is on average 33% slower than using the local NVMe SSD, reflecting the effects of increased latency and lower bandwidth from remote disks. Both REAP and FaaSnap significantly outperform Firecracker for most functions. FaaSnap is faster than REAP in most functions, except `recognition`, `read-list`, and `hello-world`. In these functions, the working set is very stable, making REAP's blocking working set fetching more efficient. On average, while FaaSnap performance using EBS is 28% slower than using a local NVMe SSD, it is 2.06x faster than Firecracker and 1.20x faster than REAP.

The feasibility of FaaSnap on remote storage enables it to be deployed on any machine, not just those with high-speed local SSDs, thereby extending its deployment flexibility.

7 Discussion

7.1 Warm starts vs. snapshots vs. cold starts

A FaaS function invocation can be served in a warm VM, if a warm environment exists, using a snapshot, if there are existing snapshots, or by booting a cold VM if neither exists. The Azure Function traces [29] show that less than half of the functions are invoked every hour, and less than 10% are invoked every minute. For the most frequent functions, keeping warm VMs alive and using warm starts is the best choice. Snapshots are useful for less frequently executed functions where keeping warm VMs has more overhead than benefit. Snapshots are also useful for applications with large variance in workloads, such as applications with sudden bursts of workloads. In this case, keeping warm VMs reduces resource utilization while cold starts have high latency. As shown in Section 6.6, FaaSnap can serve bursts of function invocations efficiently. For very cold functions that are rarely

invoked, snapshots are likely not worth the storage and management costs. Therefore, snapshots can be used to replace cold starts for functions invoked less frequently than those that benefit from warm VMs, and replace warm VMs when their utilization is low (e.g., on eviction).

7.2 Storage costs

While snapshots can improve performance, they do incur a real cost for cloud providers for storing and managing the snapshot files. Two factors that determine the storage cost are snapshot file sizes and storage location.

In general, the sizes of snapshot memory files are the same as the guest memory size since the snapshot is a full copy of the guest memory. These sizes are typically a few hundred MB to a few GB, which are comparable to function image sizes. In practice, since guest memory often contains zero pages, snapshot files can be saved as sparse files to reduce their sizes. In effect, though, snapshots increase the storage requirements for functions that use them. As a result, for very infrequent functions, providers can choose to not take snapshots at all to reduce overall storage requirements.

Snapshot files can be stored on local SSDs, remote block storage like EBS, or remote object storage like AWS S3. Storing snapshots on local SSDs provides the best performance, but it is a relatively limited and expensive resource. Remote storage for snapshots is cheaper and much more plentiful, but has higher latency for serving snapshots.

As a result, deploying snapshots represents a tradeoff for providers. While most of our experiments, as well as those performed by other snapshot optimization systems like REAP [33] and Catalyzer [9], measure performance using local SSDs, such results represent an upper bound in performance and should be interpreted in that light. The best case is if providers selectively use local SSDs for snapshot storage for functions invoked frequently, but not frequently enough to be served from warm VMs cached in memory (e.g., warm VMs can be evicted from memory via snapshot to local disk).

Though using network storage does introduce additional latency, it is still a viable alternative. Section 6.7 shows that when using EBS, FaaSnap snapshots still provide performance benefits. Snapshots for functions further down the invocation frequency distribution can be stored in the slowest tier object storage such as S3. Providers can also access snapshots in a hierarchical caching scheme. FaaSnap provides an even more fine-grained option for them since it divides guest memory into memory sets and loading set groups. In the future we plan to explore storing relatively small loading set files on local SSD and larger memory files on remote storage to reduce storage costs while satisfying the performance requirements of reading loading sets.

7.3 Memory footprints

When the working set estimate is a good match for subsequent invocations, such as the experiments in Section 6.2, the memory footprints of FaaSnap are similar to that of Firecracker snapshots. In those experiments, on average it consumes 6% more memory than Firecracker (anonymous and page cache combined), although not always (FaaSnap consumes less memory than Firecracker in 3 of the 12 functions). Prefetching the working set into the page cache does not significantly increase the memory footprint because the working set is likely going to be loaded on-demand in Firecracker snapshots. When the working set estimate is largely inaccurate and large portions of the loaded working set are not used by the guest VM, the memory footprint can increase for working set optimizations like REAP and FaaSnap.

7.4 Snapshot security

Reusing a snapshot for VMs has two security concerns. The first is the inherent risks when reusing an environment including in-memory state that persists across invocations. This situation for snapshots is similar to using warm VMs, and is considered acceptable in FaaS. The second is unique to restoring multiple VMs from the same snapshot. The restored instances will have the same initial states. Specifically, pseudo-random number generators (PRNGs) with identical states can lead to a security vulnerability for cryptography. Several solutions have been proposed including using a new `madvise` flag to wipe memory locations with high-value secrets when taking a snapshot [6], and using a special device to provide unique VM IDs to the restored VMs [23].

8 Conclusion

On-disk snapshots are a promising way to prevent the overhead of cold starts in serverless computing. Existing snapshot and restore approaches have inefficiencies like long initial blocking, sensitivity to working set changes, and a host-guest memory semantic gap. In this paper we propose FaaSnap, which develops several techniques and optimizations such as concurrent paging and per-region mapping to reduce the costs of guest VM major page faults. Experimental results show that FaaSnap improves function execution by up to 3.5x than the state-of-the-art, and it is only 3.5% slower than snapshots cached in memory.

Acknowledgments

We thank our anonymous reviewers and our shepherd, Adam Belay, for their insightful and constructive suggestions and feedback. We would like to thank Dmitrii Ustiugov for helping us integrate REAP as a mode into FaaSnap. We also thank Zachary Blanco for helping us set up the development environment. Funding for this work was provided in part by National Science Foundation grants CNS-1629973 and CNS-1763260, generous support from Google, and operational support from the UCSD Center for Networked Systems.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association, Santa Clara, CA, 419–434.

[2] 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 (USENIX ATC'18)*. USENIX Association, Boston, MA, USA, 923–935.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18)*. ACM, Carlsbad, CA, 263–274.

[4] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. ACM, Heraklion, Crete, Greece, 1–15.

[5] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'19)*. ACM, Santa Cruz, CA, USA, 13–24.

[6] Adrian Costin Catangiu. 2020. Introduce MADV_WIPEONSUSPEND. https://lwn.net/Articles/825230/.

[7] containerd. 2021. An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/.

[8] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference (Middleware'21)*. ACM, New York, NY, USA, 64–78.

[9] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. ACM, Lausanne, Switzerland, 467–481.

[10] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*. ACM, New York, NY, USA, 45–59.

[11] Firecracker. 2021. Firecracker Snapshotting. https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md/.

[12] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. USENIX Association, Renton, WA, USA, 475–488.

[13] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Boston, MA, USA, 363–376.

[14] Alexander Fuerst and Prateek Sharma. 2021. Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM, New York, NY, USA, 386–400.

[15] gVisor. 2021. gVisor. https://gvisor.dev/.

[16] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD'19)*. IEEE, Milan, Italy, 502–504.

[17] Knative. 2020. Kubernetes-based platform to deploy and manage modern serverless workloads. https://knative.dev/.

[18] 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 4th ACM European Conference on Computer Systems (EuroSys'09)*. ACM, Nuremberg, Germany, 1–12.

[19] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2019. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *Journal of Systems and Software* 149 (2019), 340–359.

[20] Linux Containers. 2021. Container and virtualization tools. https://linuxcontainers.org/.

[21] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, Houston, TX, USA, 461–472.

[22] 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 Symposium on Operating Systems Principles (SOSP'17)*. ACM, Shanghai, China, 218–233.

[23] Microsoft. 2012. Virtual Machine Generation ID. http://go.microsoft.com/fwlink/?LinkId=260709.

[24] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. USENIX Association, Boston, MA, USA, 57–70.

[25] OpenFaaS. 2020. Serverless Functions Made Simple. https://www.openfaas.com/.

[26] Apache OpenWhisk. 2021. Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/.

[27] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD'20)*. ACM, Portland, OR, USA, 131–141.

[28] Alastair Robertson. 2021. bpftrace. https://github.com/iovisor/bpftrace.

[29] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, Virtual, 205–218.

[30] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, Virtual, 419–433.

[31] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*. ACM, Lisbon, Portugal, 275–287.

[32] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: Ephemeral Endpoints for Serverless Networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*. ACM, Virtual, 16–29.

[33] 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'20)*. ACM, Virtual, 559–572.

[34] 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 Twentieth ACM Symposium on Operating Systems Principles*. ACM, Brighton, UK, 148–162.

[35] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. 2016. Building a Chatbot with Serverless Computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs (MOTA'16)*. ACM, Trento, Italy, 1–4.

[36] Irene Zhang, Tyler Denniston, Yury Baskakov, and Alex Garthwaite. 2013. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*. USENIX Association, San Jose, CA, USA, 1–12.

[37] 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 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11)*. ACM, Newport Beach, CA, USA, 87–98.

A Artifact Appendix

A.1 Abstract

Our artifact includes the FaaSnap daemon implementation, modified Firecracker VMM, and modified guest kernel. We also include the testing scripts and resources used in the evaluation.

A.2 Description & Requirements

A.2.1 How to access. The FaaSnap source code and related resources are available and maintained in the Github repo https://github.com/ucsdsysnet/faasnap.

A.2.2 Hardware dependencies. Firecracker requires KVM support. We used an AWS EC2 `c5d.metal` instance since it has KVM capability and high performance NVMe SSDs for fast snapshot loading. We expect machines with KVM capability and fast SSDs should suffice.

A.2.3 Software dependencies. Ubuntu 18.04.6 LTS, Docker, Golang, go-swagger, and Python 3.10.

A.2.4 Benchmarks. We include all the benchmarks and testing resources in the Github repo.

A.3 Set-up

`README.md` includes all the set-up steps, including:

- Building Firecracker and the guest kernels (vanilla and modified).
- Starting a local Redis instance and populate Redis with input data.
- Building the FaaSnap daemon and function rootfs.
- Configuring and preparing the environment.

A.4 Evaluation workflow

A.4.1 Major Claims.

- *C1*: FaaSnap achieves on average 2x better performance than Firecracker and 1.4x than REAP for function execution time. This claim is supported by the experiment E1 described in Section 6.2 whose results are illustrated in Figure 6 and Figure 7.
- *C2*: FaaSnap achieves better execution time than Firecracker and REAP when input sizes vary greatly. This claim is supported by the experiment E2 described in Section 6.3 whose results are illustrated in Figure 8.
- *C3*: FaaSnap handles bursty workloads well. This claim is supported by the experiment E3 described in Section 6.6 whose results are illustrated in Figure 10.
- *C4*: FaaSnap achieves better performance than Firecracker and REAP when using remote snapshots. This claim is supported by the experiment E4 described in Section 6.7 whose results are illustrated in Figure 11.

A.4.2 Experiment E1.

- *Preparation*: Follow instructions in `README.md` to configure `test-2inputs.json`.
- *Execution*: Run `test.py test-2inputs.json` with root privilege.
- *Results*: The execution traces of invocations are accessible on the Zipkin web page running on port 9411 of the server. TraceIDs can be used to search traces. We expect the results to be similar to Figure 6 and Figure 7.

A.4.3 Experiment E2.

- *Preparation*: Follow instructions in `README.md` to configure `test-6inputs.json`.
- *Execution*: Run `test.py test-6inputs.json` with root privilege.
- *Results*: The execution traces of invocations are accessible on the Zipkin web page running on port 9411 of the server. TraceIDs can be used to search traces. We expect the results to be similar to Figure 8.

A.4.4 Experiment E3.

- *Preparation*: Follow instructions in `README.md` to configure `test-2inputs.json`.
- *Execution*: Run `test.py test-2inputs.json` with root privilege.
- *Results*: The execution traces of invocations are accessible on the Zipkin web page running on port 9411 of the server. TraceIDs can be used to search traces. We expect the results to be similar to Figure 10.

A.4.5 Experiment E4.

- *Preparation*: Follow instructions in `README.md` to configure `test-2inputs.json`.
- *Execution*: Run `test.py test-2inputs.json` with root privilege.
- *Results*: The execution traces of invocations are accessible on the Zipkin web page running on port 9411 of the server. TraceIDs can be used to search traces. We expect the results to be similar to Figure 11.