Sabre: Hardware-Accelerated Snapshot Compression for Serverless MicroVMs

Nikita Lazarev Varun Gohil James Tsai[†] Andy Anderson[†] Bhushan Chitlur[†]
Zhiru Zhang[§] Christina Delimitrou
MIT, CSAIL [†]*Intel Labs* [§]*Cornell University*

Abstract

MicroVM snapshotting significantly reduces the cold start overheads in serverless applications. Snapshotting enables storing part of the physical memory of a microVM guest into a file, and later restoring from it to avoid long cold start-up times. Prefetching memory pages from snapshots can further improve the effectiveness of snapshotting. However, the efficacy of prefetching depends on the size of the memory that needs to be restored. Lossless page compression is therefore a great way to improve the coverage of the memory footprint that snapshotting with prefetching achieves. Unfortunately, the high overhead and high CPU cost of software-based (de)compression make this impractical.

We introduce Sabre, a novel approach to snapshot page prefetching based on hardware-accelerated (de)compression. Sabre leverages an increasingly pervasive near-memory analytics accelerator available in modern datacenter processors. We show that by appropriately leveraging such accelerators, microVM snapshots of serverless applications can be compressed up to a factor of 4.5×, with nearly negligible decompression costs. We use this insight to build an efficient page prefetching library capable of speeding up memory restoration from snapshots by up to 55%. We integrate the library with the production-grade Firecracker microVMs and evaluate its end-to-end performance on a wide set of serverless applications.

1 Introduction

Serverless is an emerging cloud computing paradigm gaining widespread popularity across applications of different classes, from lightweight interactive services [73] to highly data-parallel applications, such as machine learning and video encoding [21,24,40,47,67]. Serverless offers a Function-as-a-Service (FaaS) execution model, where applications instantiate short-lived, fine-grained resources on-demand without the overhead of provisioning and deployment typical cloud environments incur. When requests are processed, the resources are terminated, achieving a pay-as-you-go model. This both avoids resource overprovisioning, which has been a longstanding issue with cloud infrastructures [22,28,58] and reduces the end-to-end deployment cost [33,46].

Serverless is based on lightweight virtualization/isolation technologies [19] such as Docker, Google gVisor [3], Kata containers [10], NEC's LightVMs [53], and AWS Firecracker microVMs [17]. These technologies implement sandboxes for executing containerized applications with different levels of isolation. For example, gVisor implements a lightweight user space kernel capable of executing most of the system calls within the sandbox. On the other hand, Firecracker is a full lightweight virtualization technology, based on KVM, which can boot standard Linux kernels in sub-second time [17]. Due to the high isolation guarantees of microVMs, security, and performance, Firecracker is widely used in serverless clouds.

Despite its advantages, serverless and microVMs introduce a few critical overheads to performance. A major overhead is cold starts (or cold boots) – the overhead of the initial boot of container sandboxes upon a function invocation. Both industry and academia have proposed numerous techniques to mitigate cold start overheads [31,35,64,73], with one of the most promising being VM snapshotting [31]. VM snapshots capture the current state of the VM and its physical memory and save them in a file. During the next boot, the guest system is restored from the file instead of booting from scratch. Several different techniques can be used to make snapshots, depending on which parts of the guest's physical memory should be saved. For instance, Firecracker can snapshot the full guest memory or only the dirty pages. Additionally, recent studies have proposed using *working sets* of pages [71] to make serverless VM snapshots smaller, faster to fetch, and overall more efficient [20,64] in reducing cold starts. However, independent of the underlying techniques used to create snapshots, the overhead of storing and prefetching them is non-negligible. Unfortunately, the latter is on the critical path of VM restoration and therefore directly impacts cold starts.

Reducing the size of snapshots can make them significantly more efficient, for example through lossless memory compression. While memory compression has been used in domains where the application performance is not critical (e.g., *zswap* [39], *zram* [38] in Android OS for mobile devices), in serverless the restoration of a memory snapshot is on the critical path, precluding the use of existing software-based (de)compression algorithms. At the same time, there are numerous hardware implementations of (de)compression [23, 41, 51, 52, 56], however, they had not, until now, been implemented in mainstream datacenter processors [43]. In particular, Intel recently released the In-Memory Analytics Accelerator (IAA) [6] in their 4^{th} Gen Xeon Scalable CPUs, which enables efficient compression for datacenter applications at scale.

We present Sabre, a hardware-accelerated *general-purpose* memory prefetching system, which uses lossless compression mechanisms, such as IAA, to compress and restore microVM snapshots. This paper makes two major contributions. First, we characterize, for the first time, the IAA accelerator on a set of diverse benchmarks, and show its potential for compressing memory pages. We show that IAA can compress pages by $2-4.5\times$, depending on the underlying page selection algorithm. Moreover, we show that decompression can be done up to 10× faster with hardware acceleration, and with careful design, this time can be entirely hidden behind the disk I/O and page fault handling. This results in near-free decompression in terms of the overall memory restoration latency, while reducing the size and the loading time of snapshot pages.

Second, based on this characterization, we build Sabre and integrate it with the Firecracker virtual machine monitor (VMM) in a serverless environment with snapshotting support. Sabre is agnostic to the underlying page snapshotting policy, it operates entirely in the host's user space, and interacts with the IAA accelerator via the Shared Virtual Memory (SVM) mechanism. The latter enables out-of-box and transparent integration of Sabre with existing VMMs at scale.

We evaluate Sabre on its efficiency in restoring microVMs from snapshots across a wide range of end-to-end serverless applications using two methods of creating snapshots: dirty page-based and working set-based. We show that Sabre compresses microVM snapshots up to 4.5× without introducing any decompression overheads. Moreover, we show that Sabre enables up to 55% faster memory restoration, which results in an additional reduction of the end-to-end cold start time by 20% with respect to already optimized state-of-the-art snapshotting baselines.

Sabre is open-source software and it is available at the following link [13].

2 Background

2.1 MicroVMs for Serverless

Serverless is gaining popularity across many application domains by reducing the cloud provisioning overhead and enabling higher elasticity for applications with high parallelism and intermittent activity. Lightweight virtualization technologies (or microVMs) became a popular choice for cloud providers due to the isolation and fast instantiation they provide [17, 69]. Fully virtualized VMs running over Type-1 hypervisors, such as KVM [37] or Hyper-V [70], allow isolating tenants down to hardware and provide the highest security guarantees for applications running in the cloud. On the other hand, microVMs are much faster to boot and have a much smaller memory footprint than traditional Type-1 virtual machines. This means that microVMs achieve the best of both worlds between containers and Type-1 hypervisors. MicroVMs are widely used in serverless, where applications require both strong isolation guarantees and fast start-up.

Modern microVMs, such as AWS Firecracker [17], use several optimizations to boot up in sub-seconds. However, booting the VM itself is only part of the end-to-end application execution latency [31, 64], with a significant component corresponding to the initialization of the software dependencies after the boot. For applications based on complex multi-layer stacks, such as gRPC servers and JavaScript runtimes, bringing up the dependencies might be as high as several seconds [31]. Additionally, the applications themselves can contain long-running initialization routines, which also contribute to end-to-end latency. For example, machine learning (ML) services need to load the models before serving inference queries. Altogether, this makes the end-to-end execution of the first batch of requests running on freshly booted microVMs an order of magnitude slower than subsequent requests. This is known as *cold start*, and all microVMs are prone to it.

Mitigating cold starts is one of the most well-researched aspects of microVMs [65]. Existing solutions range from scheduling techniques optimized for specific applications to runtime and infrastructure optimizations [54, 73]. One solution that is generally agnostic to applications is VM snapshotting [31].

2.2 MicroVM Snapshotting and Prefetching

Snapshotting is a technology that allows storing the VM state and guest OS physical memory in a file in the local or remote filesystem. Snapshots are usually created after the VM and the application logic with all its dependencies are fully initialized and ready to serve requests. Upon the next invocation of the VM the hypervisor restores the VM state and guest memory from the snapshot, instead of booting the VM from scratch. This dramatically reduces cold start overheads.

In the most basic case, snapshots contain the entire guest physical memory. Some hypervisors, such as Firecracker, also allow dirty-memory tracking, which only stores the dirty guest pages as seen by the hypervisor. Snapshots can be organized hierarchically following the software dependencies of applications [31]. However, recovering from snapshots is far from free. In some cases, the size of the snapshots can be as high as the whole guest memory, therefore precluding the possibility of loading pages from snapshots in advance. For this reason, existing commercial microVMs implement memory restoration via *on-demand* paging. Unfortunately, on-demand paging yields a lot of page faults on the critical path of the restoration from snapshots, which slows down request execution.

A way to reduce the overhead of page faults is to enable *prefetching* of pages from snapshots. This can be efficiently done through, for example, *working set* (WS) estimation. This approach has been used to create VM checkpoints [71], and recently – for serverless microVMs [64]. Here, each snapshot is accommodated in a WS file, storing pages that are likely to be accessed during subsequent invocations. There are different ways of constructing WS files [20, 64, 71] according to various working set estimation techniques. For example, in Record-and-Replay (REAP) [64], the authors propose to record all guest pages being accessed during the first invocation of serverless functions and put them into the WS file. Upon the next invocation, the WS file can be prefetched from the disk, and the WS pages can be installed in the guest's memory to speed up the next cold invocations. REAP works well for applications with a similar working set across different invocations of the same function. When this does not hold, REAP can fail to deliver good performance; in this case, prefetching some other subset of dirty pages (or even all dirty pages) can be more beneficial. Snapshots generally consume a lot of disk space and require cloud providers to carefully provision their storage resources [2]. Even working-set-based snapshots can sometimes be as large as a few hundred megabytes [20]. This is non-negligible given that a single server might host hundreds of microVMs.

Independently of the underlying technique to create microVM snapshots and/or WS files, the efficiency of prefetching depends on the memory size that needs to be restored from the disk into the guest memory. A general rule to make prefetching-based techniques more efficient is to reduce the size of the snapshots/WS files. This also reduces the disk space needed to store snapshots. This size reduction can be achieved through memory compression. However, memory restoration happens on the critical path of the VM boot-up, and for compression algorithms with high deflate ratios, the decompression might take a long time. Moreover, such algorithms usually consume a lot of CPU time for compression and therefore VM snapshotting. This makes the use of software memory compression for microVM snapshotting undesirable.

2.3 Hardware-Accelerated (De)Compression

Software-based memory compression has been extensively used in applications where performance is not critical. For example, *zram* [38] is used in Android OS on mobile devices, while *zswap* [39] can improve the efficiency of memory swapping for non-performance critical applications. Unfortunately, this does not apply to microVM memory restoration, where decompression directly impacts the cold start overhead.

There have been many proposals for accelerating memory compression in hardware. For example, Pekhimenko et. al. [56] propose base-delta compression for on-chip caches. Hoyong et. al. [41] derive a novel compression algorithm for GPU memory. Li et al. [52] introduced a hardware accelerator for the compression of genome sequences. All such proposals are based on application-specific, special-purpose compression accelerators and algorithms, and therefore have never been implemented on commodity datacenter processors. Many (de)compression accelerators are based on FPGA cards [25, 34, 50, 57] which are only available in a small set of public clouds. However, the demand for *general-purpose* (de)compression acceleration *at scale* is actively growing.

(De)Compression is known to be one of the major sources of datacenter tax [36,42,61]. A recent study from Google [36] showed that compression accounts for up to 30% of cycles for large-scale database applications, such as BigTable and BigQuery [32,62], and it is also extensively used in many other applications. This motivates cloud providers and chip vendors to build efficient hardware accelerators [43] for *general-purpose* lossless (de)compression. The primary use cases for such accelerators are databases and query-processing engines. In particular, Intel recently introduced the In-Memory Analytic Accelerator (IAA), which is now part of commodity datacenter processors, such as the Xeon 4^{th} Generation CPUs, which are already widely available. This accelerator can perform DEFLATE compression, which is suitable for compressing memory footprints. While other compression algorithms that are optimized for performance (e.g., *Snappy*, *zstd*, *LZ4*) can compress memory faster, they typically result in much lower compression ratios, which is critical for microVM snapshotting. Their software implementations increase the amount of CPU resources required for making snapshots, especially when configured for more aggressive compression [1]. At the same time, hardware-accelerated DEFLATE yields high compression ratios as well as high speed, while requiring no CPU cycles for (de)compression. It should also be noted that hardware accelerators for other compression algorithms are also feasible [16, 26, 60] but not yet implemented in mainstream datacenter processors at scale.

IAA and other similar accelerators are designed for cloud environments. They are typically implemented as on-chip near-memory PCIe components, which allows them to be easily integrated with cloud services. For instance, IAA can run entirely in user space, it operates transparently over the application's virtual memory and can be virtualized through standard technologies, such as S-IOV [8]. All this makes IAA attractive for microVM memory snapshotting and prefetching. In this work, we explore this direction.

We first characterize the capabilities of IAA when it comes to compressing memory pages. Based on this characterization, we design Sabre, a memory prefetching system for microVMs built using IAA. Finally, we show how our memory prefetching unit integrates with serverless microVMs and evaluate its impact on end-to-end serverless benchmarks.

3 In-Memory Analytic Accelerator: Overview, Characterization, Insights

We now present an overview and characterization of the Intel In-Memory Analytic Accelerator (IAA) [6] using a set of diverse benchmarks [15]. This work mainly focuses on the compression/decompression capabilities of the accelerator. However, given that other capabilities share the same IAA frontend pipeline, interfaces, and software semantics, most of our findings also apply to these other domains. To our knowledge, this is the first publicly available characterization of the IAA hardware. The insights from this characterization are used to derive the design of Sabre, described in Section 4.

Note that given the diverse set of execution models, configurations, workloads, and variations of IAA hardware in different SKUs, it may be possible to achieve even higher performance compared to what we showcase in our characterization. Specifically, benchmarks designed specifically to stress test the accelerator may be able to improve IAA's performance further. For the purpose of this paper, we only benchmark the accelerator with a set of scenarios required to give comprehensive insights into using in-memory compression techniques in serverlerss microVMs and to derive the design of Sabre.

3.1 Overview of the IAA

Intel's IAA is a hardware accelerator first introduced in Intel's 4^{th} Gen Xeon Scalable Processors (code-named Sapphire Rapids) [12] to speed up data processing across application classes. It was designed with the primary use case being databases and query processing systems [7]. The accelerator physically resides in the uncore part of the processor's SoC near the memory controller and Last Level Cache. A single CPU can accommodate multiple IAA devices on its SoC.

The IAA accelerators are logically integrated as PCIe devices and exposed to the host as a single root complex integrated endpoint. This is set up to enable transparent integration of the accelerators with software. IAA features scalability, full virtualization support via PCIe S-IOV, Shared Virtual Memory support (SVM or SVA as defined by Linux kernel documentation) [11], and transparent user-space interaction with applications via a new ISA extension, called *ENQCMD*.

Communication and job submission to the accelerator are handled via *Work Queues* (WQs), similar to another emerging hardware – Data Streaming Accelerator (DSA) [48]. For this, software needs to create *descriptors* and *completions* allocated anywhere in the application virtual address space. Descriptors contain information describing the jobs assigned to the accelerator's *Processing Units* (PEs), such as the locations of the source and destination buffers, opcodes, and operational and memory policy flags. Descriptors are submitted to the accelerator via the ENQCMD instruction directly from user space, which writes them into the device's memory-mapped I/O (MMIO) registers. Upon receiving descriptors, the PEs fetch data based on the pointers in the descriptors. This is done through SVM which enables transparent sharing of the application's virtual memory with accelerators. When a PE finishes processing, it writes the corresponding completion record with the status information. The software can poll the completion records to identify the termination of tasks and any error information. If some application memory pages associated with data buffers are not available, the accelerator can request them via either *Page Request Service* (PRS) or through userspace page fault handling. In the latter case, software applications can resolve the page faults in a more suitable for a particular usage scenario way (e.g., by requesting pages from, for example, the network) in the user space.

IAA's job submission mechanism enables asynchronous/non-blocking and out-of-order processing of descriptors. The current hardware permits a large number of in-flight requests, which can be submitted from different threads and processes/tenants. The micro-architectural pipeline of the IAA hardware contains multiple PEs that can execute jobs concurrently. The DSA specification [4] and in-depth characterization [48] contain more details since both accelerators share the same specification for this part.

The IAA hardware contains PEs implementing different processing capabilities. These include encryption, compression, CRC offload, data filtering, scanning, extraction, selection, and expansion [5] (Table 3-1). Due to the scope of this work, we only focus on characterizing the (de)compression capability of IAA. IAA performs DEFLATE [30] compression, as defined in RFC 1951. DEFLATE is based on LZ77 matching and Huffman encoding. LZ77 matching eliminates redundancy by replacing repeated occurrences of substrings with references to a single version of the substring. This is a computationally intensive process making software implementations slow. The Huffman coder further deflates data by re-encoding the most common symbols with fewer bits using statistics of data distribution in the input stream. Internally, the IAA compression unit operates in three modes.

In the first – *Huffman-mode*, IAA performs hardware-accelerated LZ77 dictionary coding, using 4 KB windows, and encodes the results with pre-defined static Huffman tables. The second IAA mode – *Statistics-mode* is designed to sample input streams and construct the statistical data distribution to optimize Huffman tables for a particular input. Compression with input-specific Huffman tables usually allows much higher compression ratios. In Statistics-mode, IAA only constructs the histogram of the distribution of Huffman codes, but it does not write the actual Huffman tables yet. The latter

Style	Description
Fixed Block	Standard static DEFLATE; based on Huffman-mode with standard Huffman tables; enables faster compression, but under general Huffman tables, which usually results in low compression ratios.
Static Block	Similar to Fixed Block, but using user-defined Huffman tables; can result in good compression ratios if the application is able to provide Huffman tables fitting all inputs well.
Dynamic Block	Standard dynamic DEFLATE; two-phase compression with Statistics-mode followed by Huffman-mode; enables optimal Huffman tables per block and a better compression ratio, but requires more time to compress.
Canned	Allows sharing the same Huffman tables between multiple blocks of compressed data; this is important when compressing many small scattered chunks to avoid having to keep/access Huffman tables per block.

Table 1: End-to-end compression styles supported in Intel IAA.

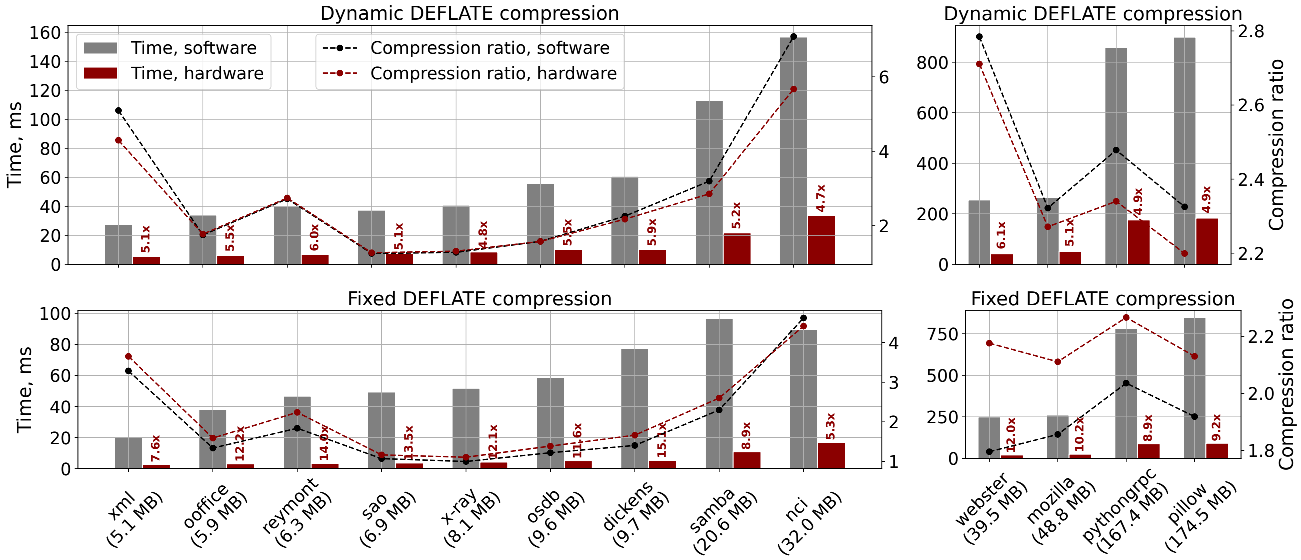

Figure 1: Comparison of software- and hardware-based DEFLATE *compression* for different datasets (sorted by uncompressed size) from Silesia Corpus and serverless VM snapshots (last two); the numbers denote speed-up of the hardware execution; a single IAA device with a single PE (engine) in blocking/synchronous mode is used; the software baseline runs on a single thread.

is done in the third mode – *Huffman-Generation mode*, which is only supported in some IAA implementations.

Based on these modes, IAA defines four main *styles* of compression (Table 1), and it is up to the control plane software to implement them. To make end-to-end (de)compression easier to implement, Intel has recently released the Query Processing Library (QPL) [7], which abstracts away the IAA modes and allows users to express compression in any of the supported modes. We next show the results of microbenchmarking IAA with different modes and implementations.

3.2 Characterizing Compression Using IAA

We characterize IAA (de)compression with a set of benchmarks written using the Intel QPL library v1.3.1. As input data, we use 11 datasets from the standard Silesia Corpus [29]; a common way to evaluate (de)compression. We also add two more datasets specific to our use case representing the dirty memory snapshots of microVMs. The snapshots were obtained during request execution for two serverless applications from vSwarm [66] and FunctionBench [44,45]: a Python gRPC server (*pythongrpc*) and the Pillow image processing library (*pillow*). Both datasets only contain dirty pages of guest memory. Table 2 shows the specification of our testbed. At the time of writing, we had access to two Sapphire Rapids systems (SKUs) with slightly different configurations. We use the most recent production-grade SKU (*Server #2* in Table 2) in all experiments unless otherwise noted.

3.2.1 Benchmarking IAA: Core Compute

We start with the characterization of the in-memory core computing capability of IAA. We assume that data is always available in memory, both for the source and destination buffers. We ensure that memories are initialized and touched to avoid page faults. This is important as page fault handling affects the accelerator's performance, and therefore we evaluate it separately. For all experiments in this section, we use a single

CPU
(Server #1)
	Intel 4^{th} Gen Xeon Scalable Processor;
	2 NUMA nodes, 56 cores/112 threads;
	Core/Uncore frequency (GHz): 1.7/ 1.8;
	LLC capacity (MB): 110
	IAA devices: 8 (4 per NUMA node)

CPU
(Server #2)
	Intel(R) Xeon(R) Gold 6438Y+;
	2 NUMA nodes, 32 cores/64 threads;
	Core/Uncore frequency (GHz): 2.3/ 1.8;
	LLC capacity (MB): 60
	IAA devices: 2 (1 per NUMA node)

IAA	available PEs per device: 8 capabilities (as per *GENCAP* register): - Huffman generation mode: *disabled*; - Page Request Service (PRS): *enabled*; - Block-on-Fault: *enabled*; WQ configuration: shared, 8 per device, size: 32
Memory	Type: DDR5; Capacity (GB): 250
Disk	Intel SSDSC2KG960G8 Sequential O_DIRECT read bandwidth (MB/s): 550

Host OS	Ubuntu 22.04;
	Kernel: 5.15, patched with [9] to enable ENQCMD;
	Kernel boot arguments: $intel\_iommu = on, sm\_on$

Guest OS **(Section 5)**	Rootfs: Debian GNU/Linux 12 (bookworm) Kernel: 4.14.174
IAA stack	Driver: idxd Middleware: Intel QPL v1.3.1

Table 2: Testbed hardware and software configuration.

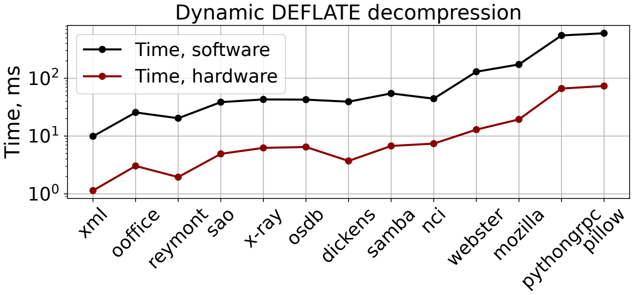

Figure 2: Comparison of software- and hardware-based DEFLATE *decompression* on the same datasets and setup as in Figure 1

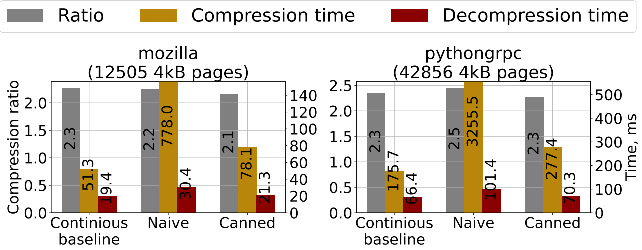

Figure 3: (De)compression of scattered 4kB pages across modes.

IAA device configured with a single PE (engine); we submit the jobs to the accelerator from a single CPU thread in the blocking/synchronous mode and wait till completion by polling associated completion records. The software baseline runs on a single CPU core.

We first compare the performance and compression ratios of IAA-enabled compression and its software implementation in QPL. Since the IAA hardware does not allow explicitly selecting compression levels (compression levels are subjective and vary across implementations), we set the default compression level-1, as defined by QPL, for the corresponding software implementation. Since our version of IAA does not offload Huffman table generation, the hardware implementation of dynamic DEFLATE compression is actually hybrid: statistics collection, LZ77 encoding, and compressed stream generation run in hardware, while Huffman table generation runs in software. The hybrid operations run on a single CPU core. Figure 1 shows the compression results.

The hardware implementation always overperforms software in compression time. The difference reaches 6.1× and 13.5× for dynamic and fixed compression, respectively. In our datasets of dirty memory snapshots, the speedup reaches 9×. The achieved compression ratios for software and hardware executions are similar. Figure 2 shows the performance of decompression. We only show the decompression of *dynamic* streams, as in these datasets, decompression performance does not depend on the compression mode. In all cases, IAA decompresses an order of magnitude faster than software.

Note that in this paper, our experiments only compare IAA to the software implementation of the same DEFLATE algorithm. To compare against many other software compression algorithms (e.g., Snappy, zstd, LZ4, etc.), please refer to the publicly available in-memory benchmarks based on the Silesia Corpus (for example, lzbench [1]). The *synchronous* throughput of IAA's (de)compression can be obtained from Figures 1 and 2 based on the size of the datasets. For example, the fixed-DEFLATE compression on the *nci* dataset reaches 1800 MB/s, and its decompression - 4600 MB/s on a single engine. These numbers are expected to be lower than the *asynchronous* streaming (de)compression (using the non-blocking mode of IAA) which we do not characterize in this paper.

We then profile (Figure 3) IAA's (de)compression for many small 4 KB sized chunks for the same datasets. As previously mentioned, the Canned—which is essentially a Static Block—mode allows sharing Huffman tables between data chunks, therefore reducing both the space and processing time when data is scattered over many small blocks. This is very useful when compressing individual memory pages. In Figure 3, the first group of bars shows the baseline compression using Dynamic Block over a continuous region. We then break it into 4 KB chunks and compress them naïvely, with the Dynamic Block, independently for each chunk. As a result, compression time explodes due to processing tables separately; the decompression time also suffers, as the Huffman tables need to be parsed. The Canned operation reduces the overhead of scattered compression. Most interestingly, it enables fast decompression of scattered data, which is only marginally higher than the continuous baseline.

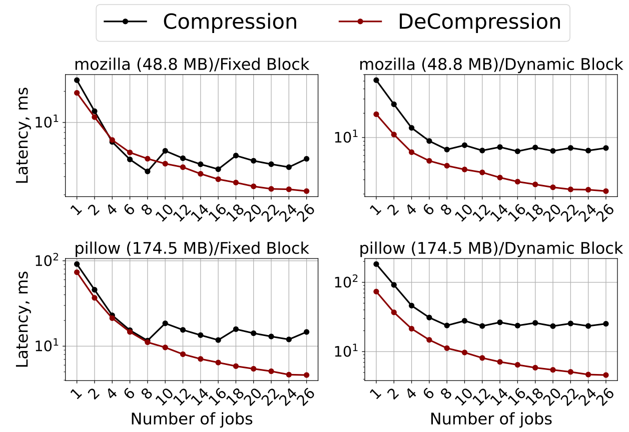

Figure 4: Latency of single-thread *synchronous* (de)compression with parallel *hardware* execution on 4 IAA devices with 8 engines each; only two benchmarks are shown for brevity.

3.2.2 Benchmarking IAA: System Integration

We now characterize the IAA hardware together with system-level aspects. We use the same datasets and setup as before.

We first evaluate how job parallelization within the IAA PEs may further speed up (de)compression. Given that our earlier SKU has more IAA engines/PEs (refer to Table 2), we use *Server #1* in this experiment. Note that at the time of writing, the QPL library did not allow crossing NUMA boundaries between the data and IAA devices. Therefore, only 4 IAA devices (32 PEs/engines in total) are utilized at most for this experiment.

There are two main ways to leverage parallelism within the IAA: with synchronous and asynchronous job submission. In the first case, a large chunk of data can be split into multiple smaller blocks, and these blocks are then submitted to multiple available PEs. The software then blocks and waits until all PEs finish processing. This allows us to reduce the time/latency of a single (de)compression job. In the asynchronous case, a *stream* of multiple blocks from potentially different dataflows/threads is supplied into the accelerator without waiting for the completion of the previous blocks, therefore utilizing the hardware at maximum capacity. This yields the highest IAA utilization and *throughput*. For fast microVM restoration, the only performance metric that matters is how fast the system can decompress a single snapshot from a single CPU thread into the microVM guest memory. We therefore only benchmark the synchronous job parallelization in this paper.

We implement parallel synchronous processing via the *non-blocking synchronous* descriptor submission. Here multiple descriptors are submitted at the same time from a single CPU thread without waiting for immediate completion of individual descriptors. Figure 4 shows that for a hardware concurrency of up to 8 compression jobs, the latency reduction reaches $4-7\times$ with respect to sequential execution. Dynamic Block compression scales worse due to the software overhead of Huffman table creation. *Decompression* scales up to 26 jobs, reaching 17× latency reduction for the *pillow* dataset. These results demonstrate how multiple IAA engines can be used to achieve even lower (de)compression latency.

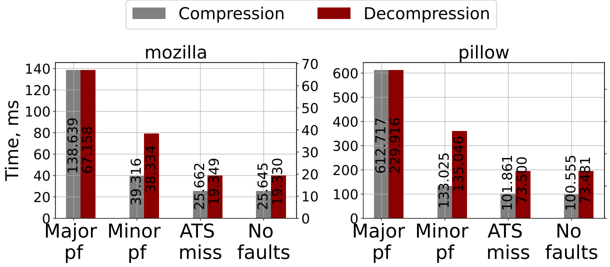

Figure 5: Impact of page faults and translation fetch on IAA performance via *PRS* with *block-on-fault* enabled

.

Until now, we have only tested the behavior of IAA when processing in-memory data, which was the majority of initial use cases for the accelerator. In-memory operation is achieved when source and destination buffers are present both in the memory and page table of the calling process. This holds when, e.g., streaming over the same buffers. However, in certain cases, data is not entirely present in memory, e.g., when processing inputs from a file or into newly allocated memory. In these cases, the accelerator must resolve page faults.

The fundamental source of page faults in systems such as IAA and DSA is the fact that they operate directly on the application's virtual address spaces via SVM [11]. As a result, similarly to CPU processing, when a page requested by the accelerator is not found in the page table, a major or minor page fault occurs. The result of the page fault handling (e.g., the translation) is then cached in the accelerator's Address Translation Service (ATS). When the translation is available in the CPU/kernel, the page fault does not happen, but the ATS must fetch the translation from the host via a translation fetch request. We now benchmark IAA in the case of page faults and translation fetch requests. We use standard 4 KB pages and an SSD disk with 550 MB/s of provisioned sequential read bandwidth. For compression, we use a single-pass Fixed Block to avoid the side effects of hybrid two-phase operations.

IAA supports two modes to handle page faults: via hardware-initiated on-demand paging via PRS or in user space with custom application-defined page fault handlers. The mode is controlled via the work queue configuration or through the PCIe device configuration if the former is not available. When PRS is enabled, the hardware can request up to *N* (specified in *PRSREQCAP* register) pages from the host concurrently. The actual page fault handling is done by the kernel through IOMMU interrupts. Figure 5 shows the time required to process descriptors in case of different hardware-

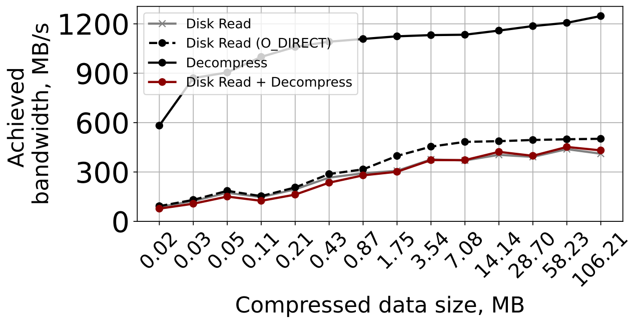

Figure 6: *Single-job, single-engine, synchronous* decompression bandwidth when reading disk inputs via *PRS* with *block-on-fault* enabled.

initiated page faults. We benchmark the worst-case scenario when the entire dataset causes page faults (12492 pages for *mozila* and 44672 pages for *pillow*). As expected, major page faults have a severe impact on the accelerator's performance. The impact of minor page faults is lower, but still $\approx 2\times$ for decompression. The ATS translation fetch has the smallest impact of ≈ 20 *us* across all pages. In general, decompression is more sensitive to page faults than compression; this is because the former is less compute-bounded.

Finally, we characterize the achieved decompression bandwidth when reading input data from an SSD (Figure 6), which is critical for our memory restoration use case. The dashed black and gray lines show achieved sequential disk read bandwidth with and without *direct I/O* (O_DIRECT). Direct I/O allows bypassing the page cache when reading files from the disk. In certain cases, this helps to reach the highest bandwidth of I/O operations. The solid black line shows the achievable bandwidth of a single-job synchronous decompression over data in memory, and the red line when reading input from the disk via hardware on-demand paging (i.e., via PRS). The latter is achieved through shared mapping of the input file into the IAA buffers and enforcing sequential I/O using *posix_fadvise*.

IAA decompression is a *streaming* operation, and it always accesses input buffers sequentially. In an ideal system, the decompression phase will completely overlap with the operation fetching data from the disk. Hence, the achieved end-to-end throughput will be decided by the slower operation - be it decompression or disk I/O. As Figure 6 shows, with default hardware on-demand paging, the achieved end-to-end bandwidth is the same as disk read without direct I/O. This demonstrates that IAA streaming processing can entirely overlap with disk I/O. It is $10-15\%$ lower, however, than the achievable bandwidth of reads with direct I/O, because IAA communicates with the disk via the OS page cache when running over PRS. A way to further improve IAA over data from disk is to replace PRS with application-specific page fault handling. However, in that case, the end-to-end behavior depends on whether IAA is configured with enabled *block-on-fault*, a feature that allows the accelerator to block and wait until data becomes available. Without block-on-fault, IAA terminates with partial completion and cannot continue decompression from the place where it stopped; as a result, the job needs to restart from scratch. Given the streaming nature of IAA, it is possible to entirely close the gap between direct disk I/O and decompression by using block-on-fault in combination with O_DIRECT reads in a custom page service handler, either in a driver or user space. We leave this to future work.

4 Sabre Design

4.1 Memory Prefetching Accelerator

We use the insights from the characterization study of Section 3 to design Sabre. Sabre is a hardware-accelerated memory snapshotting and restoration system for microVMs that is agnostic to the underlying algorithm used to identify dirty pages and create VM snapshots.

Sabre is designed to efficiently compress the guest VM physical pages to create snapshots, such that they can be decompressed (and mapped) quickly when restoring the snapshot upon a function invocation resulting in a cold start. As an input, Sabre accepts a vector of addresses for each of the guest physical memory pages which need to be placed in the snapshot, according to the underlying dirty page selection mechanism. It then compresses pages using IAA and writes them in a file. During the VM restoration process, Sabre uses fast IAA decompression in combination with efficient sequential disk I/O to quickly fetch the pages from the snapshot, decompress, and install them in the target VM's physical memory. The main goal of Sabre is to hide the decompression latency as much as possible behind the disk I/O and page fault handling (when mapping pages) such that the overhead of decompression is minimized. This is possible to achieve given the streaming nature of IAA decompression.

Sabre shows that irrespective of the method used to identify dirty pages and create VM snapshots, hardware-accelerated compression, and restoration can have a significant impact on performance. In the simplest case, dirty pages can be identified by the page tracking mechanism in the VMM (e.g., Firecracker's *Diff* snapshots); in more complicated cases, custom algorithms can be used (e.g., different working set estimation techniques [64, 72]).

Figure 7-A shows an overview of Sabre's snapshot creation pipeline along with two designs for memory prefetching, both of which are used by Sabre under different scenarios.

Snapshot creation: We first describe our snapshot creation process, which is based on two observations. First, creating a snapshot is outside of the VM restoration's critical path, so the objective is selecting the compression algorithm that achieves the highest compression ratio, which as we showed in Section 3 is dynamic DEFLATE. Second, since the VM dirty pages are distributed in a non-contiguous manner across the guest's physical memory space, the (de)compressor should operate over separate (often small) chunks of memory. As

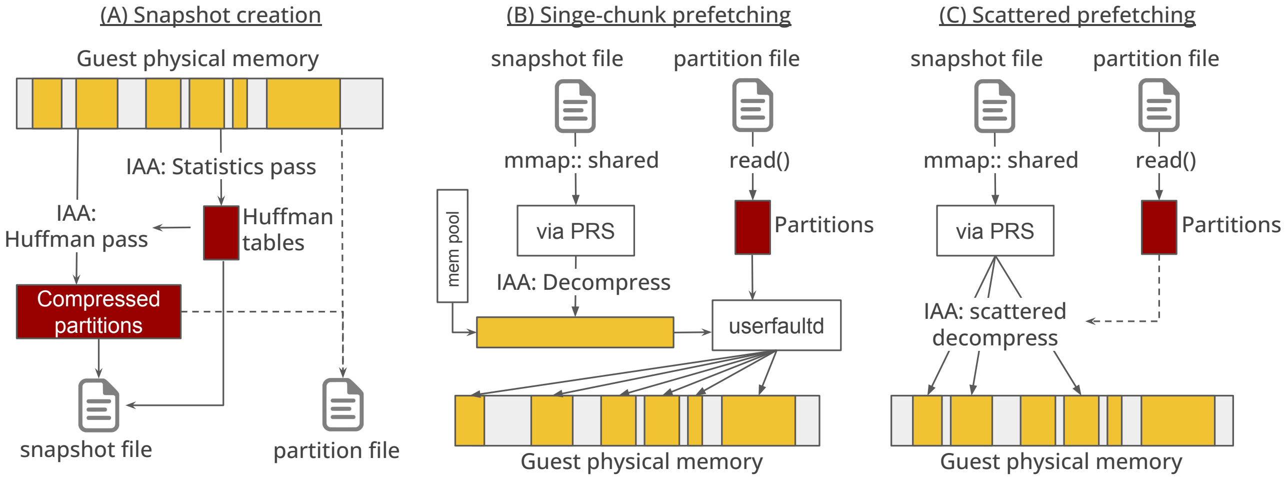

Figure 7: High-level overview of snapshotting with Sabre.

Figure 3 shows, the "Canned" style compression works best in this case: it enables implementing static DEFLATE with pre-computed Huffman tables, which closely resembles the efficiency of Dynamic DEFLATE. Sabre's snapshot creation process first runs IAA in Statistics mode to sample the statistical distribution of data in all dirty pages and create appropriate Huffman tables. It then compresses the scattered regions of dirty pages with these Huffman tables. The resulting compression stream alongside the Huffman tables is written into the snapshot file. For experimentation reasons, we also enable using Dynamic DEFLATE as well. In addition to the snapshot file itself, Sabre also writes the partition file containing the "schema" of the dirty pages, i.e., the offsets and original/compressed sizes of each partition.

Memory prefetching: The memory prefetching process is more complicated to engineer as it is on the critical path of VM restoration, and therefore needs to be carefully optimized. The main trade-off Sabre must navigate is balancing the desire to handle all partitions of dirty pages as a single contiguous memory region and the cost that comes with that.

Handling all partitions of dirty pages as a single continuous memory region is better for the accelerator, as continuous DMA is more efficient than scattered DMA, and it is also better for the PRS and disk I/O, as the underlying PRS-initiated page faults are sequential. The latter works well with sequential disk reads, therefore yielding the best utilization in terms of disk bandwidth. However, continuous decompressed partitions need to be placed by the same addresses in the guest VM physical memory as in the original VM when the snapshot was taken. Sabre implements it using *userfaultfd*, which comes at the cost of memory copy.

We implement this approach in Sabre's *single-chunk memory prefetching* shown in Figure 7-B. To reduce the overhead of allocating the decompression buffers, Sabre can optionally use a pre-allocated memory pool for the buffer for the time of decompression. The size of the pool is bounded by the sum of the sizes of dirty pages of the VMs currently restored simultaneously. This space is reusable across different restoration processes and therefore does not consume much memory. However, users of Sabre can always disable the memory pool (at $\approx 10\%$ cost of memory restoration) if the pool's impact on the memory density is an important concern. Sabre's memory prefetching relies heavily on the PRS hardware mechanism to bring snapshots from the disk. This is achieved by running IAA against a shared not pre-faulted (i.e. the actual file I/O gets initiated by PRS) mapping of the snapshot file. As Figure 6 shows, default PRS is near-optimal at handling IAA inputs from the disk, and it allows hiding the decompression time by overlapping it with the disk I/O. In most cases, the difference with sequential disk read bandwidth is marginal. We confirm that running IAA over pre-faulted or pre-fetched (via *read*) snapshot files is much slower than via PRS.

To address the high cost of partition placement when treating the entire memory region as contiguous, Sabre also implements memory prefetching based on scattered IAA decompression (Figure 7-C). Here, Sabre directly DMAs decompressed partitions into the right locations in the guest's physical memory, while still handling inputs from disk via PRS. This allows the system to bypass page installation, however, it makes the IAA hardware less efficient due to the large number of scattered DMAs and the bookkeeping of the corresponding descriptors (the current implementation of IAA does not allow chaining and batching of descriptors, so each one must be submitted separately by software). In addition, splitting the decompression stream into multiple jobs hurts the efficiency of PRS at reading data from the disk, which further slows down memory prefetching. The latter can be addressed by implementing a custom user-space page fault handler, as discussed in Section 3.

In both designs for memory prefetching, IAA decompression can be done using a single IAA job/engine or parallelized across all available engines. This is implemented via non-blocking job submission with a rotating pool of descriptors. In this mode, Sabre attempts to submit N decompression jobs at the same time, where N is the desired concurrency degree or the number of available free engines (whichever is smaller). Upon asynchronous out-of-order completion, the corresponding descriptors are returned to the pool for later reuse. This enables a streaming operation for Sabre's decompression when multiple engines are used. Note that a single IAA engine in blocking synchronous mode is capable of achieving $\approx$ 1.2 GB/s at decompressing our snapshot datasets. Since this is higher than our disk read bandwidth, we always use a single IAA job in all remaining experiments, unless otherwise noted.

The exact operation of the memory prefetching unit, such as the choice of the restoration design (between single-chunk and scattered), the compression style (dynamic or static DEFLATE) for snapshot creation, the number of concurrent decompression jobs, etc. are configured by Sabre during runtime. The desired configuration can be selected differently for each microVM's snapshotting/restoration call. Next, we microbenchmark our memory prefetching unit under different configurations and types of snapshots.

4.2 Microbenchmarking Memory Restoration

We now analyze the two design options for memory prefetching shown in Figure 7 using a dataset of microVM dirty memory snapshots with different sparsities. We create synthetic datasets from the *pillow* snapshot (Section 3) that range from *most scattered*, when each page is separated, to a case with few large contiguous regions of dirty pages. In practice, the pattern depends on the underlying mechanism used to identify snapshot pages and the applications running in the VMs.

Figure 8 shows the restoration time with each of the two restoration mechanisms when the restoration is done in hardware and software. The x-axis shows the sparsity of the dataset. In all cases, the total size of the dataset is 174.5 MB. The sparsity index denotes the number of pages in continuous regions; each region is separated by its neighbor via an empty page, which is not included in the snapshot. For instance, in sparsity 1, each individual page is separated. We use a single IAA engine in all experiments. As the top figure shows, the scattered memory prefetching design outperforms single-chunk prefetching for sparsities of more than 4 pages. This is because this design avoids additional page copying during the installation phase via userfaultfd. However, when memory partitions become as sparse as every page or two pages, the overhead of scattered DMA and suboptimal PRS handling make scattered prefetching slower than single-chunk. Given this, Sabre uses different memory prefetching strategies depending on the sparsity of the underlying snapshots.

The dashed line in Figure 8 (Top) denotes the time required to restore memory from uncompressed snapshots. We optimize this path similarly to REAP [64], where the whole snapshot is fetched as a single continuous disk read via O_DIRECT file I/O and installed via $userfaultfd$. Sabre's memory restoration overperforms prefetching of uncompressed snapshots by up to 1.9×. Note that the achieved compression ratio on this dataset is 2.2×, meaning that the theoretical upper bound of memory restoration speed-up with respect to uncompressed baselines is also 2.2×. Sabre's memory prefetching is very close to this because it hides decompression behind disk I/Os. For faster disks, these results would still hold.

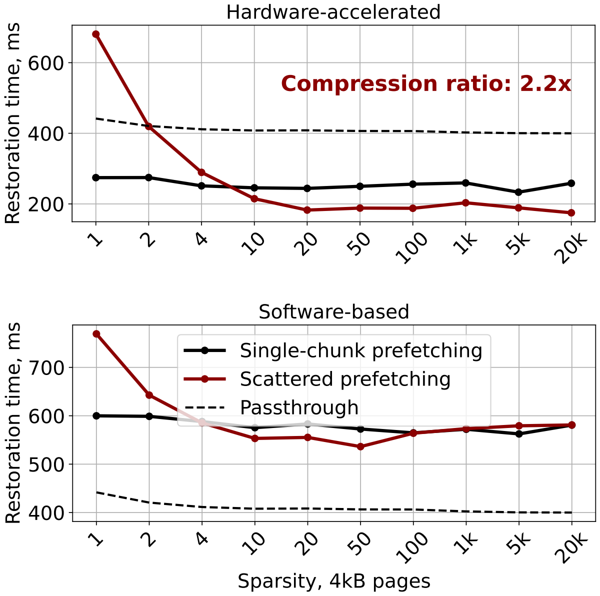

Figure 8: Single-chunk and scattered prefetching across different page sparsities; *passthrough* denotes the time to read uncompressed snapshots; the system runs at 2.3 GHz with a single IAA engine.

Figure 8 (bottom) shows the same results when running Sabre with software-based DEFLATE decompression. Across all different memory sparsities, the overhead of software decompression kills the speed-up of fetching deflated snapshots. This demonstrates that hardware acceleration is required to make snapshot compression practical.

For the sake of completeness, we additionally integrate other *software* compression algorithms optimized for performance in Sabre: *Snappy*, *Zstandard (zstd)*, and *LZ4*. We run them on a dedicated CPU core under the highest possible turbo boost frequency of *4 GHz* and repeat the snapshotting microbenchmark experiment. As Figure 9 shows, across all sparsities, the memory restoration with IAA outperforms these algorithms. The restoration times under *Zstd level-3,10*, and *20* are very close to IAA results and are much lower than prefetching uncompressed snapshots. However, as Figure 9 (Bottom) shows, they require a *significant* amount of CPU resources at the snapshot creation stage (up to several seconds at the highest frequency), and they do not demonstrate performance as high as IAA when running at lower CPU frequencies. We do not show results for *LZ4* as its compression ratios are low.

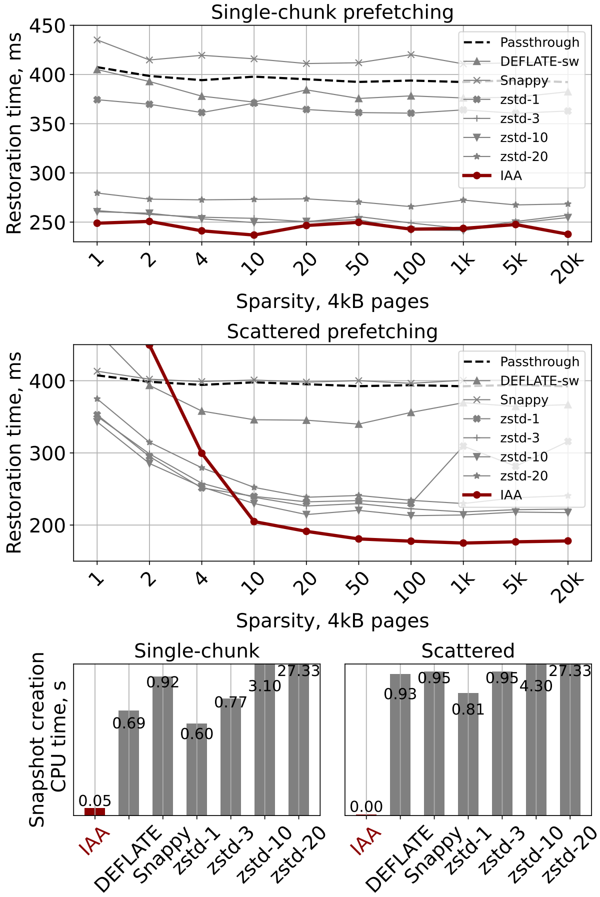

Figure 9: Sabre with different fast *software* (de)compression algorithms running on dedicated CPU cores under the turbo boost frequency of *4 GHz*; the bottom plots show CPU resources (in seconds) consumed to make the snapshots (averaged over sparsities).

4.3 Full System Implementation

We now discuss how we integrate Sabre's memory prefetching unit in an end-to-end serverless framework. We choose Firecracker microVMs [17] as the target serverless sandbox. Firecracker is the current industry-leading VMM, and it already provides good VM snapshotting capabilities. Most recent work in this space, such as REAP [64], is also based on Firecracker. To build the end-to-end serverless pipeline, we partially reuse the infrastructure of vHive – an academic framework for serverless used to showcase the effectiveness of REAP working sets. vHive allows running serverless Docker images inside Firecracker microVMs via *firecracker-containerd*. vHive also extends the native Firecracker Go SDK to support snapshotting and implements a simple orchestrator to simplify managing the serverless environment.

We write Sabre's snapshotting unit in ≈ 3500 LoC in C++17 excluding unit tests and benchmarks, using Intel's QPL library v1.3.1 and build it as a dynamic library. We then integrate it with Firecracker VMM v1.5.0 in only 50 LoC in Rust via FFI. Sabre runs in the default Firecracker's snapshotting/restoration thread, and does not require additional CPU resources. To expose Sabre's snapshotting to the higher layers of serverless frameworks, we additionally extend the Firecracker's Go SDK 1.0.0 with several new APIs. Figure 10 shows the simplified overview of the full system design. Similarly to vHive [64], our infrastructure enables running end-to-end serverless applications in Firecracker microVMs with standard Docker environments.

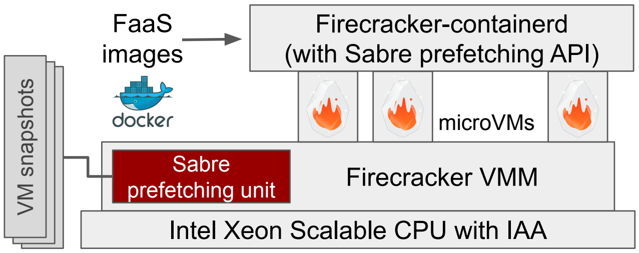
FaaS images
docker
VM snapshots
Firecracker-containerd (with Sabre prefetching API)
microVMs
Sabre prefetching unit
Firecracker VMM
Intel Xeon Scalable CPU with IAA

Figure 10: Sabre's integration in end-to-end serverless frameworks.

5 End-to-End Evaluation

Methodology: We evaluate Sabre on a large set of end-to-end serverless benchmarks from vSwarm [66], FunctionBench [44, 45], and SeBS [27]. We modify the benchmarks to run grpc servers and support server reflection so that we can invoke functions using grpcurl. The set includes only one synthetic benchmark – *python-list* implementing traversing a large sparse Python list; we include it to showcase the upper-bound of compression achievable with Sabre. We slightly modify the *dna-visualisation* benchmark from SeBS to make it use different DNA sequence datasets across different invocations (the default benchmark is always based on the same dataset; *dna-visualisation-1*). Similarly, we modify the datasets for the *model training* benchmark to use smaller 2 MB and larger 10 MB images. All other benchmarks are taken from the aforementioned suites. In all experiments, we use a single IAA engine for memory prefetching.

Sabre's memory prefetching unit is agnostic to the underlying mechanism of creating a snapshot. It can be used with dirty page-based snapshots, with working sets, and even when snapshotting the whole guest's VM memory. Since the latter is not practical for realistic serverless workloads, and therefore rarely used in practice, we only evaluate Sabre on the first two options. In all experiments, we use our testbed with the configuration shown in Table 2. We focus our evaluation on two metrics: (1) how well Sabre is able to compress snapshots of different types, and (2) what is the impact of decompression on the end-to-end cold start time.

End-to-end performance impact: Figure 11 shows the result of running the default dirty page-based snapshots of Firecracker (Diff snapshots [2]) with Sabre. Diff snapshotting is enabled via dirty page tracking in the hypervisor. We find Diff snapshots to be relatively large (a few hundred MB)

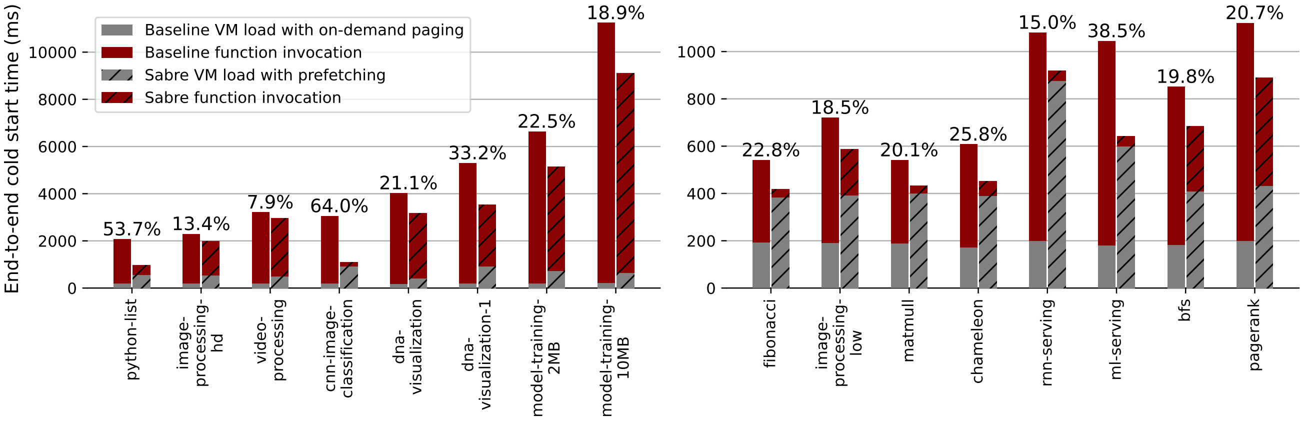

Figure 11: End-to-end evaluation of serverless cold starts with Sabre on Firecracker's default Diff snapshots with prefetching; annotated numbers show speedup of Sabre over the baseline.

and coarse-grained for all applications, which means that the scattered prefetching (Figure 7-C) works best in this case.

Figure 11 compares the end-to-end cold start latency when serving requests from a VM restored with on-demand paging (default mechanism in Firecracker) and via prefetching with Sabre. We find that in all cases, Sabre is able to compress Diff snapshots by up to 4×, and 2.5× on average. This is a significant reduction in storage requirements, given the large original size of Diff snapshots, ranging from hundreds of megabytes to several gigabytes. This is even more significant given that in serverless deployments, a physical node can host hundreds or thousands of snapshotted VMs [17].

Most importantly, in all cases, the hardware-accelerated decompression in Sabre allows restoring a compressed snapshot without any negative impact on the end-to-end latency. Moreover, in some applications, we observe up to 60% lower cold start overhead, enabled by the fast memory prefetching. The speed-up and compression effect are particularly evident for our synthetic *python-list* benchmark, where the dirty memory, i.e., the Python runtime heap storing the list, is well compressible. The same holds for *dna-visualization* as well.

Optimizing the VM snapshotting strategy: While Diff snapshots and dirty page tracking currently represent the industry standard in microVM snapshotting, they are not the most efficient way to restore memory via prefetching. More efficient mechanisms are enabled via working set estimation. We implement working sets in Sabre which are used with our memory prefetching unit for evaluation purposes. Our implementation is based on the record-and-replay technique. Similarly to the original paper [64], Sabre records working sets during the first invocation of serverless functions after the standard restoration from vanilla Firecracker snapshots. We intercept in user-space the guest memory page faults and record all accessed addresses in a vector. The recorder then groups the accessed pages to form continuous chunks, whenever possible, and saves them in a WS file. The REAP restoration uses the *passthrough* functionality of Sabre's memory prefetching unit, which resembles the original REAP specification [64], i.e., direct I/O disk reads combined with page installation in the guest physical memory via *userfaultfd*. After prefetching, the hypervisor continues serving the rest of the pages outside of the working set via standard on-demand paging.

REAP working set files tend to be much more scattered than Firecracker's Diff snapshots. We make a similar observation in our applications as well. Therefore, memory restoration through single-chunk prefetching works best in this case, and we configure Sabre accordingly for REAP snapshots. Table 3 shows the achieved compression ratios of REAP working set files and the corresponding prefetching speedup when using Sabre. Figure 12 shows the end-to-end cold start latencies across the same set of serverless benchmarks as before.

Table 3 shows that working set files are better compressible than dirty pages. The compression ratio reaches up to 4.7×; 3.2× on average. The prefetching time itself with Sabre is accelerated by $25-55\%$. On the synthetic application *python-list*, the speedup reaches 70%. As expected, there is an obvious correlation between the achieved compression ratio and prefetching speedup. End-to-end, working set prefetching speedup translates in up to 20% of improvement in the application's cold start latency. For several of the examined applications, the speedup is diminished due to the working set size being small or their compute time being high. In the latter case, the impact of accelerated restoration gets hidden behind computing. In general, compression of snapshots/working set files is more impactful on applications with larger working sets, especially if they are memory-bounded (as in the case of the *python-list* benchmark). Even when Sabre does not accelerate cold start time significantly, it still greatly reduces the size of the working set files without a negative impact on prefetching latency and/or CPU cycles for restoration.

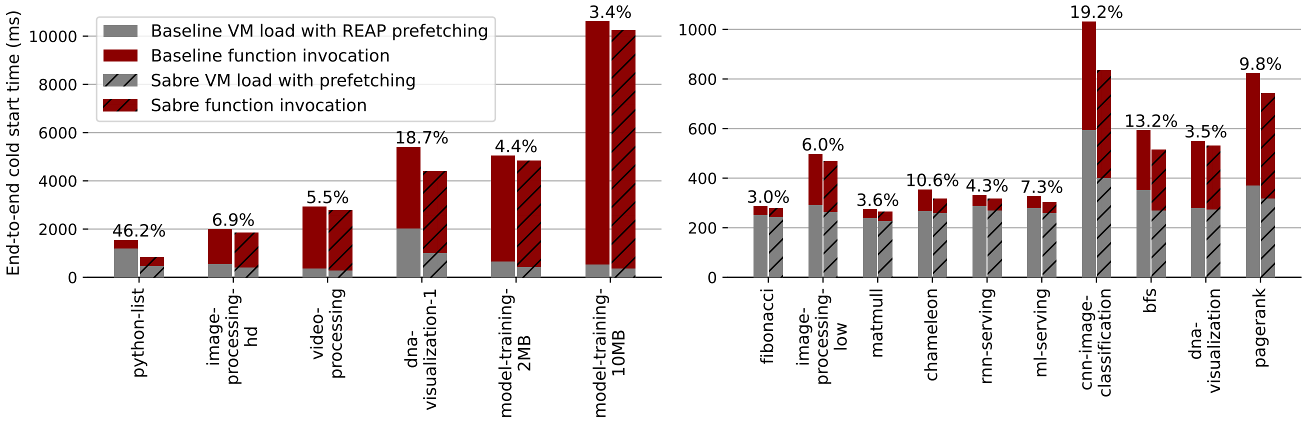

Figure 12: End-to-end evaluation of serverless cold starts with Sabre on REAP snapshots; annotated numbers show the speedup over REAP.

Table 3: Compression ratios and page prefetching speedup of Sabre over REAP working sets.

Application	**Size of REAP WS (MB)**	**Compression ratio**	**Sabre's prefetching speedup**
fibonacci	12.93	2.64×	29.02%
python list	405.23	14.82×	70.81%
image-processing-low	27.67	3.74×	45.17%
image-processing-hd	120.66	2.73×	35.27%
matmull	13.02	2.62×	32.65%
chameleon	18.45	2.90×	36.05%
video-processing	44.91	3.21×	43.30%
rnn-serving	17.64	2.50×	26.59%
ml-serving	22.10	2.67×	35.00%
cnn-image-classification	136.36	3.10×	38.73%
bfs	44.22	4.29×	49.39%
dna-visualization	16.04	2.76×	28.73%
dna-visualization-1	720.95	4.70×	55.01%
pagerank	62.23	2.94×	34.60%
model-training-2MB	171.28	3.60×	43.48%
model-training-10MB	111.71	3.57×	45.54%

6 Discussion and Future Work

6.1 Prefetching on Faster Disks and Networks

While the performance impact of memory prefetching becomes less critical as the speed of disks and NVMe/persistent memory devices increases, Sabre benefits when it comes to storage space reduction remain. This is even more critical, given the higher cost per Byte of new memory technologies. CPU-free memory decompression, especially at zero negative impact on end-to-end latency will always be beneficial, independent of the underlying storage technology. In future work, we plan to evaluate Sabre on CXL-enabled memory devices and explore the potential of serving compressed snapshots from them. The byte-addressable organization of CXL and other similar memory disaggregation devices will make the integration with near-memory accelerators, such as IAA, much more efficient than when using commodity disks, essentially eliminating all overheads of PRS discussed in Section 3.

Additionally, the streaming nature of hardware accelerators, such as IAA, allows combining Sabre with any streaming I/O, including networking. This makes Sabre attractive for *remote snapshotting* – another technology in serverless microVMs where snapshots and/or working set files are served from centralized storage or a remote server. Fast streaming decompression can have a dramatic reduction of network bandwidth consumed for snapshotting, which is critical for highly multi-tenant and geographically distributed datacenters.

6.2 Further Optimizing Sabre

The biggest limitation in our current design of Sabre is its integration with the disk via the standard IAA's PRS mechanism. This disallows bypassing the OS page cache when feeding an input to IAA, and results in lower effective bandwidth utilization than direct I/O. This can be optimized by redesigning the default PRS in one of two ways.

First, PRS can be replaced with user-space page fault handling for input buffers. Disk I/O can be initiated separately in user space using the *read* system call combined with *O_DIRECT* file opening. In this case, the IAA page fault handler only needs to wait until the disk DMA catches up with the sequential data transfer. This requires enabling the *block-on-fault* feature of IAA, otherwise, the input stream would need to be resubmitted from the beginning every time IAA reaches pages that have not been fetched yet. Alternatively, one can directly connect the disk's and IAA's DMA engines, and allow streaming of compressed inputs directly into the accelerator through a small FIFO buffer. However, this can only be done in the host kernel in custom IAA drivers and is challenging to implement in a scalable way.

A second limitation in the current design's single-chunk prefetching (Figure 7-B) is using the COPY-based `userfaultfd` mechanism. The original REAP snapshots [64] suffer from the same inefficiency. Starting with Linux kernel 5.13, `userfaultfd` can be handled via minor page faults and *UFFDIO_CONTINUE* page installation. Instead of copying pages from the continuous buffers, one can install them from the page cache via the underlying page table modification with zero-copy. This will, however, complicate managing the decompression/prefetching buffers to ensure they are never reclaimed, while the corresponding microVM is running.

6.3 Beyond MicroVM Snapshotting

VM memory compression and fast restoration go well beyond microVM snapshotting and serverless. For example, VM live migration [18,59] can benefit from hardware-accelerated compression and on-the-fly decompression of memory pages. The recent work [14] is exploring this opportunity for KVM. This can dramatically accelerate applications heavily relying on VM migration, including VM bin-packing for cloud management [55], low-latency cloud-native applications such as vRAN [49, 68], and fast fault-tolerance solutions [63]. We plan to extend Sabre to benefit these applications as well.

7 Conclusion

MicroVM snapshotting and restoration via page prefetching is the most effective technique for reducing the cold start overhead in serverless. Memory compression is a promising technique to reduce the size of VM snapshots and speed up prefetching during memory restoration, but it is only efficient if decompression is fast. We showed that emerging hardware accelerators for *general-purpose* compression are suitable for microVM memory restoration as well. We first characterized the Intel IAA accelerator and then designed Sabre, a system for fast prefetching of VM memory from compressed snapshots. We showed that Sabre compresses snapshots of real serverless applications up to 4.5×, and speeds up prefetching by up to 55% compared to uncompressed baselines. This results in up to 20% of end-to-end performance improvement for cold function invocations over the most optimized snapshotting technologies.

Acknowledgements

We sincerely thank our shepherd, the Intel IAA team, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was supported in part by NSF CAREER Award CCF-2326182, a Sloan Research Fellowship, an Intel Research Award, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] In-memory benchmarks for compression alogorithms. https://github.com/inikep/lzbench.

[2] Firecracker snapshotting documentation. https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md, 2023.

[3] Google gvisor. https://gvisor.dev/docs/, 2023.

[4] Intel data streaming accelerator (dsa), architecture. https://www.intel.com/content/www/us/en/content-details/759709, 2023.

[5] Intel in-memory analytic accelerator (iaa), architecture specification. https://www.intel.com/content/www/us/en/content-details/721858, 2023.

[6] Intel in-memory analytic accelerator (iaa), use guide. https://www.intel.com/content/www/us/en/content-details/780887, 2023.

[7] Intel query processing library (qpl). https://www.intel.com/content/www/us/en/developer/tools/query-processing-library/overview.html, 2023.

[8] Introducing intel scalable i/o virtualization. https://www.intel.com/content/www/us/en/developer/articles/technical/introducing-intel-scalable-io-virtualization.html, 2023.

[9] Linux kernel patch to enable enqcmd. https://lore.kernel.org/lkml/20210920192349.2602141-1-fenghua.yu@intel.com/T/#rd6d542091da1d1159eda0a44a16e57d0c0dfb209, 2023.

[10] The operstack foundation. kata containers - the speed of containers, the security of vms. https://katacontainers.io/, 2023.

[11] Shared virtual addressing (sva) with enqcmd. https://docs.kernel.org/next/x86/sva.html, 2023.

[12] Technical overview of intel 4th gen xeon scalable processor family. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html#gs.0466t8, 2023.

[13] Implementation of sabre. https://github.com/barabanshek/sabre, 2024.

[14] Kvm live migration with iaa in-memory compression. https://lore.kernel.org/all/20240319164527.1873891-1-yuan1.liu@intel.com/T/, 2024.

[15] Sabre iaa benchmarks. https://github.com/barabanshek/IAA_benchmarking.git, 2024.

[16] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. Data compression accelerator on ibm power9 and z15 processors : Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2020.

[17] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[18] Mahdi Aiash, Glenford Mapp, and Orhan Gemikonakli. Secure live virtual machines migration: Issues and solutions. In *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, pages 160–165, 2014.

[19] Anjali, Tyler Caraza-Harter, and Michael M. Swift. Blending containers and virtual machines: A study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 101–113, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Amine Barrak, Fabio Petrillo, and Fehmi Jaafar. Serverless on machine learning: A systematic mapping study. *IEEE Access*, 10:99337–99352, 2022.

[22] Luiz Barroso and Urs Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[23] Jacob Breiholz, Farah Yahya, Christopher J. Lukas, Xing Chen, Kevin Leach, David Wentzloff, and Benton H. Calhoun. A 4.4 nw lossless sensor data compression accelerator for 2.9x system power reduction in wireless body sensors. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1041–1044, 2017.

[24] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. Fpga acceleration of zstd compression algorithm. pages 188–191, 06 2021.

[26] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. Fpga acceleration of zstd compression algorithm. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 188–191, 2021.

[27] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014.

[29] Sebastian Deorowicz. Silesia compression corpus. https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia, 2023.

[30] P. Deutsch. Rfc1951: Deflate compressed data format specification version 1.3, 1996.

[31] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Sérgio Fernandes and Jorge Bernardino. What is bigquery? In *Proceedings of the 19th International Database Engineering & Applications Symposium*, IDEAS '15, page 202–203, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[34] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 52–59, 2015.

[35] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Yasunori Goto. Kernel-based virtual machine technology. *Fujitsu scientific & technical journal*, 47, 07 2011.

[38] Junyeong Han, Sungeun Kim, Sungyoung Lee, Jaehwan Lee, and Sung Jo Kim. A hybrid swapping scheme based on per-process reclaim for performance improvement of android smartphones (august 2018). *IEEE Access*, 6:56099–56108, 2018.

[39] Seth Jennings. The zswap compressed swap cache. https://lwn.net/Articles/537422/, 2023.

[40] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Hoyong Jin, Donghun Jeong, Taewon Park, Jong Hwan Ko, and Jungrae Kim. Multi-prediction compression: An efficient and scalable memory compression framework for gp-gpu. *IEEE Computer Architecture Letters*, 21(2):37–40, 2022.

[42] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, jun 2015.

[43] Sagar Karandikar, Aniruddha N. Udipi, Junsun Choi, Joonho Whangbo, Jerry Zhao, Svilen Kanev, Edwin Lim, Jyrki Alakuijala, Vrishab Madduri, Yakun Sophia Shao, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. Cdpu: Co-designing compression and decompression processing units for hyperscale systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.

[45] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 477, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[47] Kamil Kojs. A survey of serverless machine learning model inference, 2023.

[48] Reese Kuper, Ipoom Jeong, Yifan Yuan, Jiayu Hu, Ren Wang, Narayan Ranganathan, and Nam Sung Kim. A quantitative analysis and guideline of data streaming accelerator in intel 4th gen xeon scalable processors, 2023.

[49] Nikita Lazarev, Tao Ji, Anuj Kalia, Daehyeok Kim, Ilias Marinos, Francis Y. Yan, Christina Delimitrou, Zhiru Zhang, and Aditya Akella. Resilient baseband processing in virtualized rans with slingshot. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 654–667, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. High-throughput fpga-based hardware accelerators for deflate compression and decompression using high-level synthesis. *IEEE Access*, 8:62207–62217, 2020.

[51] Sang Muk Lee, Jung Hwan Oh, Ji Hoon Jang, Seong Mo Lee, Ji Kwang Kim, and Seung Eun Lee. Live demonstration: An fpga based hardware compression accelerator for hadoop system. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 744–745, 2016.

[52] Weigang Li. Optimize genomics data compression with hardware accelerator. In *2017 Data Compression Conference (DCC)*, pages 446–446, 2017.

[53] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[54] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[55] Khine Moe Nwe, Mi Khine Oo, and Maung Maung Htay. Efficient resource management for virtual machine allocation in cloud data centers. In *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*, pages 419–420, 2018.

[56] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 377–388, New York, NY, USA, 2012. Association for Computing Machinery.

[57] Weikang Qiao, Jieqiong Du, Zhenman Fang, Libo Wang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. pages 291–291, 05 2018.

[58] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SOCC*. 2012.

[59] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. Vm live migration at scale. *SIGPLAN Not.*, 53(3):45–56, mar 2018.

[60] Sudhir Satpathy, Vikram Suresh, Raghavan Kumar, Vinodh Gopal, James Guilford, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, Ram Krishnamurthy, Vivek De, and Sanu Mathew. A 1.4ghz 20.5gbps gzip decompression accelerator in 14nm cmos featuring dual-path out-of-order speculative huffman decoder and multi-write enabled register file array. In *2019 Symposium on VLSI Circuits*, pages C238–C239, 2019.

[61] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 733–750. Association for Computing Machinery, 2020.

[62] Rajesh Thallam. Bigquery explained: An overview of bigquery's architecture. https://cloud.google.com/blog/products/data-analytics/new-blog-series-bigquery-explained-overview, 2023.

[63] Wen-Hsiu Tsai, Po-Jui Tsao, and Che-Rung Lee. Fvmm: Fast vm migration for virtualization-based fault tolerance using templates. In *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–16, 2022.

[64] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7, 2020.

[66] vhive-serverless. vswarm: Serverless framework for multi-tenant serverless computing. https://github.com/vhive-serverless/vSwarm, 2023. Accessed: Dec 6, 2023.

[67] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, 2019.

[68] Jiarong Xing, Junzhi Gong, Xenofon Foukas, Anuj Kalia, Daehyeok Kim, and Manikanta Kotaru. *Enabling Resilience in Virtualized RANs with Atlas*. Association for Computing Machinery, New York, NY, USA, 2023.

[69] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[70] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2017.

[71] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. *SIGPLAN Not.*, 46(7):87–98, mar 2011.

[72] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, page 87–98, New York, NY, USA, 2011. Association for Computing Machinery.

[73] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 1–14, New York, NY, USA, 2022. Association for Computing Machinery.