

Speeding up FaaS Warm Starts with Memory Restoration Templates

Aravinda Prasad*

Intel
Bengaluru, India

Sandeep Kumar

Intel
Bengaluru, India

Sreenivas Subramoney*

Intel
Bengaluru, India

Antti Kervinen

Intel
Tampere, Finland

Abstract

Function-as-a-Service (FaaS) is a cloud-native model where small pieces of application logic execute as isolated functions within virtual machines (VMs). Cloud Service Providers (CSPs) aim to maximize the number of VMs per system. However, idle VMs, i.e., VMs without active functions, continue to consume memory, limiting overall VM density.

Rather than using prior disk-based snapshotting solutions to reduce memory usage, we propose to place the pages of an idle VM in a compressed memory pool. Doing so, however, incurs high execution latency upon new invocations due to page decompression during on-demand page faults. To address this, we propose MRT. MRT generates *memory restoration templates* from the learned sequence of pages that incur page faults during a warm start and uses them to bulk-restore pages using Intel's In-Memory Analytics Accelerator (IAA), avoiding costly on-demand faults.

MRT lowers single-instance function execution latency from 2.63× to 1.62× relative to a hot start, eliminating 62% of the overhead that on-demand page-fault-based restoration adds over a hot start. MRT further reduces the memory footprint of an idle FaaS VM instance by 57% (geometric mean) by placing its pages in the compressed memory pool, allowing CSPs to host more active VMs per system. At scale, MRT reduces 95th- and 99th-percentile function execution latency by 70% and 58%, respectively, relative to on-demand IAA decompression.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; *Heterogeneous (hybrid) systems*; *Special purpose systems*; • **Software and its engineering** → **Virtual machines**; *Operating systems*.

*Work done while at Intel.

DIMES '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *4th Workshop on Disruptive Memory Systems (DIMES '26), September 29-October 02, 2026, Prague, Czech Republic*, <https://doi.org/10.1145/3831664.3837781>.

Keywords: FaaS, serverless computing, memory compression, warm starts, virtual machines, accelerators, IAA

ACM Reference Format:

Aravinda Prasad, Sreenivas Subramoney, Sandeep Kumar, and Antti Kervinen. 2026. Speeding up FaaS Warm Starts with Memory Restoration Templates. In *4th Workshop on Disruptive Memory Systems (DIMES '26), September 29-October 02, 2026, Prague, Czech Republic*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3831664.3837781>

1 Introduction

Function-as-a-Service (FaaS) is a rapidly emerging cloud-native workload where users can run functions (a small portion of application logic) in response to events. Functions run in isolation inside lightweight sandboxes, ranging from microVMs such as Firecracker [2, 11] to user-space kernel sandboxes such as gVisor [4]. In this paper, we focus on lightweight virtual machines (VMs). Cloud service providers (CSPs) deploy thousands of active function instances by packing as many VMs as possible on a single physical machine to maximize system utilization [11, 34].

FaaS environments face challenges with memory management because, unlike CPU cycles, VMs consume memory even when they are idle, i.e., when there are no actively running function instances. This limits the number of active VM instances that can be spawned on a system. Adding additional DRAM is also prohibitive due to its cost, as memory accounts for 33–50% of the total cost of ownership (TCO) in modern data centers [31, 35].

To address the memory management challenge, the industry has embraced snapshotting VM images to disk [17, 18, 34]. However, disk-based snapshots present several challenges, including high restoration latency due to scattered, on-demand page faults served over slow disk reads [34]. To mitigate this issue, prior work [24] compresses snapshotted memory and leverages hardware accelerators, such as Intel's In-Memory Analytics Accelerator (IAA) [21], to speed up memory (de)compression. These accelerators provide a number of decompression engines and are already available commercially [29, 32]. Yet, IAA achieves peak performance only when many (de)compression jobs are issued in parallel

across its engines [32]. Serving a single page per fault on demand submits only one job at a time, effectively serializing restoration and leaving the accelerator underutilized, which results in significant performance degradation.

In this paper, we propose compressing and placing an idle VM's pages in a compressed memory pool [22] instead of on disk. However, upon a function invocation on such an idle VM, its pages must be decompressed before execution can proceed (referred to as a *warm start*), which introduces high latency. To mitigate this overhead, we propose generating *memory restoration templates* from the learned sequence of page addresses that incur page faults during a VM warm start. When a new FaaS function is scheduled on an idle VM, MRT, using the generated template, bulk-restores this sequence of pages from the compressed memory pool to DRAM by leveraging the decompression engines of the hardware accelerator in parallel. With MRT, most warm-start page faults and their associated guest-to-host VM exits are eliminated, thereby reducing VM warm-start time and function execution latency, which in turn enables CSPs to more aggressively place idle VMs in the compressed memory pool, improving VM density.

In summary, this paper makes the following contributions: ❶ We make the case for retaining idle FaaS VMs in an in-memory compressed pool, rather than snapshotting them to disk, and characterize the warm-start overheads this approach introduces. ❷ We propose MRT, which records the sequence of pages that fault during a warm start and bulk-restores them through parallel accelerator-driven decompression, eliminating most on-demand faults and their associated VM exits. ❸ We prototype MRT in the Linux kernel and evaluate it on a production-grade serverless stack with Intel IAA. MRT lowers single-instance warm-start latency from $2.63\times$ to $1.62\times$ the hot-start latency, eliminating 62% of the overhead introduced by on-demand page-fault-based restoration, reduces the memory footprint of idle VMs by 57% (gmean), and improves 95th- and 99th-percentile latency at scale by 70% and 58%, respectively.

2 Background

Function-as-a-Service. FaaS is a cloud model in which a small unit of application logic (the “function” in FaaS) is executed on demand and in isolation by a CSP. Functions implement data-processing logic such as business rules, algorithms, or machine-learning inference. In multi-tenant FaaS environments, a typical function is executed inside a lightweight sandbox for better security and isolation. Deployed isolation mechanisms span a broad spectrum: lightweight VMs (microVMs) such as Firecracker [2, 11] run a minimal guest kernel under a hypervisor, whereas sandboxes such as gVisor [4] interpose a user-space kernel between the sandboxed application and the host. We target VM-based isolation and use the term VM throughout the paper. MRT, however,

only requires that the sandbox memory be backed by a regular host process whose pages can be placed in zswap; it therefore also applies to process-backed sandboxes such as gVisor. CSPs host many FaaS VM instances of many users on the same server, scaling the number of running VMs up and down with the number of concurrent user requests.

Memory compression. Memory compression reduces application memory footprints by storing cold pages in compressed form. In Linux, this functionality is provided by the zswap subsystem [22, 36, 37], which compresses pages using a variety of algorithms [19, 25, 26] and stores them in a compressed memory pool [16]. Each compressed page is represented by a “swap entry” in its page table entry; accessing such a page triggers a page fault, upon which the kernel retrieves and decompresses the data from the pool, restores the page contents, and updates the page table accordingly. To accelerate these (de)compression operations, modern Intel CPUs include a dedicated hardware accelerator, the In-Memory Analytics Accelerator (IAA) [21, 29]. A processor may include multiple IAA devices, each comprising several independent (de)compression engines, and the Linux kernel can employ these devices as a zswap backend in place of software compression [32]. The accelerator attains peak throughput, however, only when many (de)compression jobs are issued to its engines in parallel [32]. A fault-driven swap-in path that submits one page at a time leaves this parallelism unexploited.

3 Motivation

To rapidly scale the number of active function instances, CSPs reuse idle VMs rather than booting fresh ones, as booting requires forking a new instance and initializing the guest OS. After serving a function request, a VM moves to the idle state, where it remains ready to handle incoming requests.

However, idle VMs, despite not serving any function execution requests, continue to consume memory. Upon a new function execution request, the CSP checks for an available idle VM for the corresponding user. If available, the request is assigned to one such VM. As the VM is in the idle state, such requests experience low execution latency and are referred to as *hot starts* [13]. If no idle VM is available, then the CSP boots a fresh VM instance, leading to significantly increased execution latency. Such requests are referred to as *cold starts* [27]. From Figure 1, it can be observed that cold starts incur significantly higher function execution latency, with a $62\times$ slowdown (geometric mean), and are therefore detrimental to performance. Across the evaluated functions, cold starts take 1.3–2.6 s, whereas hot starts complete within 3–281 ms; the difference is dominated by the cost of forking a VM instance and booting and initializing the guest OS and language runtime.

Hence, CSPs face a difficult trade-off between memory and latency: ❶ Retain idle VMs to quickly serve function

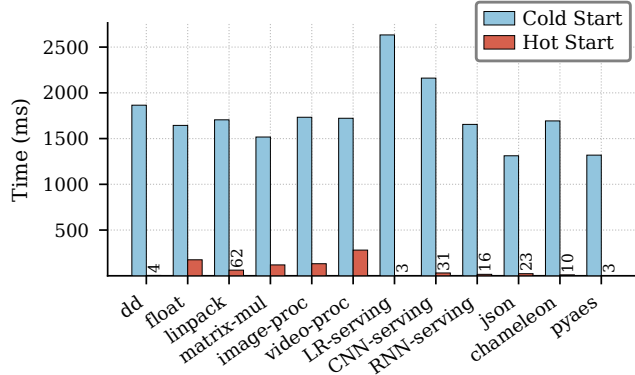


Figure 1. Hot and cold start latencies for FaaS functions from FunctionBench [23].

execution requests. But as idle VMs consume memory, this either requires provisioning additional memory or limiting the number of active VMs deployed on the system. ② Terminate idle VMs and reclaim their memory to accommodate more active VMs of other users. As FaaS functions are stateless, with no execution state retained within a VM, CSPs can safely terminate idle VMs to reclaim memory [34]. But doing so incurs high cold start latency later when a new function execution request is received for the user whose idle VM was terminated.

Takeaway: A practical solution must reduce the memory footprint of idle VMs while keeping the resulting warm-start latency within acceptable bounds.

4 Design

To address the trade-off between memory usage and cold-start latency, we propose to compress and place the memory pages associated with the idle VMs in a compressed memory pool such as zswap in the Linux kernel. When the memory pages are swapped out, their contents are compressed and stored within the compressed memory pool, which reduces the DRAM consumed by idle VMs. The reduction in DRAM depends on the compression algorithm used as well as the compressibility of the VM’s pages.

Upon a new function execution request, the CSP can assign the request to one of the compressed idle VMs. Assigning a function execution to such a VM requires decompressing memory pages from zswap and placing them in DRAM. This triggers a series of on-demand page faults as the VM transitions from the idle state to the active state. Unfortunately, these page faults are costly as they incur high decompression overheads. In addition, each page fault requires a context switch from guest to host because zswap is managed in the host, adding the cost of VM exits to the execution overheads. Concretely, each faulting access traps to the host kernel and incurs a guest-to-host VM exit; the kernel must then allocate

a free page, locate the corresponding entry in the compressed pool, decompress its contents, update the page table, and resume the guest. Because faults are served strictly on demand, these costs are paid serially for every page on the warm-start path, and none of the accelerator’s parallel decompression capacity is exploited.

Although the function execution overheads incurred by placing idle VMs in zswap are considerably lower than those of a cold start, they are still higher than those of a hot start. We thus refer to such execution requests as *warm starts*.

One approach to improve warm start latency is to employ the state-of-the-art readahead [20] technique available in zswap. Readahead reduces page fault latency by decompressing adjacent pages from the faulting address. For example, upon a page fault on page X in zswap, readahead decompresses pages $X + 1, X + 2, \dots, X + n$. But readahead is not useful for applications running inside a virtualized environment when the compressed memory pool is managed in the host because memory accesses that are sequential inside the VM need not be sequential in the host context due to two levels of address indirection (sequential accesses in the VM’s virtual address space need not result in sequential accesses in the host’s virtual address space). In addition, not all FaaS functions exhibit sequential access patterns.

Hence, we further propose optimizations to reduce the latency gap between warm start and hot start.

4.1 Memory Restoration Templates

We propose *MRT* to optimize warm start latency. Instead of handling page faults one by one on demand, MRT restores multiple memory pages of the VM from the compressed memory pool to DRAM at once, eliminating the overheads of individual page faults.

MRT is based on our observation that the sequence of virtual addresses that incur page faults during warm start is similar across multiple warm start instances of the same VM. Based on this observation, we propose to create *memory restoration templates* by profiling the warm start of a VM instance and recording the sequence of faults. We then use the collected templates to bulk-restore the sequence of pages from zswap to DRAM.

It is important to note that MRT does not restore all the memory pages associated with a VM, but merely restores a subset of the memory pages that are actually required for function execution. In particular, memory pages that are not in the critical function execution path such as pages that are never used or that are used for infrequent (housekeeping) tasks are not restored.

Furthermore, CSPs run many VM instances of the same user with identical configuration to serve parallel requests. A *template* recorded on any one such instance can therefore be reused to restore the pages of all other instances of that user on the same system.

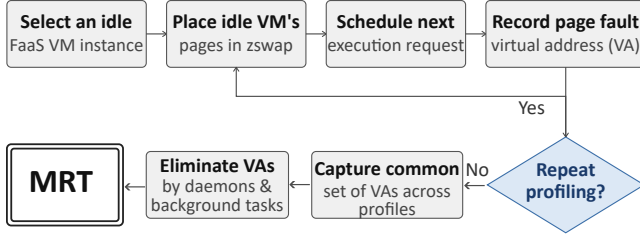


Figure 2. Steps involved in template generation.

4.2 Memory restoration template generation

We now describe in detail the steps involved in template generation. On each host, a template is generated for each combination of user, function, runtime, and VM configuration.

The first step is to pick a target idle VM instance with its associated pages placed in zswap in compressed form (Figure 2). The next step is to schedule a function execution instance on this idle VM. Upon invocation of the function, the VM starts accessing memory pages, resulting in on-demand page faults in the host or the hypervisor.

The host monitors the sequence of page faults and captures the faulting virtual addresses in a template. Tools such as `perf` [1] or `ftrace` [33] available in Linux can be used for monitoring page faults. The page faults include pages belonging to the guest VM’s physical memory, which are memory-mapped into the virtual machine monitor (VMM) process (e.g., QEMU or Firecracker), as well as the heap and stack pages of the VMM process, which hold the vCPU and device state the VMM maintains to enable virtualization.

Although a single profiling run already yields a viable template, template quality improves when the faulting virtual addresses are profiled across multiple warm starts. The *template* then comprises the virtual addresses that recur across the profiles and are therefore likely to fault in future warm starts of that VM (Figure 2).

Once multiple profiling iterations are performed, templates are filtered by eliminating the virtual addresses that occur infrequently. For example, with a 90% threshold, virtual addresses observed in fewer than 90% of the profiled warm starts are eliminated. Such eliminated pages, if required, are decompressed on demand when they are actually accessed by the function. This not only reduces the number of pages that are to be restored, but also improves memory TCO by deferring and retaining low-confidence pages in a compressed pool for a longer time.

Template-based restoration is a performance optimization and does not affect correctness. A page that is restored but never accessed (a false positive) merely consumes decompression bandwidth and DRAM capacity until it is reclaimed again, whereas a page that is accessed but absent from the template (a false negative) falls back to the regular

on-demand fault path. The filtering threshold therefore tunes the balance between restoration coverage and wasted work.

Furthermore, it is also possible that on-demand page faults in zswap are triggered by arbitrary housekeeping/background activities inside the VM. For example, faults may result from pages accessed by daemons. Such pages are not in the critical path of FaaS function execution and need not be included in the template. Identifying such pages typically requires hints or help from the VM. However, our approach of repeating the profiling and retaining pages that frequently trigger page faults across multiple iterations naturally eliminates page faults caused by background activities.

Profiling is lightweight: template generation piggybacks on regular warm starts, with the host only tracing the faulting addresses while the VM serves its requests normally. The number of profiling runs trades template quality against profiling effort; we leave a quantitative study of this trade-off, as well as of the template hit rate (i.e., the fraction of restored pages that are actually used by the function), to future work.

4.3 Triggering restoration

Restoration of the memory pages is triggered just in time, when a function execution request arrives. During a warm start, the host kernel restores the set of pages in the *template* as soon as the idle VM incurs the first page fault on zswap. Hence, the first page accessed by the VM incurs the decompression overheads, while subsequent accesses to template pages that have already been restored do not result in page faults; accesses that occur before restoration completes follow the regular on-demand fault path. To ensure restoration does not interfere with VM execution, pages are bulk-restored asynchronously in the background.

In addition, further optimizations can be employed in production cloud environments where even the first page fault on zswap can be avoided by triggering memory restoration ahead of time. Ahead-of-time restoration is possible by exploiting the workload characterization data available with CSPs [30]. For example, Azure uses time-series analysis to predict VM invocation patterns, and for a significant set of functions, Azure can accurately predict the next invocation time. This prediction infrastructure, already employed in production [30], can be exploited to restore the pages from zswap just before a function execution request is triggered.

4.4 Memory restoration template management

We briefly explain the end-to-end management of memory restoration templates in a typical cloud environment.

The cloud management software monitors the VM instances that are idle for a threshold amount of time and places the pages of the entire VM (which is the VMM process from the host point of view) in a compressed zswap pool. Upon a FaaS function invocation request on an idle VM placed in zswap, the cloud management software can

monitor and record the page faults in zswap to generate the template.

The generated *template*, which consists of an ordered sequence of page virtual addresses, is stored in a per-VM buffer by the cloud management software. For addresses that pass the frequency threshold, the template preserves their fault order from the first profiling run.

A template stores one 8-byte virtual address per tracked 4 KiB page; its size is thus bounded by $\sim 0.2\%$ of the memory it restores, and the per-VM buffers add negligible memory overhead. A template is retained as long as the function, runtime, and VM configuration it was recorded for remain unchanged, and is discarded and regenerated otherwise. Since the recorded virtual addresses are specific to the VM instances of a given user and configuration on a host (assuming disabled ASLR to ensure stable virtual-address mappings across instances), templates are not directly portable across machines: when a function is scheduled on a different host, a template is regenerated there by profiling the warm starts of a local VM instance, as described above.

As restoration of pages from the compressed pool is performed by the host OS kernel, the *template* captured in user space must be made available to it; in Linux, we define interfaces under the `/proc/<vm_pid>/` directory for uploading templates. The kernel stores these virtual addresses in a per-process buffer (Figure 3).

A control file in the same directory lets the cloud management software instruct the kernel when to perform restoration; additional interfaces allow resetting the template and enabling or disabling restoration.

The host kernel restores the pages from zswap as instructed by the cloud management software and updates the page table entries accordingly. The pages that are restored include the guest kernel and guest user space pages as well as the heap and stack pages of the VMM process. The guest kernel pages include guest kernel data structures and the guest user space pages include pages belonging to runtimes (e.g., Python, Java, and Go) that are used for FaaS function execution.

During a function execution, when a VM accesses a virtual address that has already been restored, the access proceeds normally; on a TLB miss, hardware performs a page-table walk without triggering a swap-in fault or VM exit.

5 Evaluation

We evaluate MRT on a dual-socket Intel Xeon Platinum 8490H with 60 cores per socket, 250 GB DRAM, hyperthreading disabled, and IAA [21, 29] for hardware (de)compression. We use Linux kernel version 5.17, which was the most recent mainline kernel with IAA-enabled zswap support at the time of prototyping. MRT does not rely on any kernel-version-specific features and is portable to newer kernels.

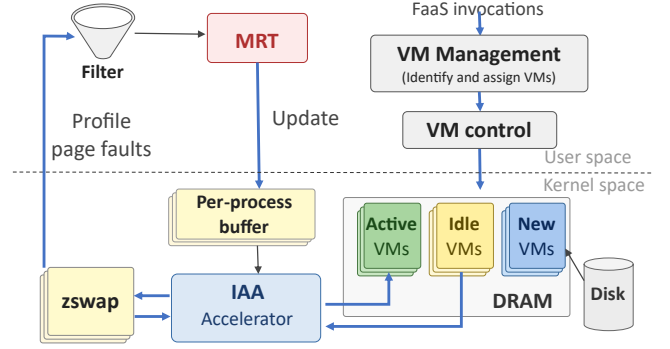


Figure 3. High-level design of MRT.

Setup and configurations. We use an industry-standard and representative open-source serverless software stack for our setup. We use Firecracker [2, 11] as our VM for function execution and use Kata [6] as a secure container runtime. Containers are managed using containerd [15]. For orchestration, we use Kubernetes [9], with Knative [7] as the serverless application platform. We use Kourier [8] for front-end request handling and gRPC [3] as the high-performance communication framework.

We configure a single-node Kubernetes cluster as our testbed. To test autoscaling of multiple function instances, we use the k6 load generator [5]. We configure Prometheus [10] for data collection to record function execution latencies. Each experiment is repeated multiple times, and we report the average across repetitions.

We use the deflate [19] compression algorithm with IAA. zswap is configured to use zsmalloc as the pool allocator. We evaluate two IAA configurations. The first is the default mode in Intel’s IAA crypto module in the Linux kernel, where compressed pages are decompressed by the IAA engine one at a time on demand during a page fault. In the second, we employ batching, which initiates parallel decompression to utilize multiple engines in the IAA device. We use swap-in readahead with a batch size of 8 (referred to as IAA-b8) to decompress eight consecutive pages from the faulting address, which is similar to sequential prefetching. We use FunctionBench [23], a benchmark suite for serverless cloud function services, for our evaluation.

Scope of the evaluation. Our evaluation covers FunctionBench workloads, which are representative of common FaaS kernels but simpler than production applications with large dependency trees; evaluating MRT on complex, end-to-end serverless applications is part of our future work. Similarly, comparison with disk-based snapshot restoration [24, 34] is also left for future work.

5.1 Results

5.1.1 Execution latency improvements. Figure 4a shows the function execution latency normalized to hot start for

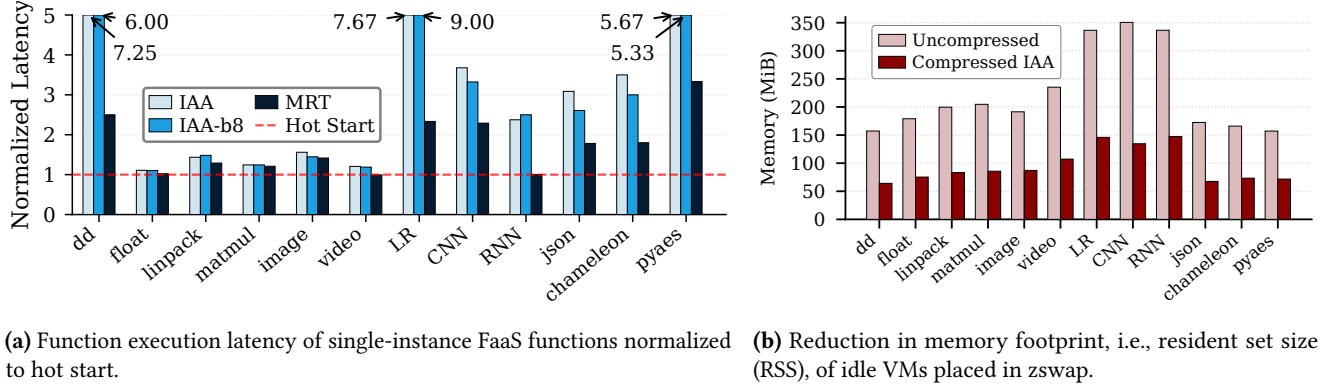


Figure 4. Latency improvements with MRT and the memory savings due to compression.

single-instance FaaS functions. Using compression accelerators alone still incurs significantly higher warm-start latencies (gmean $2.63\times$ the hot-start latency), as compressed pages are decompressed one at a time when a page fault is triggered; individual functions suffer slowdowns of up to $7.7\times$ (LR-serving). Also, batching (IAA-b8) is ineffective in reducing warm-start latencies, as most FaaS functions do not exhibit sequential access patterns, resulting in $2.51\times$ the hot-start latency. However, the proposed method, MRT, which uses templates to decompress the pages, significantly reduces warm-start latency to $1.62\times$ the hot-start latency.

The gains from MRT vary across functions and are governed by the ratio of page-restoration overhead to the function’s compute time. Functions with millisecond-scale hot-start latencies (dd, LR-Serving, pyaes, chameleon, and RNN-Serving, all at or below 16 ms) spend most of their warm start serving serialized on-demand faults, so eliminating these faults yields the largest normalized improvements; for example, LR-Serving improves from $7.7\times$ to $2.3\times$ and dd from $7.3\times$ to $2.5\times$ over hot start, while RNN-Serving matches its hot-start latency under MRT. In contrast, compute-dominated functions such as float, matrix-mul, and video-proc (hot-start latencies of 119–281 ms) amortize the restoration overhead over long executions; their warm-start penalty is small to begin with ($1.11\text{--}1.24\times$ with on-demand IAA faults), leaving MRT less room for improvement, although it still brings float and video-proc to within 3% of hot-start latency.

5.1.2 Memory savings. The reduction in the memory footprint when pages of idle VMs are placed in zswap depends on the compression algorithm employed in the accelerator (IAA uses deflate compression by default [29]). Figure 4b shows the reduction in the memory footprint of idle VMs as a result of placing them in a compressed memory pool. The memory footprint is reduced by 57.3% (gmean) across function instances, thus allowing CSPs to deploy more active VM instances on their systems. In absolute terms, the resident set size of a single idle VM instance drops from 157–351 MB to 64–147 MB, a $2.2\text{--}2.6\times$ reduction per function. The savings

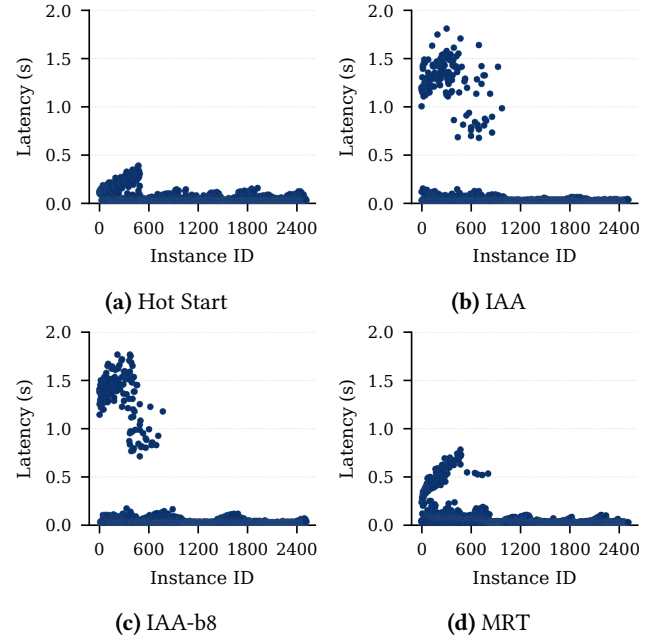


Figure 5. Hot start and warm start (IAA, IAA-b8, and MRT) latencies at scale.

are consistent across all twelve functions, indicating that the memory of idle FaaS VMs, which is dominated by guest OS and language runtime state, compresses well under deflate.

5.2 At-Scale Results

We evaluate MRT at scale using k6 on a client system to issue parallel CNN-serving requests from 150 virtual users to our single-node Kubernetes cluster.

We first issue enough requests for Kubernetes to spawn the required VMs; because no active instances exist initially, these requests trigger cold starts. We then pause the load for 120 seconds to let the VMs become idle, place their pages in zswap, and resume the load. Kubernetes reuses these idle

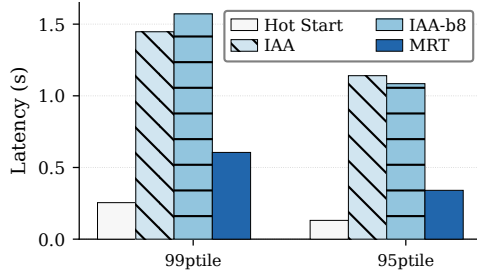


Figure 6. 95th- and 99th-percentile function execution latency at scale.

VMs instead of booting new ones, thereby triggering warm starts.

For the hot-start baseline, we repeat the same pause-and-resume procedure but retain the idle VMs’ pages in DRAM.

Figure 5 shows that the first requests incur warm-start latency under IAA, IAA-b8, and MRT. Once all idle VMs have been restored, all three configurations match hot-start latency; MRT substantially reduces the initial penalty relative to IAA and IAA-b8.

Figure 6 reports tail latency for all four configurations. MRT reduces 95th- and 99th-percentile latency by 70.12% and 58.17% relative to IAA, and by 68.59% and 61.49% relative to IAA-b8, respectively. Median latencies remain similar (42–46 ms), confirming that warm-start overhead is concentrated in tail requests that reach still-compressed VMs and is substantially reduced by MRT.

6 Related Work

Working-set record-and-prefetch. REAP [34] records pages faulted during the first invocation of a snapshot-forked VM and bulk-loads this working set from disk on subsequent invocations. Pages outside its single-pass working set fault on demand. FaaSnap [12] addresses the input dependence and overhead of this approach by tracking host-cache-resident pages instead of only faulting guest pages.

MRT shares the record-and-prefetch intuition but bulk-decompresses pages of a live, initialized VM during its idle-to-active transition. Unlike REAP, it avoids disk access and runtime reinitialization while preserving runtime state. Like FaaSnap, MRT accounts for input variability during recording by retaining faulting addresses that recur across multiple warm starts and deferring the remainder to on-demand decompression.

Hardware-accelerated (de)compression. Sabre [24] uses Intel IAA to compress snapshot or working-set pages and hides decompression behind disk I/O. It accelerates restoration but retains the snapshot file and cold-fork reconstruction; MRT instead restores a live VM directly from an in-memory compressed pool.

Snapshotting. Disk-based snapshotting [17, 18] checkpoints a booted VM and clones instances on demand; earlier systems use VM cloning and lazy guest-memory loading [14, 28]. These approaches reclaim idle memory but incur cold-fork reconstruction and may not preserve network or socket state; clones also begin with identical entropy. MRT instead retains the live VM in a compressed pool, avoiding these limitations at the cost of decompression. Finally, as discussed in § 4, swap readahead [20] restores pages spatially adjacent to the faulting address and is ineffective for virtualized FaaS workloads; MRT instead restores a recorded, profile-derived set of pages.

7 Future Work

MRT opens up several promising directions for further exploration. A central question is *when* to compress: delaying preserves warm-start latency but limits memory savings, whereas aggressive compression increases overhead. Per-function invocation patterns [30] could inform adaptive policies that navigate this trade-off. On the hardware side, faster accelerators could further reduce restoration overhead, and placing compressed pools in byte-addressable far-memory tiers such as CXL could improve VM density. Finer-grained characterization, separating decompression cost from fault-handling (VM-exit) overhead and quantifying template hit rate, unused restored pages, and profiling effort, may reveal further opportunities. How minimal templates can be constructed, how they should adapt to drifting function behavior, and how these effects manifest at scale are promising avenues for future work.

8 Conclusion

We argue that placing idle-VM pages in a compressed memory pool, rather than snapshotting them to disk, lets CSPs reclaim memory while keeping VMs live and avoiding snapshot reconstruction and state-preservation costs. MRT reduces the cost of restoring a compressed VM by recording the pages that fault during a warm start and restoring them in a single bulk, accelerator-driven operation, eliminating per-fault guest-to-host transitions and exploiting parallel decompression engines. Together, the compressed memory pool and MRT let CSPs reclaim idle-VM memory while keeping warm-start latency close to that of a memory-resident instance.

Acknowledgments

We thank the anonymous DIMES ’26 reviewers for their constructive feedback. We also thank Anubhav Jana and Mohak Chadda, who were previously associated with this project, for their contributions.

References

- [1] 2023. PerfMon Events. <https://perfmon-events.intel.com/>. Accessed: 2023-08-01.
- [2] 2026. Firecracker: Secure and fast microVMs for serverless computing. <https://firecracker-microvm.github.io/>. Accessed: 2026-08-28.
- [3] 2026. gRPC: A high performance, open source universal RPC framework. <https://grpc.io/>. Accessed: 2026-08-28.
- [4] 2026. gVisor. <https://gvisor.dev/>. Accessed: 2026-08-28.
- [5] 2026. k6: The best developer experience for load testing. <https://k6.io/>. Accessed: 2026-08-28.
- [6] 2026. Kata Containers - Open Source Container Runtime Software. <https://katacontainers.io/>. Accessed: 2026-08-28.
- [7] 2026. Knative: An Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. <https://knative.dev/>. Accessed: 2026-08-28.
- [8] 2026. Kourier. <https://github.com/knative-extensions/net-kourier>. Accessed: 2026-08-28.
- [9] 2026. Kubernetes. <https://kubernetes.io/>. Accessed: 2026-08-28.
- [10] 2026. Prometheus: From metrics to insight. <https://prometheus.io/>. Accessed: 2026-08-28.
- [11] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.
- [12] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaS made fast using snapshot-based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems (Rennes, France) (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 730–746. doi:10.1145/3492321.3524270
- [13] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From warm to hot starts: Leveraging runtimes for the serverless era. In *Proceedings of the workshop on hot topics in operating systems*. 58–64.
- [14] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 273–286.
- [15] Cloud Native Computing Foundation. 2026. containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability. <https://containerd.io/>. Accessed: 2026-06-06.
- [16] Jonathan Corbet. 2012. The zsmalloc allocator [LWN.net]. <https://lwn.net/Articles/477067/>. (Accessed on 08/04/2023).
- [17] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.
- [18] fc-snapshot 2026. Firecracker Snapshotting. <https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md>. Accessed: 2026-08-28.
- [19] Danny Harnik, Ety Khaitzin, Dmitry Sotnikov, and Shai Taharlev. 2014. A Fast Implementation of Deflate. In *2014 Data Compression Conference*. 223–232. doi:10.1109/DCC.2014.66
- [20] Ying Huang. 2017. VMA based swap readahead. <https://lwn.net/Articles/716296/>.
- [21] iaa 2026. Intel In-Memory Analytics Accelerator (Intel IAA). <https://www.intel.com/content/www/us/en/content-details/787805/enhance-business-with-faster-insights-with-intel-iaa.html>. Accessed: 2026-08-28.
- [22] Seth Jennings. 2013. The zswap compressed swap cache [LWN.net]. <https://lwn.net/Articles/537422/>. (Accessed on 08/04/2023).
- [23] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 502–504.
- [24] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. 2024. Sabre: Hardware-Accelerated Snapshot Compression for Serverless MicroVMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 1–18. <https://www.usenix.org/conference/osdi24/presentation/lazarev>
- [25] lz4 [n. d.]. lz4/lz4: Extremely Fast Compression algorithm. <https://github.com/lz4/lz4>. (Accessed on 08/04/2023).
- [26] lzo [n. d.]. Lempel–Ziv–Oberhumer - Wikipedia. <https://en.wikipedia.org/wiki/Lempel-Ziv-Oberhumer>.
- [27] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.
- [28] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. 2005. Fast Transparent Migration for Virtual Machines.. In *USENIX Annual technical conference, general track*. 391–394.
- [29] Binuraj Ravindran, Bala Seshasayee, Kanchana Sridhar, Rakib Al-Fahad, Vinodh Gopal, G Pallavi, and Mukhtadir Chowdhury. 2023. Memory Tiering with Intel® In-Memory Analytics Accelerator (Intel® IAA). (2023).
- [30] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 205–218.
- [31] SRC-study [n. d.]. Decadal Plan for Semiconductors. <https://www.src.org/about/decadal-plan/decadal-plan-full-report.pdf>.
- [32] Kanchana P. Sridhar. 2025. [PATCH v6 00/16] zswap IAA compress batching. Linux Kernel Mailing List (LKML). Available at <https://lkml.org/lkml/2025/2/5/1790>. <https://lkml.org/lkml/2025/2/5/1790>
- [33] Rostedt, Steven. 2008. ftrace - Function Tracer. <https://static.lwn.net/kernel/doc/trace/ftrace.html>.
- [34] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*. 559–572.
- [35] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. doi:10.1145/3503222.3507731
- [36] zbud-zswap [n. d.]. zswap — The Linux Kernel documentation. <https://www.kernel.org/doc/html/v5.8/vm/zswap.html?highlight=zbud>. (Accessed on 08/04/2023).
- [37] zswap-kernel [n. d.]. zswap — The Linux Kernel documentation. <https://www.kernel.org/doc/html/latest/admin-guide/mm/zswap.html>. (Accessed on 07/22/2023).