SnapBPF: Exploiting eBPF for Serverless Snapshot Prefetching

Stratos Psomadakis
National Technical University of Athens
Athens, Greece
psomas@cslab.ece.ntua.gr

Dimitrios Siakavaras
National Technical University of Athens
Athens, Greece
jimsiak@cslab.ece.ntua.gr

Chloe Alverti
University of Illinois Urbana-Champaign
Champaign, IL, USA
xalverti@illinois.edu

Symeon Porgiotis
National Technical University of Athens
Athens, Greece
sporg@cslab.ece.ntua.gr

Orestis Lagkas Nikolos
National Technical University of Athens
Athens, Greece
olagkas@cslab.ece.ntua.gr

Christos Katsakioris
National Technical University of Athens
Athens, Greece
ckatsak@cslab.ece.ntua.gr

Konstantinos Nikas
National Technical University of Athens
Athens, Greece
knikas@cslab.ece.ntua.gr

Georgios Goumas
National Technical University of Athens
Athens, Greece
goumas@cslab.ece.ntua.gr

Nectarios Koziris
National Technical University of Athens
Athens, Greece
nkoziris@cslab.ece.ntua.gr

Abstract

In this work, we design SnapBPF, an eBPF-based snapshot prefetching mechanism, targeting VM-sandboxed serverless functions, which enables the efficient capture and prefetching of function working sets in kernel-space. SnapBPF deduplicates function working sets in memory and obviates the need for separately serializing them on disk. We complement SnapBPF with a lightweight paravirtualized interface to efficiently handle VM-sandbox memory allocations without requiring any snapshot pre-processing. Our evaluation shows that SnapBPF is able to match and improve state-of-the-art performance with regard to i) function invocation latency and ii) memory usage for concurrent function invocations, without separately serializing working sets on disk or requiring any preemptive snapshot scanning.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Virtual machines**; **Memory management**.

HotStorage '25, Boston, MA, USA

ACM ISBN 979-8-4007-1947-9/2025/07
https://doi.org/10.1145/3736548.3737823

Keywords

Serverless, FaaS, Virtualization, Snapshots, eBPF

ACM Reference Format:
Stratos Psomadakis, Dimitrios Siakavaras, Chloe Alverti, Symeon Porgiotis, Orestis Lagkas Nikolos, Christos Katsakioris, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2025. SnapBPF: Exploiting eBPF for Serverless Snapshot Prefetching. In *17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '25), July 10–11, 2025, Boston, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3736548.3737823

1 Introduction

In Function-as-a-Service (FaaS) [1–6], the dominant serverless computing paradigm, users upload their code as a function, which the FaaS providers then sandbox, deploy and scale on their infrastructure. One of the dominant overheads of FaaS stems from cold starts [7, 8], i.e., when new function sandboxes need to be spawned to handle incoming requests. As FaaS providers typically resort to virtualization (VM)-based sandboxes for stronger tenant isolation guarantees [9], this exacerbates the cold-start overhead.

To alleviate this overhead, function snapshotting has been proposed by academia [10, 11] and adopted by the industry [12, 13]. For VM-sandboxed functions, the snapshotted function memory, i.e., the memory of the VM sandbox after the function has been initialized and pre-warmed [14, 15], is serialized to storage as a file. This snapshot file is then memory-mapped to act as the memory of newly-spawned

pre-warmed VM sandboxes, which will serve incoming function invocations.

While function snapshotting obviates the need for booting a fresh VM for each cold function invocation, it still suffers from the latency of faulting-in the snapshotted memory from storage. Previous works [11, 16, 17] have proposed capturing and prefetching the function working set from the snapshot file in userspace to minimize this overhead. However, as shown in Table 1, these approaches have limitations. All of them require separately serializing the working set on disk. Moreover, approaches that use the userfaultfd mechanism [18] are unable to deduplicate the function working sets in memory.

To that end, we design SnapBPF, a kernel-space approach, which leverages eBPF [19] to enable the efficient capture and prefetching of function working sets into the OS page cache. Our key insight is that modern SSDs relax the need for sequential I/O. This allows us to skip the serialization of the function working set to storage as a separate file. We instead employ eBPF to hook the readahead mechanism of the OS page cache to asynchronously fetch the working set chunks of the snapshot from storage. In this way and in contrast to state-of-the-art, SnapBPF deduplicates function footprints in memory via the OS page cache, without redundant userspace copies, and also obviates the need for serializing function working sets to separate files on disk.

SnapBPF also employs a lightweight paravirtualized interface to efficiently handle memory allocations for the VM sandbox. The VM kernel marks memory allocations via their page table entries. When the host handles the nested faults for these page table entries, it serves them with anonymous memory allocations instead of unnecessarily fetching the page from the snapshot. Prior art tackles the problem by prescanning the snapshot, based on either page contents (zero pages) [17] or the VM kernel allocator metadata [16], in order to detect and filter such pages. By contrast, SnapBPF requires no snapshot preparation or scanning.

We implement and evaluate SnapBPF on Linux using the firecracker virtual machine monitor (VMM) [9]. Our evaluation shows that SnapBPF matches and improves upon state-of-the-art prefetching approaches. It has comparable latency to state-of–the-art and deduplicates function working sets in memory, thus keeping memory usage bounded for concurrent invocations. SnapBPF achieves this without having to rely on separately serialized working sets on disk and without doing any preemptive snapshot scanning or pre-processing.

In summary, the contributions of this paper are:

- the analysis of existing snapshot prefetching approaches and their shortcomings (Section 2),
- the design and implementation of SnapBPF, an eBPF-based kernel-space snapshot prefetching mechanism, which addresses the afore-mentioned shortcomings (Section 3),
- the evaluation of SnapBPF versus the state-of-the-art prefetching approaches (Section 4).

2 Motivation

2.1 Existing Prefetching Approaches

Existing snapshot prefetching approaches can be broadly classified into two categories based on the mechanism they employ to capture and prefetch the function working set. Regardless of the mechanism used, the capture and loading of the working set is essentially implemented in userspace. Table 1 summarizes their design choices and limitations.

Userfaultfd. REAP [11] and Faast [16] use Linux userspace page fault handling (userfaultfd) [18]. When spawning a new VM sandbox, they register a userspace page fault handler, which gets triggered on VM memory page faults. When handling such a fault, the OS allocates anonymous memory to serve the fault and then hands over the fault to userspace. The userspace fault handler subsequently fetches the faulting page from the snapshot, stored on disk, and copies (*installs*) its contents to the page allocated by the OS.

Both techniques that use userspace faults first identify the function's working set, and then serialize it to storage (**record phase**). For subsequent VM sandbox creations (**invocation phase**), they both prefetch the function working set from storage and preemptively install it in the VMM via userfaultfd. Both REAP and Faast use direct IO when fetching the snapshot from storage, to bypass the page cache and avoid the overhead of intermediate memory copies. As they both rely on userfaultfd, they fail to deduplicate the function working set across different VM sandboxes, as shown in the evaluation section (Section 4, Figure 3c). The reason for this is that userfaultfd uses anonymous memory which is not shared between VM sandboxes of the same function, making it impossible to deduplicate the working set across different sandboxes in memory.

mincore / mmap. FaaSnap [17] on the other hand relies on the `mincore()` and `mmap()` system calls and the OS page cache for both capturing and prefetching the function's working set. The mincore system call returns a byte array which indicates whether each corresponding page of the calling process's virtual memory is resident in RAM [20]. FaaSnap uses the mincore system call to identify which snapshot pages have been fetched from storage into the OS page cache. Similarly to REAP and Faast, it serializes these pages to a separate working set file. In the **invocation phase**, FaaSnap memory-maps the working set file on top of the snapshot file. Instead of using userfaultfd, it relies on OS page cache prefetching

	Mechanism	On-disk WS serialization	In-memory WS deduplication	Stateless VM Allocation Filtering
REAP [11] / Faast [16]	Userfaultfd (User-space)	Yes	✗	✗
FaaSnap [17]	mincore / mmap (User-space)	Yes	✓	✗
SnapBPF	eBPF (Kernel-space)	No	✓	✓

Table 1: Comparison of snapshot prefetching techniques.

(*readahead*), using a userspace thread to issue buffered reads to fetch the working set to memory. This enables FaaSnap to deduplicate the working set across different VM sandboxes via the OS page cache. While this allows in-memory deduplication of the working set between different sandboxes, FaaSnap has to mmap each working set region separately. To reduce the number of mmap'ed regions, FaaSnap coalesces working-set regions with few non-working set pages between them into larger regions. While this reduces the mmap'ed regions to a manageable number, it also inflates the working set file, which can affect performance by amplifying IO, which we verify by instrumenting the kernel using eBPF.

2.2 VM-sandbox memory allocations

Due to the semantic gap between the VM and the host memory allocator, not all pages that will be used during the invocation of the function are captured by the working set. For ephemeral memory allocations inside the VM sandbox, i.e., for memory that is allocated during the invocation and freed afterwards, the working set pages will differ between invocations. As prior art points out [16, 17], fetching these pages from snapshot is unnecessary. The host kernel can instead provide the VMM with anonymous memory.

Faast and FaaSnap both tackle this issue by resorting to scanning and pre-processing the snapshot file. FaaSnap patches the VM kernel to zero pages when they are freed. It then scans the snapshot file for zero pages and maps those zero regions of the snapshot file to anonymous memory. Faast relies on the allocator metadata of the VM kernel to identify pages that are not actively used in the snapshot and routes faults for these pages to anonymous memory.

Regardless of the mechanism employed by each approach, both rely on preemptive snapshot scanning and pre-processing to optimize the handling of VM memory allocations.

3 SnapBPF

In this section, we present the design and implementation of SnapBPF. We first describe the kernel-space eBPF-based capture and prefetch mechanism and then continue with the paravirtualized PTE marking interface.

3.1 eBPF Capture and Prefetch

SnapBPF uses eBPF [19] to hook the readahead mechanism of the OS page cache and both capture and prefetch the function working set in kernel-space.

Capturing the working set. To capture the function's working set, we use kprobes [21] to hook the Linux kernel path that adds pages to the OS page cache. Specifically, we hook the function `add_to_page_cache_lru()`. kprobes allow for users to dynamically create hooks associated with kernel functions, where user-provided eBPF programs can be attached to. eBPF programs attached to such hooks are triggered whenever the associated function is executed, and are provided with an execution context, e.g., for function kprobes, the associated function arguments. For SnapBPF, the function arguments passed to the SnapBPF eBPF program include the file offset of the page that is about to be added to the page cache. In this way, once we attach the SnapBPF eBPF program to this hook, we are able to track the file offsets of the pages that are fetched into the page cache from the function snapshot.

The capture phase is then as follows. We spawn a new VM sandbox using the function snapshot. Before actually booting the VM sandbox, the VMM creates the kprobe, as described above, and attaches to it the SnapBPF eBPF capture program. Finally, it invokes the function to capture its working set. The SnapBPF eBPF capture program will be triggered for every page that is added to the system's page cache. Consequently, it has to filter out any pages that do not belong to the function snapshot file, i.e. the pages that are not fetched by the VMM. SnapBPF stores the filtered page offsets, which comprise the working set, in an eBPF map [22]. Additionally, Linux by default uses readahead to prefetch pages from disk and hide storage latency. Hence, we disable readahead in order to only fetch and capture the working set pages in this phase. Once the function invocation finishes, the VMM reads the offsets

from the eBPF map and stores them to disk. Note, that we only store the page offsets and not the pages themselves, as prior art does.

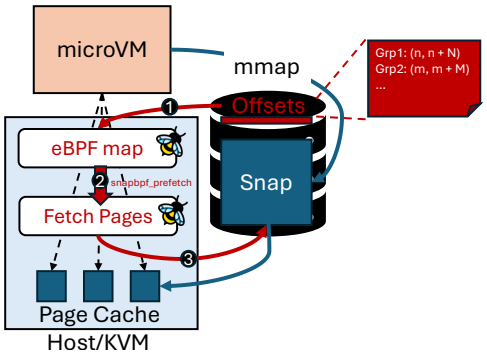


Figure 1: SnapBPF Prefetching. Note that SnapBPF captures the offsets of the working set pages, not the pages themselves. SnapBPF will fetch the pages directly from the snapshot file.

Loading the working set. Once the file offsets for the pages comprising the working set have been captured, we first group them into contiguous ranges of offsets and sort them based on the earliest access time of any of the pages in each group. We trigger the prefetching of the pages based on this sorted group order, ensuring that read requests for the pages needed the earliest are issued first.

Figure 1 shows the steps that load the working set from a function snapshot. When a new VM sandbox is spawned from the function snapshot, in order to handle an incoming function invocation, the VMM first reads the grouped file offsets of the function working set from disk and loads them into the kernel via an eBPF map ①. It then attaches the SnapBPF prefetch eBPF program to the same kprobe used earlier, and triggers the prefetching by accessing the first page of the snapshot ②. The SnapBPF prefetch eBPF program will then read the grouped offsets from the eBPF map and will start issuing consecutive read requests for each contiguous range of offsets from the snapshot file, in the afore-mentioned sorted order, to fetch them into the OS page cache ③.

As the Linux kernel sandboxes eBPF programs, which prevents them from, for example, issuing block requests to storage or manipulating the OS page cache, we implement an eBPF helper function, more specifically a kfunc [23] (`snapbpf_prefetch()`) ②, which wraps around the Linux page cache readahead routine that prefetches pages from storage (`page_cache_ra_unbounded()`). Once it issues the read request for the last group of offsets, the eBPF program will disable itself.

By issuing read requests directly to the snapshot file, SnapBPF obviates the need to separately serialize the working set to disk and instead only uses metadata to drive the prefetching. As we show in Section 4, this does not penalize performance, as, in contrast to spindle HDDs, modern SSDs don't have the same limitations with regard to high-IOPS, non-sequential I/O. Nonetheless, we do minimize the number for block requests the kernel issues to storage by grouping the pages into contiguous ranges, to reduce SW overhead. Finally, since the pages are loaded directly into the page cache, they are shared between multiple concurrent VM sandboxes for the same function, minimizing memory usage. Since SnapBPF employs eBPF and essentially works in kernel-space, there is no need for redundant userspace copies of data from the page cache, which eliminates most of the page cache overhead, that forces prior art, such as REAP and Faast, to opt for direct IO instead.

3.2 PV PTE Marking

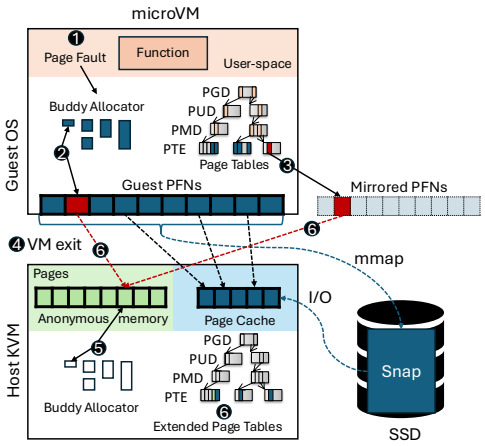


Figure 2: PV interface for VM memory allocations to avoid unnecessary IO.

When booting a VM sandbox from a snapshot file, the allocation of new pages by the VM guest OS memory manager will end up fetching pages from the on-disk snapshot, which will eventually be zeroed or overwritten. As mentioned in Section 2, prior art addresses this issue by preemptively scanning the snapshot, based on either page contents (FaaSnap [17]) or allocator metadata (Faast [16]). We instead adopt a different approach, that works online, without relying on snapshot scanning or pre-processing. We devise a lightweight paravirtualized (PV) PTE marking mechanism.

Figure 2 shows an overview of our proposed mechanism. We modify the VM (guest) kernel, so that when it attempts to allocate guest memory ① ②, it marks it in a way that the host (VMM) can detect it and use anonymous memory instead of fetching data from the snapshot file to back it up. Specifically, when the VM kernel attempts to map this

freshly-allocated memory in its page tables, instead of using the actual guest page frame number (gPFN) of the page, we set the most significant bit (MSB) of the PFN, effectively mirror-mapping this page to a higher PFN space ③.

The host kernel, specifically the Linux Kernel Virtual Monitor (KVM), when handling nested page faults for the VM ④, will be able to detect faults for such mirrored PFNs. In that case, it will use anonymous memory to serve the page fault, instead of fetching pages from the on-disk snapshot ⑤. It will then map this anonymous page to both the mirrored and the original gPFN, in the VM's nested page tables, so that when the VM subsequently reuses this memory, it also points to the anonymous page allocated by the host ⑥.

In this way, SnapBPF is able to handle the memory allocations of the VM sandbox without redundant I/O from the snapshot file or scanning the snapshot for pages that should be filtered or skipped before-hand. We also note that while we implement our PV PTE marking mechanism for Linux and KVM the design is essentially hypervisor and OS agnostic.

4 Evaluation

We evaluate SnapBPF and compare it with existing state-of-the-art prefetching approaches, namely REAP and FaaSnap.

Methodology. We implement SnapBPF on Linux v6.3 and the firecracker VMM v1.11 [9]. We use functions representative of common FaaS workloads from the FunctionBench [24] suite, as well as three real-world workloads from FaaSMem [25] (html_serving, graph_bfs, bert). We instrument firecracker to track the end-to-end latency for function invocations. We experiment with a single function instance as well as with 10 concurrent function instances, invoking them with identical inputs, to showcase the benefit of the deduplication and snapshot pages sharing enabled by SnapBPF . We consider evaluating the effect of varying function inputs on SnapBPF's memory deduplication for future work.

For the SnapBPF evaluation (Figure 3), we enable both the eBPF prefetching and the PV PTE marking mechanisms. We then provide a breakdown of the effect of each mechanism in Figure 4. For the Linux readahead baseline, we set the readahead window to the default Linux kernel value of 128KiB, i.e., 32 4KiB pages.

Hardware Setup. We evaluate SnapBPF on a 2-socket AMD EPYC 7402 CPU [26], with 24 hyperthreaded cores per socket. Each socket has access to 128GiB of DDR4 memory. We use a 480GiB Micro 5300 TLC NAND flash SATA SSD [27] to store the function memory snapshots for all methods and the function working sets for FaaSnap and Faast. To minimize noise, we pin the VMM threads on specific cores of the first socket. We also disable hyperthreading and set the CPU core frequency to 2.5GHz.

Latency. In Figure 3a, we show the end-to-end latency of REAP, FaaSnap and SnapBPF, when executing a single function instance. SnapBPF outperforms REAP, as it doesn't have to copy pages from userspace to kernel-space via userfaultfd. It also matches and in some cases outperforms FaaSnap in terms of E2E function invocation latency, by avoiding the redundant copying of the working set to userspace in the prefetching phase, and by maintaining leaner working sets, similar to REAP.

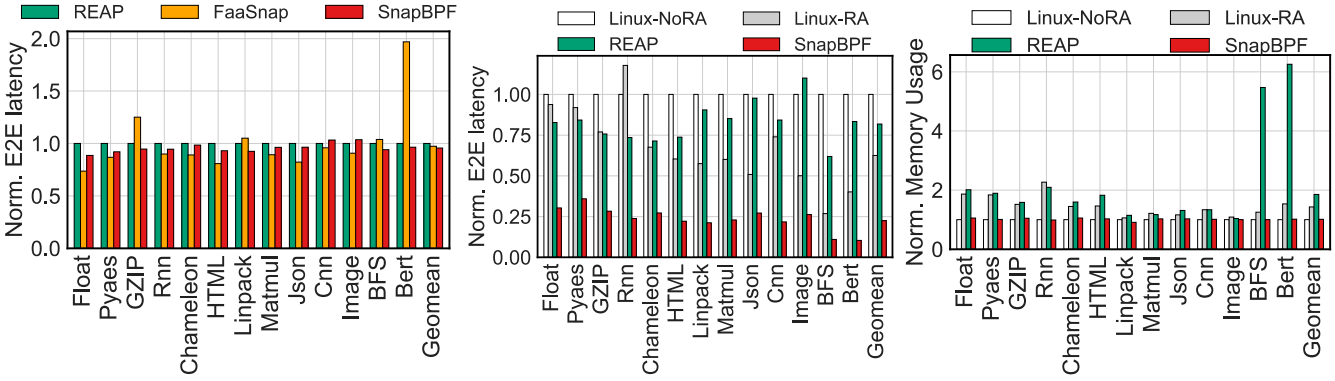
The shortcomings of userfaultfd-based approaches are more pronounced when executing 10 concurrent instances of the same function, where deduplication and sharing of snapshot pages comes into play. Figure 3b shows the E2E latency for this scenario. We compare SnapBPF with vanilla firecracker (no snapshot page prefetching) with Linux readahead both disabled (*Linux-NoRA*) and enabled (*Linux-RA*), as well as with REAP. SnapBPF outperforms vanilla firecracker as it efficiently prefetches the offsets representing the invocation working set. Moreover it outperforms REAP because it enables deduplication of the snapshot pages and sharing them among all 10 function instances. Notably, for functions with large working sets, such as Bert, SnapBPF is able to achieve 8x lower E2E latency than REAP.

Memory. Figure 3c shows the system-wide memory usage when running 10 concurrent VM sandboxes of the same function. Userfaultfd-based approaches are unable to deduplicate the working set between different sandboxes, leading to increased memory usage with concurrent function invocations. In this scenario, SnapBPF reduces memory usage by up to 6x for functions with large working set, such as BFS and Bert.

During the experiment, we observed that Linux KVM would some times result in excessive Copy-on-Write allocations, when handling nested page faults, which would diminish the deduplication benefits. We found out that this was due to the fact that KVM would under certain circumstances forcibly handle *read* nested page faults as *write*. This in turn forced the host kernel to CoW the page cache pages to anonymous memory. Consequently, we patched KVM to only opportunistically write-map read nested page faults, i.e., doing it only for faulted-in and already writable pages.

Breakdown Analysis. Figure 4 breaks down the effect of our eBPF prefetching and PV PTE marking mechanisms, in terms of function execution latency, when using firecracker to restore and invoke functions from a snapshot. We use the default Linux readahead behavior (*Linux-RA*) as the baseline and show the speedup achievable when using i) only the PV PTE marking mechanism (pink bar) and ii) PV PTE marking combined with eBPF prefetching (red bar).

Functions that during their invocation allocate large amounts of memory, see significant improvements from our PV PTE marking mechanism, as it is able to redirect the nested page faults for such allocations to anonymous memory and



(a) E2E function latency for a single function instance.

(b) E2E function latency for 10 concurrent function instances.

(c) Memory consumption for 10 concurrent function instances.

Figure 3: SnapBPF matches and outperforms user-space solutions without requiring separate working set files.

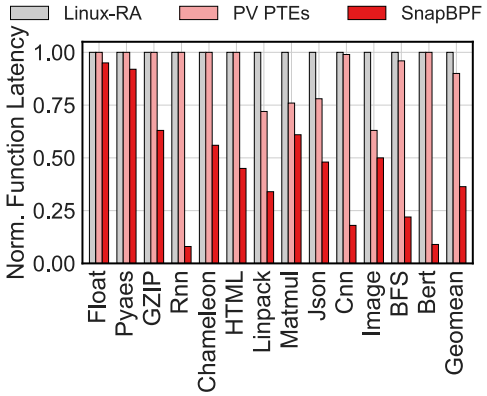


Figure 4: Breakdown analysis of the PV PTE marking and eBPF prefetching SnapBPF mechanisms.

eliminate the unnecessary fetching of pages from the snapshot file. The E2E latency for the image processing function (Image), for example, is improved by more than 2x. Note that SnapBPF is able to achieve this without having to keep track of and scan for stale or unused pages in the snapshot as prior art does. On the other hand, functions that rely on initialized state, e.g., models, like RNN and Bert, benefit only minimally, if at all, from the optimization of anonymous memory allocations. For these functions, the optimized working set prefetching is the dominant factor.

SnapBPF Overheads. We measure the latency of loading the offsets into the kernel, via the eBPF map, to be less than 1% of E2E latency on average (~1-2ms). We consider a comprehensive analysis of the computational and memory costs of SnapBPF for future work.

5 Related Work

eBPF. Similarly to SnapBPF, other works [28–30] have proposed using eBPF to make the OS page cache programmable, albeit targeting different use cases. eBPF has also been explored for filesystems [31], storage functions [32], and extending the OS memory manager [33].

Page Cache. Previous research has also focused on optimizing page cache performance and prefetching [34–37], albeit without targeting userspace extensibility and programmability or the FaaS use case.

FaaS Snapshotting. Optimizing function snapshotting has also been studied outside the context of working set prefetching, taking advantage of HW acceleration to improve performance with snapshot compression [38, 39] or evaluating the performance of FaaS snapshotting on storage devices with different performance profiles [40]. For container-based sandboxing, disaggregated memory has also been explored to optimize FaaS snapshotting [41–43].

6 Conclusion

SnapBPF employs eBPF to enable the efficient and programmable snapshot prefetching for VM-sandboxed serverless functions in kernel-space. SnapBPF is able to match and even outperform state-of-the-art approaches, without requiring separately serializing and storing the function working sets. It is able to deduplicate function working sets in memory via the OS page cache, without redundant userspace copies, while also enabling the online filtering of VM-sandbox memory allocations via a lightweight paravirtualized PTE marking mechanism.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was funded by the European Union under Horizon Europe grant 101092850 (project AERO).

References

[1] Amazon Web Services. AWS Lambda, 2025. URL https://aws.amazon.com/lambda.

[2] Microsoft Azure. Azure functions, 2025. URL https://azure.microsoft.com/en-us/products/functions.

[3] Huawei Corporation. Huawei Cloud Functions, 2025. URL https://developer.huawei.com/consumer/en/agconnect/cloud-function/.

[4] Google Corporation. Google Serverless Computing, 2025. URL https://cloud.google.com/serverless.

[5] Alibaba Corporation. Alibaba Serverless Application Engine, 2025. URL https://www.aliyun.com/product/aliware/sae.

[6] Cloudflare Corporation. Cloudflare Workers, 2025. URL https://workers.cloudflare.com/.

[7] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX annual technical conference (USENIX ATC)*, 2018. URL https://www.usenix.org/conference/atc18/presentation/wang-liang.

[8] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. Serverless cold starts and where to find them. *arXiv preprint arXiv:2410.06145*, 2024.

[9] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020. URL https://www.usenix.org/conference/nsdi20/presentation/agache.

[10] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020. URL https://doi.org/10.1145/3373376.3378512.

[11] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. URL https://doi.org/10.1145/3445814.3446714.

[12] Firecracker Development Team. Firecracker Snapshotting, 2025. URL https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md.

[13] Amazon Web Services. Improving startup performance with Lambda SnapStart , 2025. URL https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html.

[14] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. Pronghorn: Effective checkpoint orchestration for serverless hot-starts. In *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024.

[15] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

[16] Yongshu Bai, Zhihui Yang, and Feng Gao. Faast: An efficient serverless framework made snapshot-based function response fast. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 174–185, 2024.

[17] Lixiang Ao, George Porter, and Geoffrey M Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022. URL https://doi.org/10.1145/3492321.3524270.

[18] Linux Kernel Documentation. Userfaultfd, 2025. URL https://docs.kernel.org/admin-guide/mm/userfaultfd.html".

[19] Linux Kernel Documentation. eBPF, 2025. URL https://docs.kernel.org/bpf/.

[20] Linux man pages. mincore(2), 2025. URL https://man7.org/linux/man-pages/man2/mincore.2.html.

[21] Linux Kernel Documentation. Kernel probes (kprobes), 2025. URL https://docs.kernel.org/trace/kprobes.html.

[22] Linux Kernel Documentation. Bpf maps, 2025. URL https://docs.kernel.org/bpf/maps.html.

[23] Linux Kernel Documentation. Bpf kernel functions (kfuncs), 2025. URL https://docs.kernel.org/bpf/kfuncs.html.

[24] Jeongchul Kim and Kyungyong Lee. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019. URL https://doi.org/10.1109/CLOUD.2019.00091.

[25] Chuhao Xu, Yiyu Liu, Zijun Li, Quan Chen, Han Zhao, Deze Zeng, Qian Peng, Xueqi Wu, Haifeng Zhao, Senbo Fu, and Minyi Guo. FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024. URL https://doi.org/10.1145/3620666.3651355.

[26] Inc. Advanced Micro Devices. Amd epyc 7402, 2019. URL https://www.amd.com/en/support/downloads/drivers.html/processors/epyc/epyc-7002-series/amd-epyc-7402.html.

[27] Micron Technology Inc. 5300 series sata nand flash ssd, 2019. URL https://advdownload.advantech.com/productfile/PIS/96FD25-ST1.9T-M53P/file/96FD25-ST19T-M53P_Datasheet20200120180650.pdf.

[28] Dusol Lee, Inhyuk Choi, Chanyoung Lee, Sungjin Lee, and Jihong Kim. P2cache: An application-directed page cache for improving performance of data-intensive applications. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023. URL https://doi.org/10.1145/3599691.3603408.

[29] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. {FetchBPF}: Customizable prefetching policies in linux with {eBPF}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024. URL https://www.usenix.org/conference/atc24/presentation/cao.

[30] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. Cache is king: Smart page eviction with ebpf. *arXiv preprint arXiv:2502.02750*, 2025.

[31] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[32] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. {XRP}:{In-Kernel} storage functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[33] Konstantinos Mores, Stratos Psomadakis, and Georgios Goumas. ebpf-mm: Userspace-guided memory management in linux with ebpf. *arXiv preprint arXiv:2409.11220*, 2024.

[34] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021. URL https://doi.org/10.1145/3447786.3456242.

[35] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[36] Pei Cao, Edward W Felten, Anna R Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer*

Systems (TOCS), 14(4), 1996.

[37] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, 1994.

[38] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. Sabre:{Hardware-Accelerated} snapshot compression for serverless {MicroVMs}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024. URL https://www.usenix.org/conference/osdi24/presentation/lazarev.

[39] Yuqiao Lan, Xiaohui Peng, and Yifan Wang. Snapipeline: Accelerating snapshot startup for faas containers. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024. URL https://doi.org/10.1145/3698038.3698513.

[40] Christos Katsakioris, Chloe Alverti, Vasileios Karakostas, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Faas in the age of (sub-) μs i/o: a performance analysis of snapshotting. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, 2022. URL https://doi.org/10.1145/3534056.3534938.

[41] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast {RDMA-codesigned} remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[42] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, et al. Trenv: Transparently share serverless execution environments across different functions and nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024.

[43] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. Cxlfork: Fast remote fork over cxl fabrics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025.