Expeditious High-Concurrency MicroVM SnapStart in Persistent Memory with an Augmented Hypervisor

Xingguo Pang, Yanze Zhang, and Liu Liu, *University of Macau;*
Dazhao Cheng, *WuHan University;* Chengzhong Xu
and Xiaobo Zhou, *University of Macau*

https://www.usenix.org/conference/atc24/presentation/pang

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Expeditious High-Concurrency MicroVM SnapStart in Persistent Memory with an Augmented Hypervisor

Xingguo Pang[1], *Yanze Zhang*[1], *Liu Liu*[1], *Dazhao Cheng*[2], *Chengzhong Xu*[1], *Xiaobo Zhou*[1*]

University of Macau[1] *WuHan University*[2]

Abstract

The industry has embraced snapshotting to tackle the cold starts and efficiently manage numerous short-lived functions for microservice-native architectures, serverless computing, and machine learning inference. A cutting-edge research approach FaaSnap, while innovative in reducing page faults during on-demand paging through prefetching the profiled working set pages into DRAM, incurs high caching overheads and I/O demands, potentially degrading system efficiency.

This paper introduces PASS, a system leveraging byte-addressable persistent memory (PMEM) for cost-effective and highly concurrent MicroVM SnapStart execution. PASS, functioning as a PMEM-aware augmented hypervisor in the user space, revolutionizes MicroVM memory restoration. It constructs complete address indexing of the guest memory mapped to single-tier PMEM space, enabling zero-copy on-demand paging by exploiting PMEM's direct access feature. This approach bypasses the cache layer and maintains guest OS transparency, avoiding invasive modifications. Experimental results, derived from real-world applications, reveal that PASS substantially decreases SnapStart execution time, achieving up to 72% reduction compared to the Firecracker hypervisor on the PMEM filesystem, and 47% reduction compared to FaaSnap. Moreover, PASS achieves double the maximum concurrency compared to both Firecracker and FaaSnap. It improves the cost-effectiveness by 2.2x and 1.6x over the Firecracker and FaaSnap, respectively.

1 Introduction

The cold start issue, characterized by the latency incurred during instance initialization, significantly impacts short-lived functions, leading to extended response times and negative user experiences [4, 15, 23, 26, 33]. To address this issue, the industry is increasingly adopting a snapshot-based approach, particularly in MicroVM environments. This approach, known as SnapStart, leverages a hypervisor feature to perform comprehensive memory state checkpointing of MicroVMs, storing these states as files. SnapStart dramatically reduces startup times by restoring a MicroVM's memory from a pre-saved snapshot, thus bypassing the time-consuming process of initializing and setting up dependencies from scratch. Beyond accelerating startup times, SnapStart also shortens overall execution times. This is particularly beneficial for short-lived functions in microservice-native architectures [13, 14, 18, 52], serverless computing frameworks [32, 36, 44], and machine learning inference workloads [21,47,51], where minimizing latency is crucial.

In current production platforms, such as AWS Lambda SnapStart [35], the MicroVM snapshot restoration process encounters a significant performance bottleneck. This issue stems from frequent page faults during on-demand paging, particularly problematic within modern tiered memory architectures. While lazy loading techniques are implemented to reduce initialization time and improve memory efficiency, they inadvertently cause a high frequency of page faults, which in turn, significantly slow down function execution.

FaaSnap, a forefront research approach cited in [7], introduces a non-blocking method that prefetches the profiled working set pages into DRAM, thus accelerating MicroVM SnapStart execution. This technique involves copying pages in batches of 1,024 into user-space memory buffers before remapping them, significantly reducing the overhead typically associated with page faults. However, each prefetching cycle involves a notable pre-warm time due to the movement of data from disk to DRAM. This becomes inefficient, especially for ephemeral workloads where the majority of pages are accessed only once, leading to the underutilization of resources by caching snapshots into DRAM. Furthermore, the reliance on DRAM caching limits the capacity for concurrent SnapStart executions. The intensive prefetching also demands a higher level of I/O, in contrast to on-demand methods like AWS Lambda SnapStart, where pages are loaded more gradually. This approach, while reducing memory overhead, can interfere with concurrent workload disk operations, potentially deteriorating the overall system efficiency.

[*] The corresponding author. Email: waynexzhou@um.edu.mo.

In this paper, we explore the potential of leveraging Persistent Memory (PMEM), a byte-addressable memory device on the memory bus, to enable on-demand and direct data access for expeditious and high-concurrency MicroVM SnapStart in a cost-effective manner. Storing snapshots on single-tier PMEM, which is inherently persistent, allows for direct byte-addressable access. This capability makes it feasible to proactively establish page mappings from PMEM's memory space to a MicroVM's guest memory. Consequently, the MicroVM can directly access its snapshot memory pages in PMEM through these pre-built mappings, bypassing the need for repeated address translations and page table lookups. This proactive method offers the opportunity to efficiently tackle the page fault bottleneck, facilitating fast SnapStart execution. However, fully capitalizing on the potential of PMEM for efficient SnapStart poses three technical challenges.

Challenge I: Efficient page mapping in PMEM. Prefetching-based methods such as FaaSnap [7] and REAP [38] are tailored for tiered memory architectures and perform partial mapping of the profiled working set pages. However, these methods are not readily applied to single-tier PMEM systems. In PMEM, partial mappings are dispersed non-continuously, leading the OS to exert additional effort to merge these disjointed segments into complete mappings, which could negatively affect system performance. To fully leverage PMEM's capabilities for a MicroVM's SnapStart, it is essential to augment the hypervisor, enabling it to efficiently map PMEM's memory space to the guest's physical memory.

Challenge II: Direct data access in PMEM. Enabling the Direct Access (DAX) feature in a PMEM filesystem improves performance by bypassing cache intermediaries for direct data access. However, simply mounting snapshots on a DAX-enabled filesystem does not fully harness this advantage for MicroVM SnapStart. As the PMEM filesystem and the MicroVM each maintain separate address mapping tables for a snapshot memory file, a semantic gap arises in page recognition, leading to expensive DAX faults during a MicroVM's on-demand paging. Deactivating DAX avoids these faults but at the cost of losing PMEM's byte-addressable direct access, a key performance feature. Additionally, the overhead from converting between on-disk file formats and in-memory data structures can further impair SnapStart's efficiency. Consequently, the challenge lies in eliminating DAX faults while still preserving the direct access capability, which is essential for the efficient execution of MicroVM SnapStart.

Challenge III: Ephemeral workloads. Swapping data into DRAM from persistent storage devices introduces substantial overhead, particularly for non-iterative, single-access workloads such as those in linear DNN inference. These workloads produce many "ephemeral pages", data accessed temporarily and then quickly discarded. The core challenge lies in the inefficiency of existing caching-based methods, which create unnecessary replication overhead for these short-lived, single-use pages. For workloads rich in ephemeral pages, PMEM with its DAX capability provides an avenue for more streamlined single-use access by avoiding the replication overhead for data being cached into DRAM. Since ephemeral pages gain no lasting advantage from being cached, leveraging PMEM DAX could circumvent the redundancy and offer a more suitable solution for such fleeting data interactions.

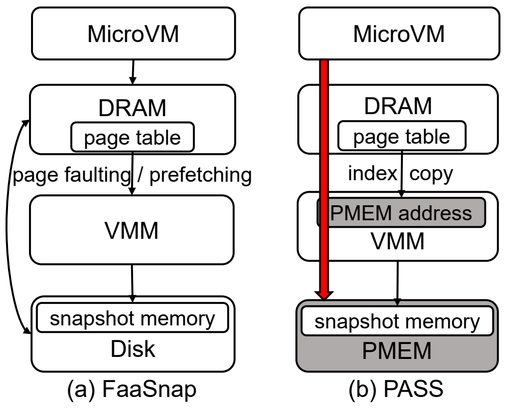

Figure 1: (a) FaaSnap employs page faulting or prefetching to copy memory pages from the secondary storage into DRAM to execute microVM workloads. (b) PASS allows for direct access to PMEM in a single-tier memory architecture.

We introduce PASS, a PMEM-aware augmented hypervisor designed to enable low-latency, high-concurrency SnapStart execution in MicroVMs by constructing complete address indexing of the guest memory during snapshot restoration. Utilizing the compatibility of byte-addressable PMEM and DRAM in accommodating the same page format, PASS allows direct access to restored snapshots in PMEM through memory load instructions. This innovative mechanism facilitates zero-copy on-demand paging, mapping guest memory directly to the PMEM in user space, bypassing both the cache layer and the PMEM filesystem. Leveraging hardware MMU mappings and a hash-based snapshot address table, PASS gains precise control over the PMEM in the user space, thereby greatly enhancing MicroVM SnapStart execution by fully exploiting the on-demand paging capabilities of byte-addressable PMEM in virtualized environments.

As illustrated in Figure 1, unlike FaaSnap which involves costly data movement across different memory tiers, PASS implements zero-copy on-demand paging that bypasses caching intermediaries. PASS not only minimizes page faults by pre-building complete guest memory mappings and leveraging PMEM DAX capability but also, crucially, eliminates DAX faults. This is achieved by treating PMEM as a primary memory, rather than as a disk-style device mounted in a PMEM filesystem. It is particularly advantageous in memory-constrained environments where, unlike FaaSnap which requires paging data to the storage, PASS's on-demand paging efficiently manages memory resources.

The key contributions of PASS are summarized as follows:

- PASS leverages single-tier PMEM's byte-addressability to proactively establish complete page mappings from PMEM's memory space to guest memory. This eliminates repetitive address translations and enables MicroVMs to directly access snapshot memory pages.
- PASS employs a zero-copy, on-demand paging mechanism with a hash-based address table, allowing MicroVMs to bypass caching layers and directly access PMEM-stored snapshots, thereby significantly enhancing SnapStart execution and memory resource efficiency.
- PASS has been implemented through augmentation of the Firecracker 1.4.1 hypervisor, utilizing KVM for x86 hardware-assisted virtualization. It ensures non-intrusive and guest OS-agnostic operations. The open-source artifact is available on GitHub for community use.

Our experimental framework employs the nine applications from Functionbench [17], a suite of practical function workloads widely used in public cloud services. The experimental analysis highlights PASS's significant performance enhancement. In SnapStart execution time, PASS surpasses the Firecracker hypervisor on the PMEM filesystem by up to 72% and FaaSnap by up to 47%. These improvements are even more notable in scenarios of high concurrency or memory pressure. Moreover, PASS achieves double the maximum concurrency of both Firecracker and FaaSnap, with 2.2x and 1.6x better cost-effectiveness, respectively. These results demonstrate PASS's potential as a highly efficient and economically viable option for production environments.

2 Motivational Studies

2.1 Page Fault Overhead in SnapStart

In this case study, we investigate the page fault overhead in the SnapStart execution, focusing on the approach currently used in production platforms, AWS Lambda SnapStart integrated within the Firecracker VMM. As in FaaSnap, we adopted the nine applications from Functionbench [17]; see experimental setup in 5.1. We utilized Firecracker v1.4.1 as the VMM, leveraging Linux KVM to create and manage MicroVMs. Each MicroVM is configured with 1 vCPU and 1GB of DRAM, a typical setup in a lightweight virtualization environment.

The experimental results, as depicted in Figure 2, demonstrate that page fault overhead contributes significantly to the SnapStart execution time, accounting for 40% to 60% whether the snapshot memory is mounted in a PMEM filesystem or SSD disks. This high overhead is primarily attributed to the inefficiency of the lazy loading scheme in handling page faults. A key factor behind this inefficiency is the lack of pre-established page mappings from the host to the guest's physical address space before SnapStart execution. To further underscore the significance of pre-established page mappings,

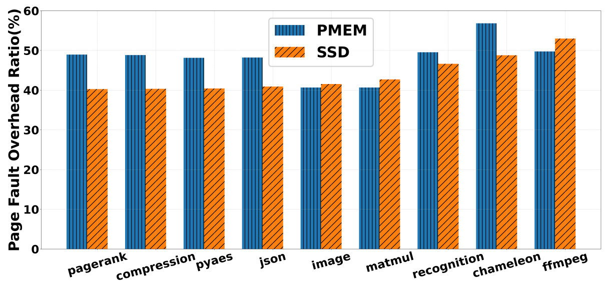

Figure 2: Page fault overhead in SnapStart execution.

Table 1: SnapStart performance on an ephemeral workload.

Approach \ Metric	execution time	memory footprint
Lambda SnapStart	240 ms	83 MB
FaaSnap	218 ms	243 MB
DRAM-cached	204 ms	1,024 MB

we conduct a comparative analysis of end-to-end execution latency between cached SnapStart and warm start scenarios. Our results find that cached SnapStart is, on average, over 30% slower than a warm start. This considerable latency difference highlights the impact of missing mappings on performance: without these pre-established mappings, page faults continue to incur substantial overhead during SnapStart execution.

2.2 SnapStart under Ephemeral Workloads

Ephemeral workloads, characterized by their one-time, non-iterative access to data, play a pivotal role in evaluating the efficacy of MicroVM SnapStart methods. This case study focuses on comparing the execution performance of such workloads, a json parse function from Functionbench with a 20MB data input, using three SnapStart approaches: Lambda SnapStart on a PMEM filesystem with DAX enabled, FaaSnap, and DRAM-cached snapshots which preloads the snapshot memory files into the in-DRAM filesystem before execution.

The performance of these approaches is quantified in Table 1. Lambda SnapStart on PMEM completed the task in 240 ms with a 83 MB memory footprint. DRAM-cached snapshots and FaaSnap completed the task more quickly, in 204 ms and 218 ms respectively, but used significantly more memory, 1,024 MB and 243 MB, respectively. Despite being slower by 15% and 9% than the DRAM-cached and FaaSnap approaches, Lambda SnapStart on PMEM with DAX enabled reduces DRAM usage by 92% and 66%. This marginal performance latency for short-lived ephemeral workloads with poor data reuse implies that the PMEM DAX capability is a compelling alternative. It offers considerable memory usage efficiency without a significant performance penalty, making it a promising solution for SnapStart execution.

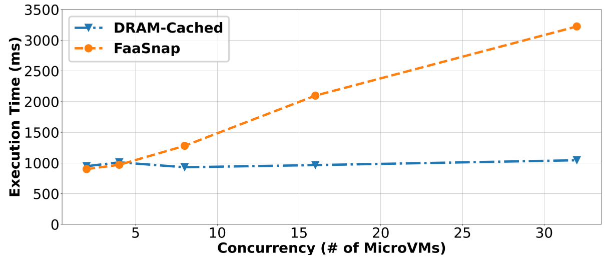

Figure 3: SnapStart execution time under high concurrency.

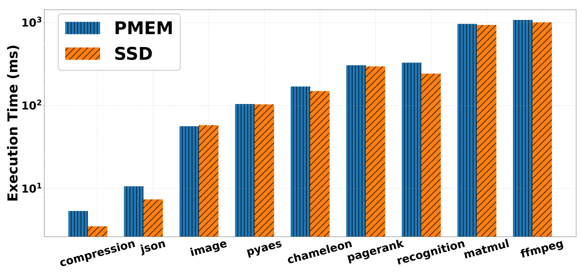

Figure 4: SnapStart execution time by FaaSnap.

2.3 SnapStart under High Concurrency

In this case study, we evaluate the performance of SnapStart execution under various concurrency levels, comparing FaaSnap with DRAM-Cached snapshots using the CNN image recognition workload. Figure 3 shows that the SnapStart execution time for DRAM-Cached warm start remains consistent even as the number of concurrent MicroVMs increases from 1 to 32. On the other hand, the performance of FaaSnap does not scale as well. Its average SnapStart execution time more than triples when the number of concurrent MicroVMs is increased from 1 to 32.

The observed decline in FaaSnap's performance under high concurrency levels is largely due to the interference from the I/O-intensive prefetching process, which becomes increasingly disruptive as more MicroVMs initiate SnapStart concurrently. This prefetching process, a key component of FaaSnap's design for anticipating future snapshot data needs, competes for resources with the ongoing memory operations of the MicroVMs. The resulting contention adversely affects the disk operations associated with the concurrent workload, leading to a significant reduction in the SnapStart efficiency.

2.4 FaaSnap in a PMEM Filesystem

In this case study, we evaluate FaaSnap's performance in a PMEM filesystem with DAX enabled. A MicroVM is configured with 2 vCPUs and 2GB of DRAM, a setup recommended for FaaSnap's prefetching method. The workloads are derived from the nine applications included in Functionbench.

Figure 4 presents an intriguing finding: for most workloads, the execution time of SnapStart when using the FaaSnap framework on PMEM with DAX is slower compared to its performance on SSD. This discrepancy prompted a deeper investigation into the underlying causes.

Our profiling process revealed a key difference in the way FaaSnap operates. On the PMEM filesystem, FaaSnap recorded zero working set pages because of the DAX feature. This outcome is linked to FaaSnap's dependency on tracking pages accessed in DRAM to determine its working set. However, in the SSD-based system, FaaSnap successfully profiled a significant number of working set pages. This variance in profiling capability helps explain the observed longer SnapStart execution time on PMEM with FaaSnap, highlighting its suboptimal performance in a DAX-enabled PMEM environment compared to SSDs. It's important to note that if the DAX feature is disabled, the performance characteristics of PMEM become more closely aligned with those of SSDs.

Table 2: Two types of kernel faults are observed: (1) page fault triggered by KVM, and (2) DAX fault from the ext4-dax PM filesystem due to KVM page fault handling.

Event	Time	Proportion
Eng-to-End Execution	517 ms	100%
Page Fault[a]	301 ms	58.2%
DAX Fault[b]	147 ms	28.4%

[a]Measured by kvm_mmu_page_fault.
[b]Measured by dax_iomap_pte_fault, a part of page fault overhead.

2.5 DAX Faults in a PMEM Filesystem

In this study, we evaluate the practicality of mounting a MicroVM memory snapshot on a PMEM filesystem to leverage the PMEM DAX feature [16]. Contrary to expectations, we find this method introduces significant page fault overhead.

As shown in Table 2, deploying a MicroVM snapshot on a PMEM filesystem for the CNN image recognition workload, a function from Functionbench, results in page faults dominating the execution time. Specifically, the handling of page faults, indicated by the `kvm_mmu_page_fault` event, accounts for 58.2% of the workload's total execution time. The DAX feature of a PMEM filesystem is designed to expedite data access. However, during page validation, if a page is missing, the page fault handler's retrieval of the page via the filesystem interface leads to conflicts with the DAX-mapped addresses, resulting in substantial DAX faults.

These DAX faults, logged by `dax_iomap_pte_fault` event, add significant overhead to the MicroVM when accessing the snapshot memory file on a PMEM filesystem. Our measurements indicate that they contribute to 28.4% of the workload's execution time. In essence, DAX faults constitute over half of the total page fault overhead, undermining the expected performance benefits of using a DAX-enabled filesystem for memory snapshots.

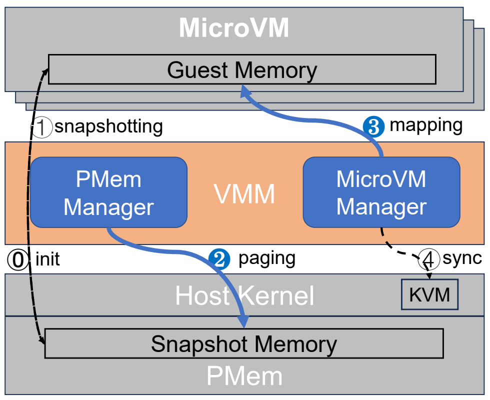

Figure 5: PASS architecture and workflow. PASS augments the Firecracker VMM by introducing two new components.

[Summary] The case studies reveal that 1) Page fault overhead critically hinders SnapStart execution time; 2) The DAX feature in PMEM can enhance memory access efficiency for certain ephemeral workloads; 3) High concurrency significantly degrades FaaSnap's performance; 4) FaaSnap's prefetching is incompatible with PMEM DAX feature; and 5) MicroVM memory snapshots on a PMEM filesystem underutilize DAX capabilities. Direct management of PMEM may optimize SnapStart's use of PMEM's potential.

3 PASS Design

3.1 Overview

PASS, a PMEM-aware augmented hypervisor, is designed to accelerate MicroVM SnapStart with high concurrency and cost-effectiveness. It leverages the byte-addressability and direct access capability of PMEM, bypassing the filesystem to fully unlock the hardware's inherent potential. PASS transforms MicroVM memory restoration by constructing the complete address indexes of the guest memory mapped to the PMEM space, thus enabling zero-copy and on-demand paging for direct data access. As illustrated in Figure 5, the PASS system comprises two principal components: the PMEM Manager and the MicroVM Manager. The PMEM Manager is tasked with allocating PMEM space for storing MicroVM snapshot memory states and managing a hash-based snapshot address table. The MicroVM Manager is in charge of MicroVM instances, which is responsible for pre-building a full, page-aligned address indexing for each MicroVM's memory after its allocation in PMEM.

The workflow of PASS involves five main steps in two phases. At the initial phase, ⓪ the PMEM Manager initializes the whole PMEM space for direct access via memory addresses and makes it ready for receiving and storing snapshotted memory. In the snapshotting phase, ① a MicroVM directly snapshots its memory state into the PMEM space allocated by the PMEM Manager, instead of storing the snapshot memory as a file. When a VM snapshot restoration request is received, ❷ the PMEM Manager allocates the appropriate memory space guided by its hash table for the guest memory use. The hash table contains the function name as the key and the base address of the snapshot memory with the offset as the value. Next, ❸ the MicroVM Manager pre-builds the full address index for the MicroVM's guest memory. Finally, ④ the MicroVM Manager registers the constructed address indexing in the KVM memory region for synchronization.

PASS's integrated management of PMEM allocation and snapshot restoration significantly reduces page fault overhead, facilitating efficient MicroVM SnapStart execution.

3.2 Pre-fault Page Mapping

To reduce cold start latency, Firecracker VMM employs a lazy loading strategy using the `mmap()` system call. This technique defers the mapping of physical memory pages to their corresponding virtual addresses until the first access attempt. The aim is to avoid the significant increase in cold start latency that can result from immediately populating the entire snapshot memory, including page tables, into DRAM. In a tiered memory architecture, this on-demand mapping is necessary because page tables derived directly from disk snapshots are unable to confirm memory mappings for a running MicroVM without inducing page faults.

As depicted in Figure 6(a), this lazy loading approach reserves a contiguous virtual address space without establishing actual physical-to-virtual page mappings until a page fault occurs. Consequently, when a MicroVM accesses memory pages during snapshot restoration, it encounters frequent page faults. Each fault incurs considerable overhead as the system must navigate the page table, locate the storage location of the page, and then populate the relevant page table entry. This method, however, overlooks the capabilities of modern PMEM devices, which allow data to be accessed directly, similar to DRAM—via memory load instructions, thereby enabling direct access through mapped memory addresses and avoiding the substantial overheads associated with data movement and page table entry population.

FaaSnap introduces a prefetching approach to expedite SnapStart by creating profiles of a workload's actively used pages, known as the working set. The set is stored and then utilized during subsequent SnapStart executions. As Figure 6(b) demonstrates, FaaSnap preemptively fetches and maps only these pre-identified working set pages from the snapshot memory to the guest VM's address space, optimizing the restoration process. However, this mapping strategy has a limitation. Pages not included in the working set, the "unrecorded" pages, do not get pre-mapped. Consequently, should a MicroVM attempt to access any of these unmapped pages, it will incur a page fault, leading to performance penalties. Therefore, while FaaSnap optimizes SnapStart for the known working

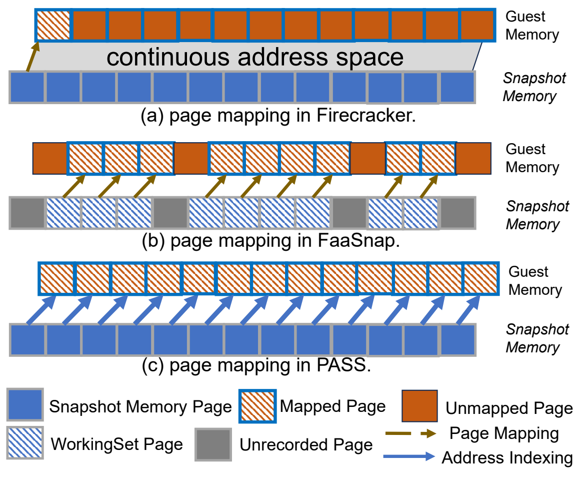

Figure 6: Page mapping in Firecracker, FaaSnap, and PASS. (a) Firecracker: validates mappings on the first guest page access. (b) FaaSnap: prefetches working set pages, leaving unrecorded pages of snapshot memory unmapped. (c) PASS: pre-faults complete address indexing for all guest pages.

set, it may not efficiently handle unexpected accesses to pages outside of this set, resulting in potential page faults.

PASS introduces a method for pre-fault page mapping, which involves mapping a MicroVM's snapshot memory onto PMEM prior to SnapStart execution. The innovation of PASS lies in its ability to fully index the guest memory addresses in advance, setting up all necessary mappings before the MicroVM is started. As demonstrated in Figure 6(c), unlike the partial mapping approach of FaaSnap, PASS ensures that all snapshot memory pages, including those previously unmapped, are assigned to the guest memory. This comprehensive mapping significantly reduces the incidence of page faults during operation.

In traditional systems, the process of mapping from a disk snapshot to guest memory is multilayered, involving separate logical and physical stages that can introduce latency. PASS, however, benefits from the single-tier architecture of PMEM, which simplifies this process. Logical mapping, the translation of PMEM snapshot pages to guest memory addresses, and the subsequent physical mapping occur concurrently in a single step. This streamlined approach contrasts with FaaSnap in the tiered memory architecture, where the logical mapping is a distinct step from the physical allocation of host memory and the copying of disk snapshot pages.

In PASS, address indexing is akin to OS page mapping. By pre-building address indexes, PASS facilitates a direct translation from guest memory to host PMEM, bypassing expensive data transfers and page table entry updates. This method of pre-fault page mapping capitalizes on the DAX capability of PMEM and is designed to streamline the lifetime management and synchronization of address indexing. The development of pre-built address indexing within PASS is underpinned by three critical sub-components: complete address indexing to establish all necessary mappings, robust management of the lifespan of these indexes, and meticulous synchronization of the indexing process. Each sub-component plays a pivotal role in ensuring the seamless operation of the pre-fault page mapping process in PASS.

Complete Address Indexing. Building partial address indexing for a MicroVM presents challenges due to the uncertainty in determining the appropriate subset of snapshot pages to map and their corresponding address ranges. This approach often relies on extensive profiling of access patterns, necessitating thousands of executions to gain prior knowledge. This can be especially burdensome for functions that frequently snapshot, as rapidly changing snapshots offer little opportunity for effective profiling. Moreover, if future access patterns diverge significantly, prefetching-based methods, like those used in FaaSnap, may lose their effectiveness.

In contrast, PASS constructs complete address indexing. This approach is both beneficial and cost-effective, as it involves merely constructing indexes without the need for caching pages to DRAM. The indexing process is relatively low-cost and can be completed quickly, considering that a typical MicroVM uses only 1-2 GB of guest memory. This is a stark contrast to the FaaSnap scheme, which involves mapping individual sub-regions of the snapshot memory separately before combining them into a complete mapping. Such a process introduces considerable scheduling overhead for the OS. PASS, however, achieves complete address indexing by sequentially mapping guest addresses directly to PMEM. We deliberately trigger page faults by accessing all virtual pages in advance for establishing the actual page mappings. Thereby PASS bypassed the significant overhead associated with the fragmented mapping approach of FaaSnap. Consequently, full-range indexing in PASS proves to be both practical and advantageous for managing MicroVM memory space.

Address Indexing Lifetime Management. In PASS, address indexing is constructed during the snapshot restoration process. This complete set of indexes remains active throughout the entire SnapStart execution of a MicroVM. Since the address indexing of the snapshot memory is pre-established, there is no need for dynamic updates during SnapStart. Upon the shutdown of a MicroVM, the indexing is automatically decommissioned. At this point, the MicroVM Manager communicates with the PMEM Manager to initiate the reclamation of the indexed PMEM region, subsequently marking this region as "available". This efficient lifecycle management of address indexing ensures that system resources are optimally utilized and promptly freed when no longer needed.

Address Indexing Synchronization In PASS, the guest memory is concurrently managed by both the VMM and the KVM. To prevent inconsistency issues, such as violations of the Extended Page Table, PASS implements a synchronization mechanism. This mechanism aligns the address indexing created in the user space with the corresponding address space

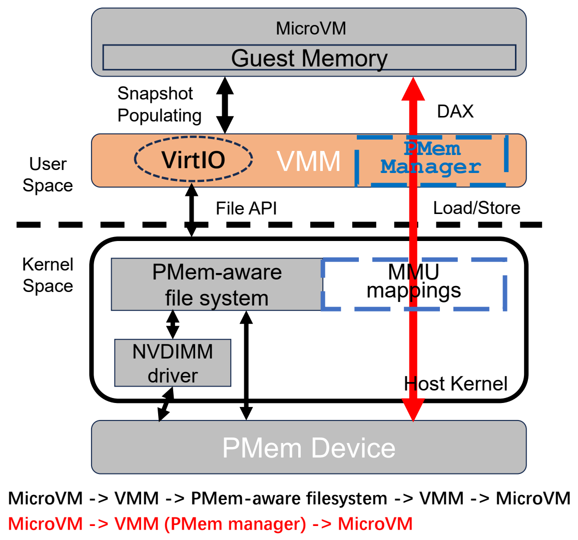

Figure 7: Datapath for on-demand memory paging. Left: the black lines represent the datapath in Firecracker on the PMEM filesystem. Right: the red lines represent the datapath in PASS.

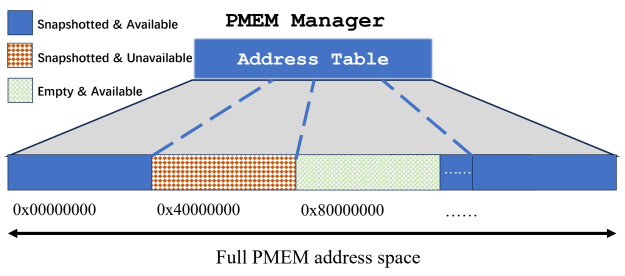

Figure 8: Snapshot memory management via the hash-based address table in PASS's PMEM Manager.

in the host kernel of the MicroVM. By bridging the semantic gap between user-space and kernel-space representations of the guest memory, this synchronization ensures that the address indexing in PASS can be accessed both reliably and consistently during MicroVM operations. This is crucial for maintaining the integrity and efficiency of the memory management system in a virtualized environment.

3.3 Zero-copy On-demand Paging

PASS introduces an innovative zero-copy on-demand paging mechanism. This mechanism uniquely manages PMEM snapshots directly in user space, bypassing the kernel. This approach avoids traditional filesystem interactions, instead utilizing direct hardware MMU mappings. Figure 7 illustrates the datapath for on-demand memory paging. In Lambda SnapStart, as used in Firecracker on the PMEM filesystem, the datapath involves four stages: guest memory to VMM, VMM to PMEM-aware filesystem, back to VMM, and finally to guest memory, with three transitions between user and kernel spaces. In contrast, PASS simplifies this process to just two stages: guest memory to VMM (serving as PMEM Manager) and back to guest memory, reducing the transitions to only one user-kernel switch. This streamlined approach enhances efficiency and reduces latency in memory management.

The paging mechanism in PASS is zero-copy in the sense that it involves only the copying of page indexes, not the actual memory pages. This operation is efficiently executed by the MicroVM, which directly accesses pages using their indexes. The guest OS recognizes these pages as allocated by the PMEM Manager and updates its page tables to reflect this. By not requiring the immediate transfer of the snapshot memory state into volatile DRAM, PASS significantly reduces the overhead associated with the multi-stage memory snapshot population, a common issue in other systems like Firecracker VMM and FaaSnap.

The effectiveness of zero-copy on-demand paging in PASS hinges on its proficient management of PMEM space. This is achieved by the system's PMEM Manager. By mapping guest memory pages directly to snapshot memory pages, PASS circumvents the host OS's page caching overhead. Furthermore, by bypassing the PMEM filesystem, the system effectively avoids DAX faults that typically occur during PMEM device access. This bypass is achievable because the snapshot memory size remains constant, and the memory for each MicroVM is continuously presented in PMEM, allowing effective space management via byte-addressable memory management.

The PMEM Manager in PASS is designed for specialized management of the PMEM device, utilizing a hash-based snapshot address table, as depicted in Figure 8. This table allows the user-space Manager to directly control PMEM operations, eliminating the need for frequent transitions to kernel space and reducing multi-stage interactions. Control over the PMEM is established through hardware MMU mapping. This occurs during the PMEM space initialization phase, wherein the PMEM Manager collaborates with the OS. The hash-based address table plays a pivotal role by recording the state of the PMEM and the address space of the snapshot memory pages stored within the PMEM. This information is crucial for the efficient execution of SnapStart. In the following, we will detail the PMEM management process which includes four core functions or areas of responsibility integral to the PMEM Manager's operation.

PMEM Space Initialization. We configure the PMEM device as a *devdax* device, facilitating direct access to the persistent storage. The PMEM Manager, responsible for the PMEM device space, first secures permission from the OS for access. Subsequently, it maps the entire PMEM device space by traversing and obtaining the hardware MMU mappings for each physical page on the PMEM. This completion of mappings readies the PMEM Manager for critical functions like snapshotting and restoring MicroVM memory states, integral for the efficient execution of the SnapStart function.

Table 3: Comparison of PASS, Firecracker, and FaaSnap.

Approach / Metric	**Firecracker**	**FaaSnap**	**PASS**
Profiling	No	Yes	No
Mapping	No	Partial	Complete
On-demand	Yes	Partial	Yes
Zero-copy	No	No	Yes
PMEM support	Yes	No	Yes

PMEM Space Allocation. *Snapshot Storage*: PASS builds upon the snapshotting mechanism of Firecracker VMM but copies the snapshot memory to the native byte-addressable PMEM. The PMEM Manager allocates continuous PMEM space for storing each snapshot memory. Utilizing the hash-based address table, it records the address space of these snapshots. The allocation of new or existing continuous PMEM space depends on if the snapshot memory is newly generated or needs to be updated with existing content. As depicted in Figure 8, for content updates, the Manager locates and updates the existing address space. During snapshotting, this space is marked as "snapshotted and unavailable" to prevent overlapping requests. For new snapshots, the PMEM Manager assigns an "empty and available" continuous address space.

Snapshot restoration: To restore a MicroVM's snapshot, the PMEM Manager references the hash-based address table to locate the necessary data. This enables the MicroVM to directly interact with the mapped PMEM pages, avoiding traps or faults. The PMEM Manager can allocate an available PMEM region for any MicroVM at any time, marking it as "snapshotted and available" in Figure 8. If a snapshot memory is not recorded in the address table, the Manager does not allocate PMEM space for its restoration.

Page-granularity Partitioning. As illustrated in Figure 8, the PMEM Manager ensures that the starting and ending addresses of the PMEM region, which are mapped to the guest memory, align with page size boundaries (typically 4KB). This alignment is crucial for enabling precise page-level mapping. After the restoration of its snapshot memory, the MicroVM's kernel autonomously partitions this snapshot memory into pages. This partitioning adheres to the page size specified by the PMEM Manager, which is designed to be identical to the page size used in the host OS. This alignment ensures consistency in memory management across both the MicroVM and the host system.

PMEM Space Reclamation. After the completion of the SnapStart execution, the PMEM Manager proceeds to recycle the PMEM space previously allocated to the MicroVM. It then marks this space as available for future requests. This status is indicated as "snapshotted and available" in Figure 8.

In summary, Table 3 presents a comparative analysis of PASS, Firecracker, and FaaSnap, highlighting their key characteristics relevant to SnapStart execution.

4 Implementation

We have implemented PASS by augmenting the Firecracker 1.4.1 VMM, utilizing KVM for x86 hardware-assisted virtualization. This implementation includes roughly 1,300 lines of Rust code integrated into the open-source Firecracker VMM. Additionally, we have developed a suite of tools comprising 1,500 lines of Python code for performance profiling and bottleneck identification in the system. The PASS artifact is open-source for community use on GitHub at `https://github.com/DISCOPASS/PASS`.

By embedding PASS functionality directly within the VMM layer, the system can efficiently manage both PMEM and guest memory, which is crucial for rapid snapshot booting processes. The memory page mappings are established during the initial launch of a MicroVM and are subsequently utilized for SnapStart executions.

Populating Page Tables. To build the guest memory space address index from PMEM, we create a *GuestMemoryMmap* list. Each element of this list is a *GuestRegionMmap* struct, comprising two main components: a struct that details the host memory region, and a 64-bit integer representing the guest address within the MicroVM. We iterate through this list to populate the page mappings for user-space memory, subsequently passing them to the KVM region through the `kvm_userspace_memory_region` interface in the Firecracker hypervisor. This setup enables quick translation of guest addresses to host physical addresses during MicroVM runtime, leveraging the pre-built address index.

Address Index Synchronization. In our system, while the KVM operates in the kernel space, the page table is constructed in user space. Merely populating these mappings in user space would lead to page faults since KVM in the kernel space is initially unaware of these mappings. To synchronize the user-space page table with the kernel's KVM, we utilize the Linux `userfaultfd` interface, which allows us to register the user-space mappings with the KVM. When the first page fault occurs, the interface intervenes to establish the full mapping, making it recognizable by KVM. Notably, our approach diverges from the standard usage of `userfaultfd`. Instead, we employ `userfaultfd` solely for the initial mapping initialization. Post this setup, the `userfaultfd` handler exits, avoiding ongoing operation and polling. This modification significantly reduces synchronization overhead.

PMEM Address Management. Linux cgroups do not offer a way to manage PMEM. We treat PMEM as primary memory in *devdax* mode, avoiding DAX faults. This necessitates the full mapping of the entire address space, as partial mapping would limit page access via `mmap()`. We developed the PMEM Manager in PASS, assigning a unique, non-overlapping address range to each MicroVM. For efficiency, these addresses are aligned with the MicroVM memory size boundaries, typically in 1GB increments. Thus, each starting address allocated by the PMEM Manager is a multiple of 2^{30}.

This strategy effectively prevents overlapping or out-of-bound allocations by concurrent users. As a result, PMEM can be directly `mmapped` and accessed safely throughout the MicroVM lifecycle, ensuring conflict-free operation.

5 Evaluation

5.1 Experimental Setup

Testbed. Our study utilizes the Intel® Optane™ PMEM architecture, a key player in persistent memory. It is configured with 128 (8x16) GB DDR4 DRAM; 512GB Intel Optane PMEM 200 series on an Interleaved four-DIMM configured in AppDirect mode; Intel® Xeon® Gold 5317 processor X2; Linux OS Ubuntu 22.04.3 LTS x86_64 with kernel 5.10.130.

The guest MicroVMs were allocated 1 vCPU and 1GB DRAM each, typical of serverless configurations in AWS Lambda. MicroVMs operate on Debian with kernel 5.14.

Prior to each test, we ensured a clean slate by dropping the page cache for all relevant files, including the snapshot memory file and the working set file. This step is crucial to ensure that our performance measurements accurately reflect PMEM reads, as opposed to cached DRAM accesses. By flushing the page cache, we eliminate potential biases from caching, thereby benchmarking the true PMEM access performance within the guest MicroVMs under a realistic environment.

Approaches. For comparison with PASS, we evaluated the following approaches for SnapStart execution:

- **Lambda SnapStart**: The standard approach on SSD.
- **Vanilla**: Lambda SnapStart on the PMEM filesystem (*ext4-dax*), enabling DAX for performance enhancement.
- **FaaSnap**: The state-of-the-art. It accelerates MicroVM SnapStart by employing a prefetching technique [7].
- **DRAM-Cached**: While effective, it is unsuitable for production platforms due to substantial memory demands.

Workloads As in FaaSnap, we adopted the nine applications from Functionbench [17]. Table 4 lists these functions, which encompass a diverse range of applications including web services, multimedia, scientific computing, machine learning, and graph processing. These workloads were chosen for comprehensive performance analysis across different scenarios.

Metrics. We evaluated the MicroVM SnapStart approaches using the following metrics: *SnapStart execution time*, *Concurrency*, and *Cost-effectiveness*.

5.2 SnapStart Execution Time

In this experiment, we evaluated the SnapStart execution time of five different approaches across nine workloads. As shown in Figure 9, PASS consistently outperforms or matches the performance of FaaSnap, improving SnapStart execution time by 1% to 47%. Against Vanilla, PASS shows even more substantial improvements, reducing SnapStart execution times by 3% to 72% across various workloads. Moreover, PASS closely matches the performance of DRAM-Cached, showcasing its efficiency. Specifically, in the workload "matmul", which represents a worst-case scenario, PASS's performance was less than 3% behind DRAM-Cached, underscoring its robustness even in the most demanding situations.

A deeper dive into PASS's performance reveals its strengths. For instance, in ephemeral workload "recognition", PASS is 47% and 72% faster than FaaSnap and Vanilla, respectively. FaaSnap's prefetching technique, although innovative, introduces contention with the process of guest memory page fetching, particularly in the case of the single vCPU configuration. This contention adversely impacts the critical path of guest memory page fetching in scenarios where caching is less beneficial. In contrast, PASS's approach of complete mapping of the guest memory with zero-copy on-demand paging circumvents this issue, thus enhancing performance, especially in ephemeral workloads.

When compared to the DRAM-cached approach, PASS even outperforms under certain workloads like "image" and "recognition". This can be attributed to two main factors: firstly, the on-demand paging in both PMEM and DRAM is similarly fast; secondly, the page mappings in PASS are synchronized with the MicroVMs, providing a more efficient operation than the DRAM-cached approach, where page mappings are not as promptly synchronized.

Table 4: Functions used in the performance evaluation.

Function	Description	Input
Compression	file compression	file
Json	deserialize and serialize json	json
Image	rotate a JPEG image	JPEG
Pyaes	AES encryption	string
Chameleon	render HTML table	table size
Recognition	PyTorch ResNet image recognition	JPEG
PageRank	igraph PageRank	graph size
Matmul	matrix multiplication	matrix size
FFmpeg	apply grayscale filter	video

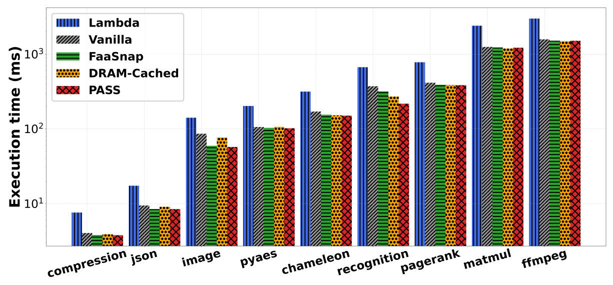

Figure 9: SnapStart execution time of different approaches.

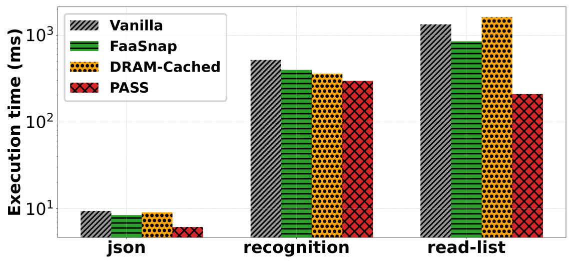

Figure 10: The execution latency of ephemeral workloads.

Table 5: A microscopic performance view of four approaches.

	Total time	Page fault	DAX fault
Vanilla	517ms	301ms	147ms
FaaSnap	441ms	218ms	0ms
DRAM-Cached	361ms	144ms	0ms
PASS	297ms	47ms	0ms

Due to the markedly lower performance of the standard Lambda SnapStart, it will not be included in further studies.

Ephemeral Workload. We extended the performance analysis of PASS to encompass additional categories, focusing on the management of ephemeral workloads. Figure 10 illustrates that PASS outperformed Vanilla (by 56% and 72%) and FaaSnap (by 38% and 47%) in the "json" and "recognition" tasks. We introduced a microbenchmark, "read-list", which entails reading a 512MB list once, page by page. This benchmark, depicted at the end of Figure 10, specifically evaluates the performance of ephemeral, single-pass workloads. In the "read-list" test, PASS demonstrated a substantial performance increase, executing 6.38 times faster than Vanilla and 4 times faster than FaaSnap. Overall, PASS significantly enhances efficiency in handling short-lived, single-pass workloads.

A Microscopic View. We examined the "recognition" workload to provide a detailed analysis of PASS, Vanilla, FaaSnap, and DRAM-Cached in the SnapStart execution time. As detailed in Table 5, the DRAM-Cached approach incurred a significant page fault overhead of 144ms (39.8%), markedly higher than the 47ms (15.8%) observed in PASS. This difference is attributed to DRAM-Cached's page mappings being established but not fully synchronized with the KVM. Moreover, each SnapStart in DRAM-Cached necessitates remapping pages due to the closure of snapshot file descriptors. FaaSnap prefetches pages from disk to DRAM but does not fully construct mappings to guest memory. As a result, FaaSnap still incurs significant page fault overhead of about 218ms, accounting for 49.6% of its total execution time. PASS, on the other hand, efficiently reduces page faults and avoids the overhead associated with disk-style DAX faults. Additionally, PASS minimizes page faults by pre-faulting page mappings through full address indexing, resulting in a more efficient performance than both Vanilla and FaaSnap approaches.

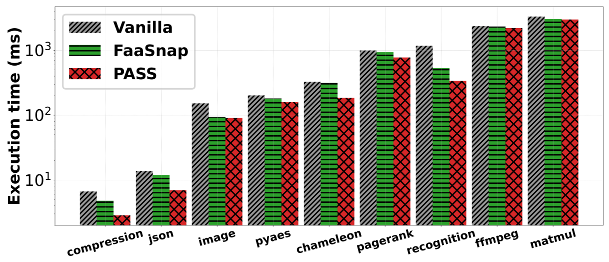

Figure 11: SnapStart execution time under a high concurrency.

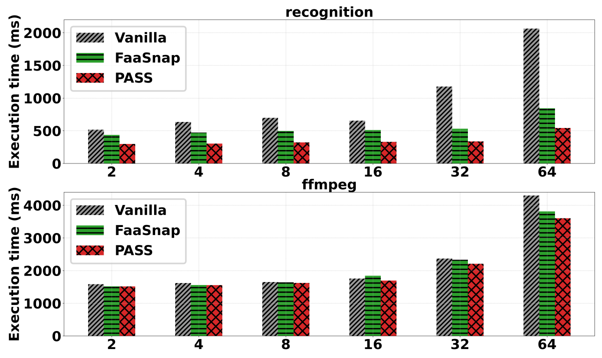

Figure 12: SnapStart execution under different concurrency.

5.3 High Concurrency

Impact of High Concurrency on SnapStart. We evaluated the performance of SnapStart approaches across the nine workloads, increasing the number of concurrent MicroVMs from 1 to 32. DRAM-Cached was excluded as it opposes SnapStart's aim to minimize warm instances and conserve DRAM resources. Figure 11 illustrates that PASS outperformed Vanilla (by 1.56x to 6.38x) and FaaSnap (by 1.38x to 4x) in the execution time under the concurrency level 32.

For a detailed analysis, we focused on "recognition" and "ffmpeg" workloads, incrementally increasing the number of concurrent MicroVMs from 2 to 64, in powers of 2. Under concurrency levels of 1-32, FaaSnap maintained relatively stable execution times, unlike the other approaches. At 32 concurrent tasks, PASS accelerated the execution by 1.6x and 3.5x compared to FaaSnap and Vanilla, respectively, for the "recognition" workload, as illustrated in Figure 12.

On the other hand, as also shown in Figure 12, in the initial stages of the "ffmpeg" workload, under concurrency levels of 1-32, PASS did not demonstrate much performance advantage over FaaSnap and Vanilla. The unique, resource-intensive nature of "ffmpeg", with its frequent video data sampling and repetitive computations, constrains the efficiency gains achievable by PASS. However, as the concurrency increased to 64, FaaSnap and Vanilla began to lag significantly behind PASS, likely due to their increased memory contention and processing overhead under high concurrency.

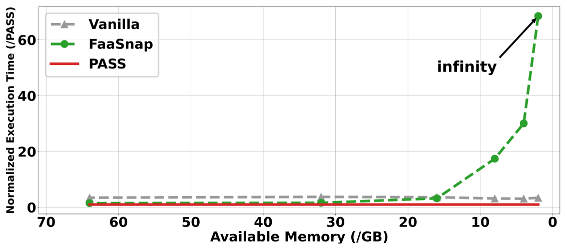

Figure 13: Execution time under varying memory pressure.

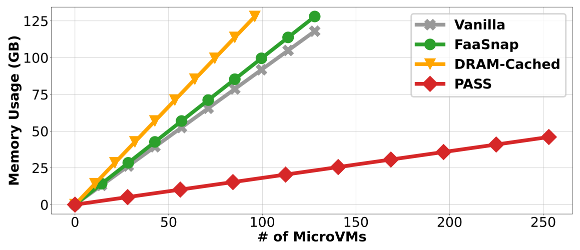

Figure 14: Maximum MicroVM concurrency.

PASS's superior high-concurrency performance is credited to its efficient PMEM memory management for swift allocation and zero-copy on-demand paging. This approach is increasingly advantageous at higher concurrency levels. When concurrency reaches 64, Vanilla and FaaSnap suffer performance drops due to prolonged OS memory and CPU allocation times. In contrast, PASS enables direct MicroVM access to a PMEM memory pool, requiring only CPU reallocation between batches, thus facilitating faster MicroVM execution even at a concurrency level of 64.

Impact of Memory Pressure on SnapStart. In this study, we assessed the performance of PASS, FaaSnap, and Vanilla under varying memory constraints, ranging from 64GB down to 2GB in powers of 2. We sent 32 requests for "recognition" functions to the server under varying levels of memory pressure. The CPU resources remained sufficient for precise performance measurements under 32 requests.

Figure 13 illustrates the performance differences, showing the execution times of FaaSnap and Vanilla relative to PASS. We observed that as memory availability decreased, PASS showed increasingly significant performance advantages over FaaSnap, ranging by 1.48x to 30x, and over Vanilla by about 3.5x. This is particularly notable in memory-constrained environments. For FaaSnap, when under less than 32 GB of available memory, the invocations appeared to run concurrently from a logical perspective but had to be executed sequentially in physical terms due to the memory resource limitation. In the worst-case scenario, where only 2GB of memory is available, FaaSnap is unable to run, and its execution time is recorded as "infinity". In such scenarios, FaaSnap's reliance on paging data to storage becomes a bottleneck, whereas PASS's strategy of on-demand paging proves to be more efficient in managing limited memory resources, thereby maintaining higher performance levels. Unlike FaaSnap, Vanilla also relies on PMEM DAX technology, so its performance is not sensitive to varying memory pressure like FaaSnap. However, compared to PASS, Vanilla relies heavily on the PMEM filesystem, which cannot fully unlock PMEM's performance for SnapStart applications like PASS is able to achieve.

Maximum Concurrency. To determine the maximum concurrency that PASS, FaaSnap, Vanilla, and DRAM-Cached can sustain, we sequentially launched MicroVMs until reaching the system's concurrency limit. During the test, the MicroVMs were not reclaimed. We sent individual calls to each MicroVM for the "recognition" function, and verified their proper operation based on the returned results. Every MicroVM that successfully launched and responded was counted as an active MicroVM. Figure 14 shows that PASS achieves the highest concurrency in launching MicroVMs. Specifically, PASS achieves 2x the maximum concurrency compared to both Vanilla and FaaSnap, and 2.63x improvement over the DRAM-cached approach. Interestingly, FaaSnap outperforms DRAM-cached because it prefetches less memory for each MicroVM, thereby conserving more DRAM resources. Vanilla achieves performance nearly comparable to FaaSnap. This is partly due to its use of PMEM DAX capability, which helps mitigate the increased memory pressure relative to FaaSnap. However, Vanilla, FaaSnap, and DRAM-cached all are ultimately constrained by the host's DRAM capacity.

In contrast, though it utilized about 48GB DRAM due to runtime overheads, PASS ensures that MicroVMs are not severely constrained in DRAM resources or subject to interference, by pre-allocating necessary PMEM space via its PMEM manager. This strategy allows PASS to maintain higher MicroVM concurrency compared to the other approaches.

Cost-effectiveness. The current unit cost of 128 GB DIMM DDR4 DRAM is \$16.61/GB, while the unit cost for 512 GB Optane PMEM 200 series is \$7.85/GB, and STAT SSD is \$0.35/GB. Considering that PASS, with approximately 48 GB of DRAM and 262 GB of PMEM, can double the maximum concurrency of both Vanilla and FaaSnap, its cost-effectiveness becomes apparent. In contrast, Vanilla requires 128 GB of DRAM and 130 GB of PMEM, and FaaSnap utilizes 128 GB of DRAM and 512 GB of SSD. Therefore, based on these configurations and costs, PASS achieves 2.2x and 1.6x improvements in cost-effectiveness over Vanilla and FaaSnap, respectively. This significantly underscores the economic advantage of PASS in production environments.

5.4 Discussions

Overhead. Before activating PASS, its PMEM manager must access all PMEM data to create MMU mappings for userspace control. This one-time setup incurs a minor initial delay, but it is conducted offline, ensuring no impact on PASS's online allocation of PMEM space for SnapStart.

Consistency Guarantee. Lambda SnapStart ensures data consistency firstly by triggering a page fault for synchronizing the DRAM data from the disk snapshot, and secondly through a copy-on-write mechanism that maintains data integrity during write operations in DRAM. PASS leverages the unique properties of PMEM to streamline this process. By utilizing a single-tier memory architecture where data is directly accessed from PMEM, PASS eliminates the need for hierarchical data flushes. Thus, it simplifies the consistency model by reducing the synchronization overhead typically required between DRAM and other storage media.

Intel Optane PMem Discontinuation. The design principles of PASS and its VMM enhancements are designed to be independent of any specific persistent memory hardware. While our experiments utilized Intel Optane PM, the architecture of PASS is flexible and readily adapted to alternative persistent memory technologies like CXL-attached memory.

6 Related Works

Cold Start. Various approaches have been proposed to mitigate cold starts, including container reuse, pre-warming, storage optimization, and snapshot-based solutions. Container reuse techniques, such as SAND [4], SOCK [28], and Pagurus [24], aim to repurpose warm but idle containers from previous function invocations. While effective in reducing start-up time, these techniques might compromise isolation. Pre-warming strategies, referenced in works like Orion [26] and Icebreaker [30], involve keeping instances ready for immediate use, but can be resource-intensive. Storage optimization methods, such as studies [6, 14, 18, 25, 29, 40, 41, 48], focus on efficient data management to speed up function deployment. An emerging approach is SnapStart which is particularly suitable for functions executed infrequently, where maintaining warm instances is less efficient. Our work, PASS, focuses on SnapStart in MicroVM environments.

SnapShotting. Snapshotting captures a system's complete state at a specific moment, enabling efficient restoration and replay of prior workload scenarios. Innovative snapshotting works are seen in Catalyzer [12], Firecracker [3], Fireworks [34], and FaaSnap [7], where snapshots are used to efficiently reproduce system conditions. These technologies are often paired with record and replay tools like ClusterRR [42] and Vidi [53] to facilitate this process. Furthermore, NVOverlay [43] aims to facilitate frequent snapshotting directly in PMEM. TreeSLS [46] represents an advancement in snapshotting technology. By leveraging single-level non-volatile memory, it reduces data movement overhead, significantly improving upon earlier systems like Aurora [37].

SnapStart. Efforts to enhance snapshot restoration, particularly in lightweight MicroVM environments like Firecracker, have seen significant advancements recently. Work in [8] optimized AWS Lambda SnapStart through efficient datapath management for file image transfers. While recent approaches like FaaSnap [7] and REAP [38] have explored prefetching memory pages to quicken SnapStart execution, they encounter limitations due to expensive data movement between memory tiers. This is where PASS comes into play, which leverages PMEM's byte-addressability and DAX capability for faster data access. Container-based snapshot restoration methods, as seen in works like [6, 10, 11, 39], offer effective solutions for applications where the overhead of full virtualization are unnecessary. Unikernels, in studies like [9, 49], are optimized for lightweight virtualization and snapshot restoration. PASS currently focuses on the utilization of MicroVMs.

PMEM. Optimizing memory use is crucial across various computing domain, particularly in cloud computing. Industry leaders like Intel [1] and MemVerge [2], along with research initiatives such as MEMTIS [20], vTMM [31], LL-FREE [45], and TIPS [19], focus on balancing capacity and latency by managing page migrations between DRAM and PMEM. PASS is specifically designed for accelerating SnapStart in cloud services using PMEM, distinct from the broader focus on heterogeneous memory coordination. Unlike PMEM file mapping efforts in DaxVM [5], cfFS [22], HashFS [27], and ArckFS [50], which optimize PMEM filesystems for quick cloud application access, PASS uniquely maps directly from the raw persistent device. It treats PMEM as the primary memory source for guest applications, leveraging its speed and persistence to enhance SnapStart performance, thereby streamlining memory architecture more effectively.

7 Conclusion

In conclusion, PASS revolutionizes MicroVM SnapStart execution by harnessing the full potential of PMEM's byte-addressable nature, delivering low-latency and high-concurrency performance. Our innovative PMEM-aware hypervisor design, with zero-copy on-demand paging and precise PMEM control through hardware MMU mappings and a hash-based address table, eliminates the bottlenecks of traditional address translation and DAX faults. Implemented in the Firecracker hypervisor and validated by Functionbench workloads, PASS significantly outperforms Lambda SnapStart and offers a superior alternative to FaaSnap. With its demonstrated enhancements in execution time, concurrency, and cost-effectiveness, PASS stands as a practical approach to efficient memory management for virtualized environments.

Looking ahead, we aim to extend PASS's capabilities to container and unikernel environments with additional real-world workloads, seeking to broaden its applicability.

8 Acknowledgement

The work was supported in part by the Science and Technology Development Fund, Macao S.A.R (FDCT) projects 0078/2023/AMJ and SKL-IOTSC(UM)-2024-2026.

References

[1] Intel memory-optimizer. https://github.com/intel/memory-optimizer. Accessed on July 2, 2023.

[2] MemVerge. https://www.memverge.net. Accessed on July 2, 2023.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In Proceedings of the USENIX NSDI, pages 419–434, 2020.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In Proceedings of the USENIX ATC, pages 923–935, 2018.

[5] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. Daxvm: Stressing the limits of memory as a file interface. In Proceedings of the IEEE/ACM MICRO, pages 369–387. IEEE, 2022.

[6] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In Proceedings of the ACM EuroSys, pages 398–415, 2023.

[7] Lixiang Ao, George Porter, and Geoffrey M Voelker. FaaSnap: FaaS made fast using snapshot-based VMs. In Proceedings of the ACM EuroSys, pages 730–746, 2022.

[8] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in *AWS* lambda. In Proceedings of the USENIX ATC, pages 315–328, 2023.

[9] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip redundant paths to make serverless fast. In Proceedings of the ACM EuroSys, pages 1–15, 2020.

[10] Wei Chen, Rao Jia, and Xiaobo Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In Proceedings of the USENIX ATC, pages 251–263, 2017.

[11] CRIU Dec. Checkpoint/restore in userspace, 2015.

[12] 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 ACM ASPLOS, pages 467–481, 2020.

[13] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In Proceedings of the USENIX ATC, pages 419–432, 2023.

[14] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In Proceedings of the ACM ASPLOS, pages 152–166, 2021.

[15] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In Proceedings of the ACM SoCC, pages 443–458, 2023.

[16] 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 ACM SYSTOR, pages 13–25, 2022.

[17] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In Proceedings of the IEEE CLOUD, pages 502–504. IEEE, 2019.

[18] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In Proceedings of the USENIX ATC, pages 805–820, 2021.

[19] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. Tips: making volatile index structures persistent with dram-nvmm tiering. In Proceedings of the USENIX ATC, pages 773–787, 2021.

[20] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In Proceedings of the ACM SOSP, pages 17–34, 2023.

[21] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. TETRIS: Memory-efficient serverless inference through tensor sharing. In Proceedings of the USENIX ATC, pages 473–488, 2022.

[22] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. {ctFS}: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In Proceedings of the USENIX FAST, pages 35–50, 2022.

[23] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In Proceedings of the USENIX ATC, pages 53–68, 2022.

[24] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In Proceedings of the USENIX ATC, pages 69–84, 2022.

[25] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. SONIC: Application-aware data passing for chained serverless applications. In Proceedings of the USENIX ATC, pages 285–301, 2021.

[26] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In Proceedings of the USENIX OSDI, pages 303–320, 2022.

[27] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking file mapping for persistent memory. In Proceedings of the USENIX FAST, pages 97–111, 2021.

[28] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In Proceedings of the USENIX ATC, pages 57–70, 2018.

[29] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In Proceedings of the USENIX NSDI, pages 193–206, 2019.

[30] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In Proceedings of the ACM ASPLOS, pages 753–767, 2022.

[31] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vtmm: Tiered memory management for virtual machines. In Proceedings of the ACM EuroSys, pages 283–297, 2023.

[32] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In Proceedings of the USENIX ATC, pages 205–218, 2020.

[33] Jiuchen Shi, Kaihua Fu, Quan Chen, Changpeng Yang, Pengfei Huang, Mosong Zhou, Jieru Zhao, Chen Chen, and Minyi Guo. Characterizing and orchestrating vm reservation in geo-distributed clouds to improve the resource efficiency. In Proceedings of the ACM SoCC, pages 94–109, 2022.

[34] 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 ACM EuroSys, pages 663–677, 2022.

[35] AWS Lambda SnapStart. https://docs.aws.amazon.com/lambda/latest/dg/API_SnapStart.html, 2023.

[36] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In Proceedings of the ACM ISCA, pages 1–15, 2023.

[37] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora single level store operating system. In Proceedings of the ACM SOSP, pages 788–803, 2021.

[38] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Proceedings of the ACM ASPLOS, pages 559–572, 2021.

[39] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In Proceedings of the ACM MEMSYS, pages 53–65, 2019.

[40] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In Proceedings of the USENIX ATC, pages 443–457, 2021.

[41] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In Proceedings of the ACM EuroSys, pages 1–16, 2019.

[42] Wei Wang, Zhiyu Hao, and Lei Cui. Clusterrr: a record and replay framework for virtual machine cluster. In Proceedings of the ACM VEE, pages 31–44, 2022.

[43] Ziqi Wang, Chul-Hwan Choo, Michael A Kozuch, Todd C Mowry, Gennady Pekhimenko, Vivek Seshadri, and Dimitrios Skarlatos. Nvoverlay: enabling efficient and scalable high-frequency snapshotting to nvm. In Proceedings of the ACM/IEEE ISCA, pages 498–511. IEEE, 2021.

[44] Ziqi Wang, Kaiyang Zhao, Pei Li, Andrew Jacob, Michael Kozuch, Todd Mowry, and Dimitrios Skarlatos. Memento: Architectural support for ephemeral memory management in serverless environments. In Proceedings of the IEEE/ACM MICRO, pages 122–136, 2023.

[45] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, Daniel Lohmann, Dominik Töllner, Christian Dietrich, Illia Ostapyshyn, Florian Rommel, Daniel Lohmann, et al. Llfree: Scalable and optionally-persistent page-frame allocation. In Proceedings of the USENIX ATC, volume 65, 2023.

[46] Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. Treesls: A whole-system persistent microkernel with tree-structured state checkpoint on nvm. In Proceedings of the ACM SOSP, pages 1–16, 2023.

[47] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: A native serverless system for low-latency, high-throughput inference. In Proceedings of the ACM ASPLOS, pages 768–781, 2022.

[48] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In Proceedings of the ACM SoCC, pages 1–12, 2019.

[49] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. {KylinX}: A dynamic library operating system for simplified and efficient cloud virtualization. In Proceedings of the USENIX ATC, pages 173–186, 2018.

[50] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace nvm file systems with the trio architecture. In Proceedings of the ACM SOSP, pages 150–165, 2023.

[51] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. {PetS}: A unified framework for {Parameter-Efficient} transformers serving. In Proceedings of the USENIX ATC, pages 489–504, 2022.

[52] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In Proceedings of the ACM ASPLOS, pages 1–14, 2022.

[53] Gefei Zuo, Jiacheng Ma, Andrew Quinn, and Baris Kasikci. Vidi: Record replay for reconfigurable hardware. In Proceedings of the ACM ASPLOS, pages 806–820, 2023.