FaaS in the Age of (sub-)μs I/O: A Performance Analysis of Snapshotting

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

Chloe Alverti
National Technical University of Athens
xalverti@cslab.ece.ntua.gr

Vasileios Karakostas*
University of Athens
vkarakos@di.uoa.gr

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

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

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

ABSTRACT

Although serverless computing brings major benefits to developers, the widespread adoption of Function-as-a-Service (FaaS) creates severe challenges for the cloud providers. Irregularity in function invocation patterns and the high cost of *cold starts* has led them to allocate precious DRAM resources to keep function instances always *warm*, a clearly sub-optimal and inflexible approach. To cope with this issue, both state-of-the-art and state-of-practice approaches consider *snapshotting* as a viable mitigation, thus directly associating cold start latency with storage performance.

Prior studies consider storage to be inert, rather than the evolving hierarchy that it truly is. In this work, we evaluate cold start and warm function invocations on instances restored from snapshots residing on devices across different layers of the modern storage hierarchy. We thoroughly analyze and characterize the observed behavior of multiple workloads and identify fundamental trade-offs among the devices. We conclude by motivating and providing suggestions for the inclusion of the modern storage hierarchy as a decisive factor in serverless resource provisioning.

KEYWORDS

serverless; FaaS; snapshots; storage; virtualization; persistent memory

ACM Reference Format:
Christos Katsakioris, Chloe Alverti, Vasileios Karakostas, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2022. FaaS in the Age of (sub-)μs I/O: A Performance Analysis of Snapshotting. In *The 15th ACM International Systems and Storage Conference (SYSTOR '22), June 13–15, 2022, Haifa, Israel.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3534056.3534938

*This work was done while the author was at National Technical University of Athens.

SYSTOR '22, June 13–15, 2022, Haifa, Israel

ACM ISBN 978-1-4503-9380-5/22/06…$15.00
https://doi.org/10.1145/3534056.3534938

1 INTRODUCTION

Serverless computing has recently received significant attention by both industry and academia [36, 23, 58, 47]. Developers create applications based on the notion of functions, avoiding the explicit allocation of resources (e.g., VMs); instead, service providers are responsible for allocating and managing resources when functions are invoked. Serverless provides three important benefits: (i) it abstracts away platform details allowing the programmer to write code without worrying about infrastructure management; (ii) it enables resource and energy efficient execution as resources are consumed according to workloads' demands, and (iii) it supports a pay-per-use model, which is beneficial for both application developers, who avoid the error-prone task of resource provisioning, and for service providers that can manage their own resources more efficiently. These benefits designate serverless as the next step in the evolution of cloud computing.

However, serverless still faces many challenges, with the problem of cold starts being one of the most important across different platforms [51, 47, 53, 54, 58]. In the FaaS paradigm, functions can execute instantly if the corresponding application code/execution environment is already loaded in memory *(warm start)*. Otherwise, there is a start-up penalty as dependencies must be loaded from storage *(cold start)* for the invoked function to run. In the case of virtual machines, cold start entails booting and launching a VM image from storage to execute the requested function, while warm start presupposes a running VM residing in memory.

Several studies [17, 53, 54, 47, 51] have shown that cold start latencies can slow down function execution by orders of magnitude. Thus, cold starts impact both users and providers. On one hand, users may be reluctant to embrace serverless, particularly for applications with critical latency constraints. On the other hand, providers are affected financially, as users are not charged while their requests experience cold starts [54].

To address cold starts, prior works have mainly followed two orthogonal approaches: (i) minimizing their number by keeping instances warm [47], and (ii) mitigating cold start latency to sustain user experience. In this paper we focus on the latter, where various schemes have been proposed [2, 44, 24]. In particular, we focus on snapshotting, a recently proposed and promising technique [17, 54, 51], which stores the state of a fully booted function instance in storage and then loads it upon a following cold function invocation, avoiding entirely cold boot latencies.

However, all prior works considering snapshotting assume traditional *slow* storage, such as SSDs. Storage systems nowadays support (sub)microsecond-scale I/O via emerging memories (e.g., Optane DCPM) and low-latency I/O devices (e.g., Optane NVMe) [10]. In this paper we seek to answer: What is the impact on cold start of alternative storage devices for snapshots? What are the implications to leverage persistent memory as a snapshot store? Are there new opportunities for improved resource management?

To answer these questions, we perform an extensive analysis of cold start performance when snapshots are stored in three different devices (DCPM, NVMe and flash SSD). We use Firecracker [2] as virtualization technology to sandbox serverless workloads, adopted from FunctionBench [37, 38]. Our analysis shows, among other, that high-performant storage can occasionally: (i) speed-up cold start by 8×, (ii) narrow the gap between cold and warm function invocation below ~20%, and (iii) release significant amounts of DRAM resources via direct access to storage (persistent memory). We demonstrate how various function characteristics (e.g., execution time, read-only % of working set) affect its cold start performance across different devices. Finally, we discuss how snapshot placement on a modern storage stack can be promoted to a new decisive factor for FaaS orchestration and resource management, cross-cutting scheduling decisions (e.g., warm function instances lifetimes).
In summary, the contributions of this paper are:

- We perform an extensive performance analysis of snapshotting on top of a modern storage hierarchy using multiple serverless workloads. To the best of our knowledge, this is the first work to quantify serverless execution from the snapshot storage perspective.
- We show that the selection of storage device may significantly affect the performance and close the gap between cold and warm invocations.
- We identify application characteristics and device trade-offs, and provide suggestions for leveraging the benefits of a modern storage hierarchy in the context of serverless snapshot resource management.

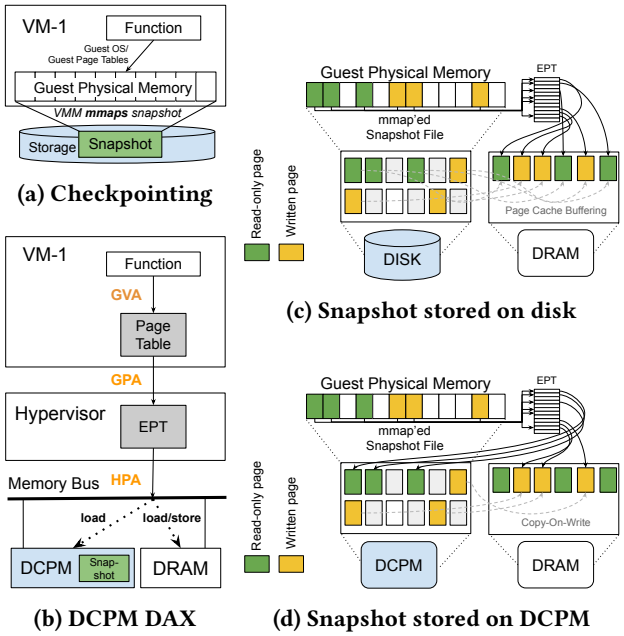


Figure 1: Cold starts using VM snapshots. Different storage media allow for buffered or direct access.

Table 1: Storage devices characteristics. For DRAM and DCPM we report the performance of a single DIMM.

Type	Latency	Read BW	Write BW	Access
DRAM	80ns	23 GB/s	23 GB/s	Direct
Optane DCPM [29]	300ns	6.8 GB/s	1.85 GB/s	Direct
Optane NVMe [30]	10μs	2.2 GB/s	2 GB/s	Buffered
SATA Flash SSD [31]	180μs	0.5 GB/s	0.5 GB/s	Buffered

2 BACKGROUND AND MOTIVATION

Lightweight virtualization. To colocate thousands of FaaS users while preserving isolation and security guarantees [2], numerous serverless platforms and sandboxing alternatives have been proposed based on containers [24, 3, 1, 9, 18, 12], unikernels [42, 40, 43, 48, 13], VMs [2, 5, 8], or other approaches [4, 20, 41, 45, 49], with VMs being among the most common choice for FaaS providers [2, 56] to serve functions. In this paper we focus on Firecracker [2], an open-source VMM (Virtual Machine Monitor) developed by AWS to power Lambda [26] and Fargate [25], providing lightweight virtualization sandboxes termed *microVMs*. Based on KVM [39] and resembling a stripped-down QEMU [11], Firecracker features a minimal emulation layer, providing storage and network access via VirtIO [46] block and net devices, backed by files and TAP [16] interfaces. Firecracker has minimal memory footprint (3MB) and can boot VMs in ~125*ms* [2].
Snapshots as a cold start mitigation mechanism. State-of-the-art [17, 54, 57] and state-of-practice [7] approaches adopt snapshots as a technique to mitigate cold start delays.

Once a function instance is fully initialized (e.g., already serving a function handler), its state and memory are serialized onto files on storage (i.e., snapshot). When a request arrives for a function with no running instance, snapshots can be used to create a new instance by loading/restoring directly the stored state and avoiding entirely the boot procedure. The newly created instance is instantly ready to serve requests. Firecracker's snapshotting design, influenced by Catalyzer [17], reports restoration latencies below 10ms [7].

Snapshot restoration. To restore a function instance from snapshot files, Firecracker (i) loads VMM's and microVM's state from the corresponding file, and (ii) creates a private mapping of the guest's snapshotted physical memory file (`mmap` – Figure 1a). The low restoration latency reported [7] is largely attributed to the design decision of letting the created mapping populate on demand. Any first-touch access on guest physical pages results in a VM-exit and a page fault on the host. These occur on the critical path of a cold start function invocation and recent studies [54] have shown that they significantly deteriorate cold start performance. However, those studies only consider traditional storage devices (e.g., flash SSDs) for snapshot files placement. In this paper, we seek to answer how the technique performs in the presence of a modern storage hierarchy, where heterogeneous and high-performant devices co-exist.

Storage trends. Storage systems evolve rapidly as microsecond I/O devices are integrated into commodity stacks, shaking the software ecosystem [10]. Storage hierarchies today include high-performant devices, such as persistent memory (e.g., 3D XPoint – Intel Optane DCPM [33]), low-latency SSDs (e.g., Intel Optane NVMe [32]), NVMe and SATA Flash SSDs [34]. Table 1 summarizes the device characteristics in our experimental setup; access latency varies by orders of magnitude and bandwidth discrepancies reach 12×.

Persistent memory, apart from its superior latency and read bandwidth, is also different in terms of access, being connected to the system via the memory bus, like DRAM, and accessible via CPU `load`/`store` instructions. This groundbreaking access path blurs the decades-old distinction between slow-but-persistent storage and fast-but-volatile memory. The direct access (DAX [14]) OS mechanism maps virtual pages from process address spaces directly to persistent memory physical locations.

Cold start execution under buffered and direct snapshot access. Figures 1c and 1d depict cold start execution when snapshots are restored from disk and persistent memory, respectively. As discussed earlier, first-touch memory accesses on microVM's physical memory result in VM-exits due to Extended Page Table (EPT) violations, and page faults.

With traditional storage (e.g., SSDs), faults entail copying (buffering) the memory-mapped snapshot file pages into DRAM (page cache) and then setting microVM's EPT to point at the new DRAM copies (Figure 1c). Thus, all microVM accesses to its physical memory eventually refer to DRAM locations, but all the initialization faults are subject to the cost of storage-to-DRAM data movement (I/O). This is a dominant overhead in cold start latency, and is sensitive to the underlying storage characteristics (latency and bandwidth).

With persistent memory and direct access, read faults entail only the costs of populating EPT and guest page tables on VM-exits, to point directly to DCPM locations (Figure 1d). Overheads from storage-to-DRAM transfers of whole pages are eliminated for those accesses, and parts of the guest physical memory end up directly backed by persistent memory. The trade-off is that subsequent read accesses to these locations are always served by the device (Figure 1b), suffering higher latency and lower bandwidth compared to DRAM. Write faults still transfer the whole page into DRAM, leveraging Copy-on-Write semantics over the privately mapped snapshot file (to preserve its content). In this paper, we study how sensitive serverless workloads are to these trade-offs.

3 METHODOLOGY

3.1 Evaluation Platform

We conduct our experiments on a single host (similar to [54, 17]), equipped with an Intel Xeon Gold 5812T Cascade Lake CPU with 2×16 physical cores, with frequency fixed at 2.7 GHz and SMT disabled. Each socket is equipped with 160GB DRAM and a 128GB Intel Optane DCPMM in AppDirect mode, while the host has also a 200GB Intel Optane DC P4801X Series SSD over PCIe 3.0 and a 240GB Intel SSD 520 Series over SATA3. Hereinafter, we refer to them as DRAM, DCPM, NVMe and SSD, respectively. Disks are formatted with the ext4 filesystem and we use ext4-DAX for DCPM.

We use Linux 5.1 on host, Firecracker v1.0.0 as VMM, and Linux 5.14 for guest microVMs, minimally configured, similar to the kernels provided by the Firecracker developers. In both host and guest kernels, THP [15] is set to `always`. In general, we adhere to the AWS Firecracker team's production guidelines [6], but we use Firecracker without the Jailer process.

Software stack adjustments. On EPT violation, by default, KVM sets the write permission bit on EPT if the page is writable, regardless of whether the fault that triggered the VM-exit was read or write. This eliminates the need for subsequent VM-exits to update the bit in cases of a WaR (Write-after-Read) page access pattern. However, it also eliminates the core advantage of snapshotting over DCPM: zero copies of read-only pages. To prevent Copy-on-Write operations on every DCPM snapshot access, we modify the host's kernel (only for DCPM experiments) to fall back to what the native page fault handler does: we do not set the write permission bit on EPT violations due to reads. This penalizes workloads

that perform WaR accesses (by suffering more VM-exits) but enables the benefit of zero copying.

In Firecracker, we disable the page cache readahead functionality, as it severely harms scaling microVM instances to many cores when the instances are restored from snapshots stored on NVMe or SSD, by saturating their bandwidth (§4.2). We use `madvise` and the `MADV_RANDOM` option for the memory mapping of the guest physical memory file.

3.2 Experimental Process

3.2.1 **Common Design**. All our experiments include one or more Firecracker instances waiting to restore a microVM with one vCPU and 512MB memory. The microVMs are always pinned to the physical cores of the NUMA node local to the DCPM module, and snapshot files are stored on the device under examination. MicroVMs' disk images are read-only mounted on the guests, and served from a DRAM-backed filesystem on the host, which allows us to better isolate the performance of snapshotting itself across the devices, without burdening them with guests' local I/O.

To accommodate functions' input and output data, we deploy a single-node MinIO [27] server (acting as a S3-compatible store) on the same physical host as the microVMs, but on a separate NUMA node, serving objects from a DRAM-backed filesystem as well. Thus, we take into account any software overheads on the critical path (e.g., the network stack), without adding obtrusive noise from data transfers over the wire to our measurements. The latter is a topic that merits its own interest with respect to analysis and optimization [53, 58, 56, 44, 52].

Finally, a multithreaded client uses Firecracker's API to communicate to the VMM processes which snapshot file should be used to restore a microVM. Once a microVM is restored and resumed, client threads can issue requests. In all experiments, after initiating a connection to the server inside their assigned microVM, client threads issue two identical consecutive requests, allowing the evaluation of both cold and warm function invocations. The client runs on the host, on a different NUMA node than the VMM processes to avoid interference. Snapshot files are removed from the page cache between successive runs, to closely model cold function invocations in real FaaS environments.

3.2.2 **Workloads**. To model FaaS workloads, we employ functions from FunctionBench [37, 38], a representative serverless benchmark suite. Table 2 presents the adopted functions. We use gRPC [22] as a communication fabric between function servers and clients, and Protocol Buffers [21] as their messages' serialization format[1].

[1]Source code is available at https://github.com/cslab-ntua/fbpml-systor22.

Table 2: FaaS workloads adopted from FunctionBench.

Name	Description
`helloworld`	No actual workload; for reference
`lr_serving`	Logistic regression serving (`scikit`)
`pyaes`	Python AES encryption of a string
`chameleon`	HTML table rendering
`cnn_serving`	JPEG classification CNN (Tensorflow)
`image_rotate`	JPEG image manipulation
`json_serdes`	JSON serialization & deserialization
`matmul_fb`	Matrix multiplication (`numpy`)
`video_processing`	Gray-scale effect application (`opencv`)
`lr_training`	Logistic regression training (`scikit`)

3.2.3 **Metrics**. There are multiple ways to measure cold start latency [17, 54, 51]. Our approach mostly resembles that of SnapFaaS [51]; total cold start latency measures from the instant the client contacts the pre-forked VMM process to restore the microVM, until the client receives a response from the invoked function inside it. We run every experiment 10 times and report the average latency.

We break total latency down to the following: (i) *VMM loading* that includes the time between the point that the client attempts to contact the VMM to restore the microVM until the point it receives a response that the microVM has been resumed; (ii) *function execution* as the total duration of the handling of the incoming request as perceived (and measured) by the function itself, and (iii) *client overhead* that includes the time between the point that the client attempts to contact the gRPC server inside the microVM, until the point the client receives a response from the function, minus *function execution*. Client overhead includes the time required to restore the gRPC connection with the server, the (intra-node) network overhead, and the time required to (de)serialize the request and the response (which are both minimal for all our functions).

4 EXPERIMENTAL RESULTS

4.1 Single microVM Experiments

We use a single client thread to evaluate the cold start latency of a single microVM booted from a snapshot stored in one of the three available devices: DCPM, NVMe or SSD. Our purpose is to study the performance of each function in isolation across all different devices, and deduce some first fundamental characteristics.

Function classification. Figure 2 illustrates the average latency of both cold and warm invocations of the functions adopted from FunctionBench, along with their breakdowns, when a single microVM is running. We classify the functions into three categories, shown in Table 3, based on their observed behavior in the case of warm requests. *Light* functions'

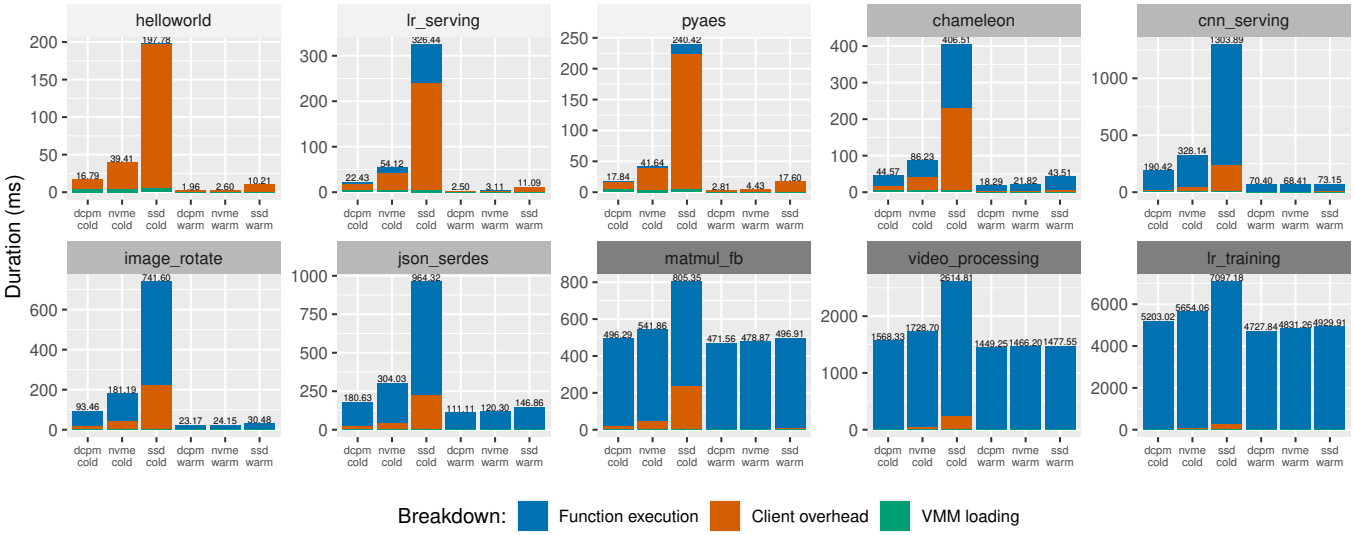


Figure 2: Average latency of cold start and warm function invocations, along with their breakdowns, when a single MicroVM is running. (Note that vertical scales vary per function.)

Table 3: Categorization of functions.

Class	Functions
Light	`lr_serving`, `pyaes` (and `helloworld`)
Mid	`chameleon`, `cnn_serving`, `image_rotate`, `json_serdes`
Heavy	`matmul_fb`, `video_processing`, `lr_training`

requests are handled very quickly by warmed-up instances and their workload is minimal. In contrast, *Heavy* functions' latency is noticeable, regardless of the cold start. *Mid* functions stand in the middle; they are handled fast, but their workload is not trivial.

4.1.1 **Cold start execution**. First, we focus our analysis on cold start performance through pairwise comparisons of devices in adjacent layers of the storage hierarchy. As discussed in §2, cold start invocations suffer from page faults, whose latency is affected by the medium's characteristics, i.e., its latency and whether the access is direct or buffered.

SSD vs NVMe. The difference between the two devices, across all functions, is evident in Figure 2. In particular, *Light* functions experience a 5–6× average slowdown in total latency when the snapshot is stored on the SSD rather than the NVMe. The slowdown is significant for *Mid* functions too, about 3.2–4.7×. *Heavy* functions appear to be the least affected, featuring 25%–51% average slowdowns.

To investigate these slowdowns, we introduce Figure 3, which aggregates per device the delay of each individual constituent of the average total latency of every function.

Starting with *VMM loading*, we observe that the measured delays form disjoint clusters, ranging a few hundreds of

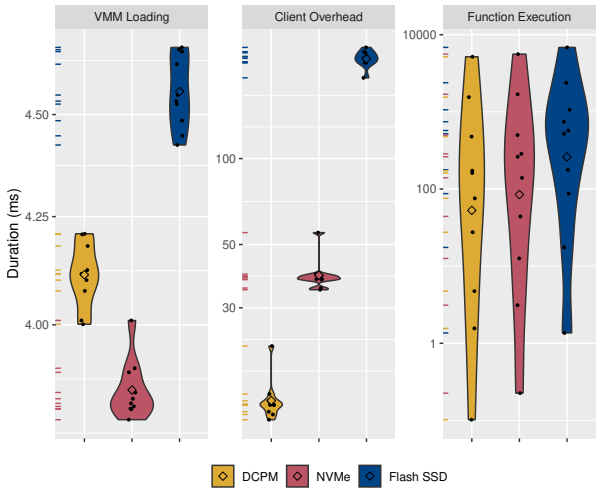


Figure 3: Individual constituents of total latency in cold start invocations, per device, across all functions. (Note that vertical scales vary per constituent.)

μs on the axis, which indicates that *VMM loading* depends on the underlying medium where the snapshot is stored. Indeed, it entails reading a small 12KB snapshot file where microVM state is stored, as well as reading file metadata (using open) to create a private memory mapping for guest physical memory. Nevertheless, the contribution of *VMM loading* in total function latency is negligible overall.

In the same figure we observe that, similarly, *client overhead* depends strongly on the underlying device. By its definition, elucidated in §3, *client overhead* encompasses a number of page faults and EPT violation VM-exits that need to be served. These refer either to pages containing code – both

kernel (e.g., the network stack) and userspace (e.g., CPython and the necessary modules) – or to fresh allocations which also need to touch guest physical frames lying on the device. As a result, the values in Figure 3 form disjoint clusters for both SSD and NVMe, which render the superiority of the latter visible. *Client overhead* delays for functions booted from snapshots stored on SSD exhibit a significant slowdown of 4.5−6.3× compared to the corresponding on NVMe.

When it comes to *function execution* delays, even though Figure 3 hints some differences across the storage hierarchy, it is otherwise rather inconclusive. Therefore, the total latency of a function is highly dependent on the nature of its actual workload. Figure 2 illustrates that there is always an improvement in *function execution* delays when using NVMe instead of SSD. For the particular constituent, *Light* functions on SSD suffer a significant 5.6−6.9× slowdown on average compared to NVMe. *Function execution* times' slowdown is decreased for *Mid* functions to 2.8–4×, whereas it is 15%–41% for *Heavy* functions.

Takeaway 1: Employing faster block devices (e.g., Optane NVMe) to store snapshots can be enough to speedup cold starts up to 6× compared to traditional flash-based SSDs.

NVMe vs DCPM. The differences between those two devices are generally modest in comparison with the differences between NVMe and SSD. In particular, *Light* functions experience on average a 2.3−2.4× slowdown in total latency when the snapshot is stored on NVMe rather than on DCPM. The slowdown for *Mid* functions is significantly lower, about 68%–94%. *Heavy* functions are the least affected, with a mere 9%−10% slowdown in total latency when NVMe is employed in place of DCPM.

The reason for the improved performance of DCPM over NVMe is twofold. First, DCPM is a device with lower access latency. Second, there is a difference on the amount of data transferred from each medium, as DCPM features direct rather than buffered access. In the case of DCPM, only pages that are written are copied to DRAM, via Copy-on-Write faults that keep the snapshot file intact. Read-only pages' content, on the other hand, is never copied to DRAM; instead it is accessed via `load` instructions directly on persistent memory (§2). As a result, it is the ratio of the read-only and writable pages, on each function's working set, that dictates the amount of data that will be copied to DRAM. In contrast, when first accessing a page stored on NVMe, regardless of whether it is a read or a write access, an additional transfer of that page into the page cache is always imposed.

To investigate this aspect, we examine the percentage of read-only and writable pages of the working sets of the microVMs, as derived by monitoring EPT violation VM-exits on the host, and compile Table 4. We observe that for *Light* functions, the majority of page accesses are read-only, which indicates that a larger portion of data remains in DCPM rather than being moved into DRAM. This, in turn, could explain why *Light* functions exhibit the biggest improvement from booting over DCPM rather than NVMe. The situation is analogous for *Mid* and *Heavy* functions, whose gains drop along with their percentage of read-only pages faulted-in.

Table 4: Function instances' (i) working set (WS) at 4KB page granularity, (ii) read-only/read-write accesses % to WS pages, (iii) total reads served by DCPM (in MB). The latter does not match WS, as read-only data are always fetched from the medium (DAX) and at smaller granularities than 4KB.

Function	WS (MB) [dstat]	Read-Only %	Read-Write %	Total reads served from DCPM (MB) [pmwatch]
helloworld	9.5	70	30	7.7
lr_serving	13	68	32	11
pyaes	10	67	33	8.7
chameleon	16	60	40	15
cnn_serving	49	44	56	76
image_rotate	32	44	56	34
json_serdes	45	32	68	50
matmul_fb	18	44	56	18
video_processing	51	43	57	59
lr_training	101	23	77	185

To further understand the total latency breakdown, we consult Figure 3. *VMM loading* is slightly costlier for DCPM than NVMe, but their difference lies in the range of a few hundreds of μs at most.

For *client overhead* delays, we observe that performance over NVMe is consistently worse than over DCPM. Functions booting from snapshots stored on the former exhibit on average a 2.5−3× slowdown in *client overhead* delays compared to those using the latter.

The same pattern recurs in the case of the *function execution* component of the latency breakdown, which can be better studied in Figure 2, since Figure 3 merely suggests its strong dependence on the actual workload of the function. We observe that *function execution* time is usually slightly improved in the case of DCPM. *Light* functions, whose working sets consist mostly of read-only pages, when booted from snapshots stored on NVMe experience a 2−2.7× slowdown for this constituent. *Mid* functions' slowdown in *function execution* time when using NVMe over DCPM is 60%−85%. The slowdown drops to a mere 4%−9% for *Heavy* functions, which comprise the least affected category from changing the snapshot storage medium from a DCPM to a NVMe device, and also the category with the highest percentage of writable pages, according to Table 4.

Table 5: Cold start execution slowdown (vs warm start) per function class and per storage medium.

Class	DCPM	NVMe	SSD
Light	6.3−9×	9.4−17.4×	19.4−29.4×
Mid	1.6−4×	2.5−7.5×	6.6−17.8×
Heavy	1.05−1.1×	1.13−1.18×	1.44−1.77×

Takeaway 2: Cold start invocation of functions that mainly perform read-only memory accesses and run shortly (e.g., < 100ms) can benefit by up to 2.4× by DCPM versus the Optane NVMe. If a function runs for longer periods and is CPU-bound or requires write access to the majority of its working set, the gap between the two mediums becomes negligible.

An important implication of employing a directly accessed device, like DCPM, as a snapshot store is that precious DRAM resources that would otherwise be occupied by the guest's working set, are not required anymore, and are free to be used elsewhere. In a nutshell, the read-only percentage, as reported in Table 4, represents also the reduction in occupied DRAM resources with respect to NVMe (or any buffered access device). The cost for that is that subsequent invocations which could enjoy the benefits of a warmed-up instance, are now paying the same cost of accessing the higher-latency medium. We study this effect in the next section.

This observation introduces a fundamental trade-off. On the one hand, DRAM is a low latency medium compared to DCPM, thus boosting the performance of serverless workloads that remain warm. On the other hand, DRAM's cost per GB is significantly higher than that of DCPM, constraining its capacity and rendering other layers of the storage hierarchy an important part of resource allocation policies.

Takeaway 3: Using DCPM as the snapshot store can save up to ∼70% of DRAM resources for a running function compared to an SSD.

4.1.2 **Cold start vs warm execution**. Next we study the cold start penalty with respect to the average warm execution of each function class and per device where snapshots are stored. Table 5 summarizes our results. Cold start invocations are up to 29.4× times slower compared to the respective warm ones when snapshots are stored on the SSD, whereas the slowdown drops to 17.4× for NVMe and 9× for DCPM.

This effect depends heavily on the function class. *Light* functions suffer the biggest slowdowns as their minimal workload is severely impeded by restoring the microVM and reestablishing the gRPC connection. On the contrary, *Heavy* functions' average latency is affected the least: in the case of SSD, cold start invocations still exhibit slowdowns of up to 77%, while for NVMe the overhead drops to 18% and for DCPM to a mere 10%.

Takeaway 4: High-performance storage mediums (e.g., Optane NVMe and DCPM) lead to cold start performance close to warm (e.g., only 9% slower), especially for functions with significant execution times (e.g., > 500ms).

4.1.3 **Warm execution**. We focus now on the cases where an incoming request is handled by an already running and warm function instance. Here, a significant portion of the microVM's working set is already in DRAM, thus the number of page faults – and also the I/O to the snapshot store – is expected to be minimal, especially given that a large number of pages is reused across function invocations [54].

We seek to answer if DCPM penalizes warm execution (compared to the SSD and NVMe) as functions' memory accesses to read-only portions of their working set are still served by storage, as discussed earlier. However, this is not the behavior observed in Figure 2, where it is evident that functions usually perform better over DCPM than the rest of the devices even in the cases of warm executions.

We attribute this peculiarity to the fact that no working set has really been moved into DRAM in its whole. Nondeterminism permeates several aspects of warmed up instances, leading to page faults that incur additional I/O to the snapshot store. Given the high latency of SSD and NVMe compared to DCPM, sometimes this can be enough to turn the tables in favor of the latter. Nevertheless, as recurring invocations warrant a decreasing amount of I/O from the snapshot store, the total latency of warm invocations over SSD and NVMe is expected to eventually improve due to the page cache. To quantify our hypothesis, we additionally examine the 51^{st} consecutive warm invocation on each device. We confirm that page caching effects significantly narrow the gap; latency differences among the devices do not surpass 4%, albeit DCPM usually remains the fastest. We further verify our assumption by additionally conducting the experiments using a filesystem backed by DRAM as snapshot store. For all functions, the total latency measured is slightly, yet consistently, lower than that of the DCPM-backed snapshot store, too: the slowdown of the latter compared to the former reaches 24% for *Light* functions, and merely 4% for the rest of the functions.

Takeaway 5: Employing DCPM as the snapshot store does not affect the latency of warm function invocations when microVMs run in isolation.

4.2 Scalability Experiments

In this subsection, we evaluate how functions in microVMs restored from snapshots on various devices behave as the number of concurrent microVMs is increased. Multiple client

threads attempt to restore a number of microVMs from snapshots, in parallel. To simplify our analysis, in our experiments we always run the same function across all microVMs, so that the actual workload is uniform. A real-world scenario would include an orchestrator [9, 18] featuring some sort of queuing system to leverage fewer warmed-up instances, rather than spawning a new one per incoming request. For that reason, in our experiments, each microVM is otherwise treated as if it was a distinct cold function being invoked, and is therefore associated with its own distinct disk image and snapshot. We note that all microVMs are pinned to the 16 physical cores of the NUMA node that is local to the DCPM module, equally distributed among them when necessary. Therefore, concurrent function invocations beyond 16 instances share all CPU resources.

4.2.1 **Cold start execution.** Having multiple VMM processes attempting in parallel to restore several microVMs from snapshots, all stored on the same device, stresses the read bandwidth of the underlying medium. Figure 4 summarizes our cold start scalability results.

SSD vs NVMe. We observe that the total latency using the SSD is always visibly worse than when using the NVMe. For *Light* functions, the slowdown ranges between 5.2–8.2×. It remains stable until 4 microVMs in parallel, but always increases for more of them, peaking at 16 and dropping beyond that, as NVMe's performance gets also penalized. The situation is similar for *Mid* functions, with the slowdown ranging between 3.2–7.5×, being relatively stable until 2 or 4 concurrent microVMs, peaking at 16 and dropping afterwards. Based on these findings, we verify that SSD's bandwidth for random reads (IOPS limit) is saturated when 4 instances boot from snapshots in parallel. In Figure 5, we plot one function per class employing linear scale on the vertical axis, to better illustrate the different concurrency level at which bandwidth's curve becomes apparent for the devices. *Heavy* functions are affected significantly less, with a slowdown of 1.3–2×, peaking at 16 microVMs as well.

Takeaway 6: Read bandwidth can severely affect the scalability of cold function invocations [54]; e.g., scaling stops already at 4 cores for SSD. A medium with superior bandwidth (e.g., NVMe) enables scaling to 8-16 concurrent cold start invocations each running up to 7.5× faster.

NVMe vs DCPM. Overall, we observe that the performance of the two devices is close. *Light* functions exhibit a slowdown of 2.1–2.8× when using NVMe in place of DCPM, which peaks at 4 and 8 concurrent microVMs. *Mid* scaling is similar but with a slowdown ranging from zero (only in `image_rotate` with large concurrency) up to 2×. *Heavy* functions experience the smallest slowdown when using NVMe instead of DCPM: 1%–97%.

For NVMe, *Light* and *Mid* functions appear to saturate its bandwidth (2 GB/s) somewhere between 8 and 16 instances. *Heavy* functions do not appear to be I/O-bound and their performance drops, commonly beyond 16 instances, due to CPU sharing. For DCPM, scaling follows the same trend as with NVMe; performance commonly drops beyond 8 cores, failing to scale perfectly to 16 cores, and severely deteriorates for >16 parallel function instances. This is not a straightforward result, as DCPM holds a significant advantage over NVMe with respect to read bandwidth (6.8 GB/s vs 2.2 GB/s – Table 1). We identify two reasons for this behavior.

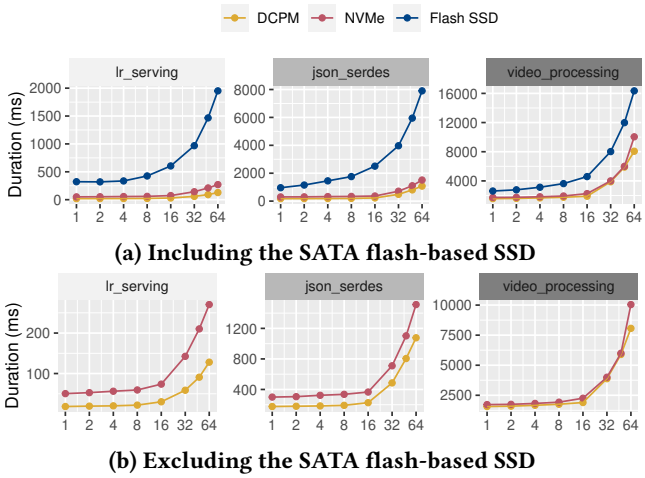
First, for NVMe, recurring page accesses during function execution are served by DRAM (page cache). This is not true for DCPM, where read-only pages remain on the medium (direct access) and recurring accesses always pay its higher latency and consume its bandwidth. Table 4 reports the total reads served by DCPM, measured in approximation using `pmwatch` [28]. In comparison to the data fetched from NVMe (equal to WS size – `dstat` [59] measurements), we observe that certain workloads (e.g., `cnn_serving`) read more bytes from DCPM during function invocation, increasing their bandwidth requirements. Others (e.g., `lr_serving`) fetch fewer data as they probably access sparsely their WS pages.

Taking into consideration the above, we observe that *Light* functions already saturate DCPM's bandwidth at 16 or more concurrent microVMs, as the throughput they collectively extract[2] is between 5–6GB/s. For *Mid* functions, where the observed scaling behaviour is similar, the maximum throughput (saturation point) observed is significantly lower, between 2.4–4.8GB/s. We discuss further this poor performance that limits DCPM scalability over NVMe.

Second, in our case many accesses are expected to be concurrent, irregular and smaller (e.g., guest's pointer dereferences) than the commonly suggested 256B access size [61, 35] for which the nominal 6.8 GB/s DCPM bandwidth is reported [50]. Past studies [61, 35] have shown that the performance of non-interleaved (as in our setup – §3.1) DCPM is severely degraded when access size is minimal (e.g., cache line size), even at the measured optimal concurrency level. Moreover, they have shown that even at optimal access sizes, the performance is non-monotonic as concurrency varies due to increased irregularity in the access pattern [61, 35]. Consequently, we find that our results corroborate with past studies as our use case exhibits a combination of characteristics known for degrading non-interleaved DCPM setups' performance under scale. To verify that the limited throughput observed can be attributed to the aforementioned characteristics, we devise a microbenchmark. A variable number of

[2]We calculate an approximation of the collectively extracted throughput by dividing the amount of data transferred from the DCPM (or the working set read from the NVMe) for each microVM with the concurrent function invocations' average response latency.

Figure 4: Average cold start latency as the number of microVMs restored from snapshots in parallel increases. (Note that vertical and horizontal scales are log_{10} and log_2, respectively, with the former being different per function.)



(a) Including the SATA flash-based SSD

(b) Excluding the SATA flash-based SSD

Figure 5: Average cold start latency as the number of microVMs restored from snapshots in parallel increases. (Note that the scale on the vertical axis is linear and varies per plot.)

threads read from a (separate each) 16MB file on each device. Assuming a 1:1 read-write access ratio, to approximate a microVM's access pattern (Table 4), we randomly copy 4KB blocks of the first half of the file into DRAM (to emulate CoW accesses), and then randomly read 64B blocks from the other half of the file to emulate irregular read-only accesses. We measure throughput similar to the one observed in our real-world experiments: peaking at 4 GB/s for 8-16 threads on DCPM, and tailing off as concurrency further increases and NVMe's throughput catches up (an analogous result is reported in [60]).

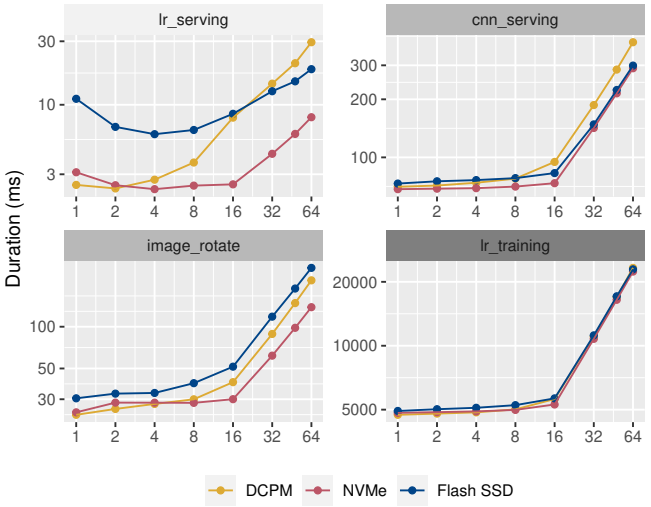


Figure 6: Average "warm" latency as the number of concurrent microVMs increases. (Note that vertical and horizontal scales are log_{10} and log_2, respectively, with the former being different per function.)

Takeaway 7: Inherent characteristics of snapshot restoration at scale (e.g., irregular pattern due to high concurrency and small access granularity) are not aligned to the best programming practices for persistent memory [61].

A setup consisting of multiple interleaved DCPM modules could, therefore, be considered as an option to compensate for the throughput loss due to the characteristics of our use case, as it can greatly improve random small read bandwidth (e.g., $\sim 2 \rightarrow 10$ GB/s for 64B accesses, as reported in [61, 35]).

4.2.2 **Warm execution**. Figure 6 shows the scalability of warm-start function invocations for a representative subset of the workloads, at least one per *class*.

SSD vs NVMe. We observe that the average latency for *Light* and *Mid* functions in instances restored from SSD is always worse than when restored from NVMe, even though the working set of the running functions is expected to reside mostly in DRAM for both cases. The maximum slowdown reaches 4×, observed in *Light* functions. This is attributed to additional page faults, even in warm runs, that incur additional I/O to the snapshot file. This effect has been already observed in §4.1 and here is magnified by concurrency, which also stresses bandwidth limits.

Takeaway 8: Storage hierarchy is a decisive factor when snapshotting techniques are put into use, as it can affect the latency even of warmed-up instances due to spontaneous page faults (hence I/O to the underlying medium).

NVMe vs DCPM. We generally observe that, warmed-up microVMs restored from DCPM rarely preserve their performance superiority at scale. The consistent performance degradation of DCPM compared to NVMe, exacerbated for function classes with higher percentage of read-only accesses (as per Table 4), constitutes a manifestation of the higher bandwidth of DRAM (page cache) compared to DCPM. Interestingly though, even in our limited single-DIMM DCPM setup (Takeaway 7) not all workloads are penalized. *Light* functions are affected the most, exhibiting a slowdown of 1.1−3.7× (beyond 2 concurrent instances). Instead, the performance of *Mid* functions fluctuates by a mere ±35% at most across the media, with NVMe being faster in most cases, but their performance is generally very close. *Heavy* functions perform equally for all mediums.

In light of the above, we observe that the greater the benefits a function class exhibits from running over DCPM on cold start invocations, the more is penalized during warm executions that scale to many cores.[3]

Takeaway 9: DCPM's limited bandwidth (compared to DRAM) may penalize warm execution scalability, especially in limited single-DIMM DCPM setups (Takeaway7).

4.3 Disk Readahead

Disk prefetching is enabled by default on Linux, so that when a file page is read from the disk into the page cache, a number of logically successive ones are fetched along with it (readahead faults). In all our previous experiments we have turned page cache readahead off.

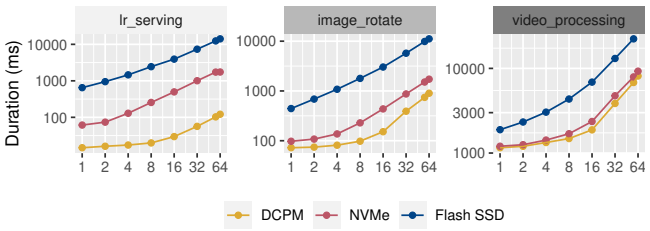


Figure 7: Cold start scalability with readahead enabled.

Figure 7 shows the average measured cold start latency for a subset of the examined functions (one per class) when readahead is enabled. Note that readahead is only relevant to NVMe and SSD that have buffered access. Juxtaposition with Figure 4 unveils that sometimes prefetching is advantageous for 1 or 2 concurrent instances (also depending on the actual workload), but severely penalizes performance scaling. Disk bandwidth is quickly saturated by fetching blocks that are never accessed. *Light* functions, that are already BW-limited, are affected the most, experiencing average slowdowns skyrocketing to an order of magnitude higher, or more.

Prior work [54] has raised the issue of the potentially harming effects of generic page cache prefetching in FaaS, identifying that cold starts tend to have irregular access patterns and cannot benefit from it. We augment on this observation studying its major bandwidth saturation effect.

Takeaway 10: Disk prefetching can needlessly saturate storage bandwidth and reduce the performance.

5 MAIN TAKEAWAYS AND DISCUSSION

In this section, we recap what we consider as the major takeaways of our analysis and seek to answer what opportunities arise with respect to better resource management for FaaS.

Cold start performance can vary by up to 8.5×, depending on which device the snapshots are stored. As latencies and bandwidth can vary by orders of magnitude on a modern storage stack, integrating snapshot placement on the decision making mechanisms of a FaaS orchestrator can affect performance. For example, real-world traces from cloud providers [47] indicate that 20% of applications are responsible for >99% of invocations. Placing popular snapshots on high-performant devices could improve overall system performance and scalability (bandwidth importance §4.2).

Functions' execution overhead and access patterns change the performance gains obtained by devices. While placing snapshots to faster mediums (e.g., Optane DCPM or NVMe vs flash SSD) is always beneficial; the benefit among devices on the higher ranks of a modern storage

[3]Note that the maximum number of concurrent requests to DCPM is dictated by the number of physical cores, and not the number of running functions (which can be thousands).Nevertheless, the number of running functions can affect the access pattern (irregularity) of the requests to DCPM, which in turn can affect the bandwidth that the medium provides [60, 61].

stack is not linear. We observe that DCPM is beneficial (over NVMe) for functions with mainly read-only access on their working sets. Direct storage access eliminates data movement to DRAM, providing ~ 2× performance benefit. At the same time we observe that this performance boost is relevant mainly for functions with execution times ~<100ms (*Light*), which comprise ~20% of function invocations [47], and are highly sensitive to initialization costs.

Fast mediums bring cold start performance close to warm for a category of workloads. Past studies [54, 51] consider cold start latency a priori prohibitive in terms of performance, assuming storage access latencies in the order of 100s of μs (flash). We find that fast devices ($\leq$ 10s of μs) can bring cold start execution performance of *heavy* functions (running for >500ms, >50% of invocations [47]), very close to warm (1.1−1.18× in our experiments). We consider this as a new decisive factor for the "keep-alive" policy applied by an orchestrator [47]. If cold starts are entirely cheap, idle function instances can be shutdown immediately, releasing precious DRAM resources.

Utilizing DCPM as snapshot store presents various trade-offs. Backing part of virtual machines' physical address spaces (read-only) with persistent rather than volatile memory, means higher latency and lower bandwidth during function execution. This may penalize warm performance for some functions. Thus, (i) explicit caching of portions of their working sets in DRAM (similar to [54]) could be worth exploring, or (ii) DCPM should be chosen only for such functions with infrequent invocation times (which are anyway likely subject to cold start latency). Meanwhile, DCPM snapshot stores can lead to 70% less utilization of DRAM resources.

6 RELATED WORK

Cold start & Snapshotting. FaaS cold start delays have been modeled and studied by prior work [58, 47, 17, 54, 51, 53]. Some works also propose new techniques in an attempt to mitigate cold starts [57, 17, 54, 51]. In particular, REAP [54] studies extensively the detrimental cost of demand paging on function invocations and proposes an optimized method to only prefetch the working set. SnapFaas [51] identifies fundamental overheads entailed by microVM snapshot restoration to propose a new technique, arguing that alternative designs would still be bound by the same overheads, unless they break the FaaS abstraction. All these works are robust and thorough, but assume traditional storage (i.e., flash SSDs) to optimize snapshotting techniques and to evaluate them. We argue that recent technological advancements render storage a distinct factor that needs to be taken into account for further tuning FaaS environments in the modern datacenter.

Modern Storage Hierarchy. Past work [60] has studied the relative performance among layers of the modern storage hierarchy, and extensively studied caching under the new prism. It proposes non-hierarchical caching and provides the novel Orthus implementations to demonstrate its efficacy. Our study focuses on the specific case of snapshotting in the context of FaaS and motivates the inclusion of storage hierarchy as a factor to be considered in resource management.

Persistent Memory. Past work has extensively studied the performance of DCPM under different configurations (e.g., interleaving, concurrency, etc) [61, 35] or under specific workloads [19], providing useful hints and suggested patterns for programming over the medium [61]. Our study points out how microVM snapshotting over DCPM breaks some of those rules, and corroborates past results showing their effect on cold and warm function invocations.

Serverless and Persistent Memory. To our knowledge, there is only one study [55] involving FaaS over DCPM. However, that work focuses on container-based FaaS, without studying snapshotting, and it lacks our thorough methodology to identify and quantify the impact of the medium in the context of FaaS.

7 CONCLUSIONS

In this paper we analyze the performance of cold start and warm executions in FaaS environments, placing snapshots across different devices of a modern storage hierarchy. We characterize the behavior of representative workloads, examine fundamental trade-offs and discuss how these can be leveraged for efficient snapshot placement. Based on these, we plan to enrich orchestrators' decision making regarding snapshot placement and function scheduling, targeting better system performance overall.

ACKNOWLEDGEMENTS

We are grateful to our shepherd, Eran Gilad, for helping shape the final version of this paper. We also thank the anonymous reviewers for their valuable feedback and suggestions that significantly improved this paper, as well as the members of the Computing Systems Laboratory at National Technical University of Athens for the fruitful discussions and comments on the work.

REFERENCES

[1] Apache Software Foundation (ASF). [n. d.] Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[2] 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)*. USENIX Association, Santa Clara, CA, (Feb. 2020), 419–434. ISBN: 978-1-939133-13-7. https://www.usenix.org/conference/nsdi20/presentation/agache.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association,

Boston, MA, (July 2018), 923–935. ISBN: 978-1-939133-01-4. https://www.usenix.org/conference/atc18/presentation/akkus.

[4] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. 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). Association for Computing Machinery, Lausanne, Switzerland, 101–113. ISBN: 9781450375542. DOI: 10.1145/3381052.3381315.

[5] The Cloud Hypervisor Authors. [n. d.] Cloud Hypervisor – Run Cloud Virtual Machines Securely and Efficiently. Linux Foundation. https://www.cloudhypervisor.org/.

[6] The Firecracker Authors. [n. d.] Firecracker – Production Host Setup Recommendations. https://github.com/firecracker-microvm/firecracker/blob/main/docs/prod-host-setup.md.

[7] The Firecracker Authors. [n. d.] Firecracker Snapshotting. https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md.

[8] The Kata Containers Authors. [n. d.] Kata Containers – Open Source Container Runtime Software. https://katacontainers.io/.

[9] The Knative Authors. [n. d.] Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. https://knative.dev/.

[10] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM*, 60, 4, (Mar. 2017), 48–54. DOI: 10.1145/3015146.

[11] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA, (Apr. 2005). https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[12] Bitnami/VMware. [n. d.] Kubeless. https://github.com/vmware-archive/kubeless.

[13] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (EuroSys '20). Association for Computing Machinery, Heraklion, Greece. ISBN: 9781450368827. DOI: 10.1145/3342195.3392698.

[14] Linux Kernel Documentation. [n. d.] *Direct Access for files*. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[15] Linux Kernel Documentation. [n. d.] *Transparent Hugepage Support*. https://www.kernel.org/doc/html/latest/vm/transhuge.html.

[16] Linux Kernel Documentation. [n. d.] *Universal TUN/TAP device driver*. https://docs.kernel.org/networking/tuntap.html.

[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*. Association for Computing Machinery, New York, NY, USA, 467–481. ISBN: 9781450371025. https://doi.org/10.1145/3373376.3378512.

[18] Alex Ellis and OpenFaaS Ltd. [n. d.] OpenFaaS – Serverless Functions, Made Simple. https://www.openfaas.com/.

[19] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *Proc. VLDB Endow.*, 13, 8, (Apr. 2020), 1304–1318. DOI: 10.14778/3389133.3389145.

[20] Google. [n. d.] gVisor: an application kernel for containers. https://gvisor.dev/.

[21] Google. [n. d.] Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers/.

[22] The gRPC Authors. [n. d.] gRPC – A high performance, open source universal RPC framework. https://grpc.io/.

[23] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. (2018). arXiv: 1812.03651 [cs.DC].

[24] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO, (June 2016). https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[25] Amazon Web Services Inc. [n. d.] AWS Fargate – Serverless compute for containers. Amazon Web Services Inc. https://aws.amazon.com/fargate/.

[26] Amazon Web Services Inc. [n. d.] AWS Lambda – Run code without thinking about servers or clusters. Amazon Web Services Inc. https://aws.amazon.com/lambda/.

[27] MinIO Inc. [n. d.] High Performance, Kubernetes Native Object Storage. https://github.com/minio/minio.

[28] Intel. [n. d.] `intel-pmwatch`. Intel. https://github.com/intel/intel-pmwatch.

[29] Intel. [n. d.] Intel® Optane™ Persistent Memory 128GB Module. https://ark.intel.com/content/www/us/en/ark/products/190348/intel-optane-persistent-memory-128gb-module.html.

[30] Intel. [n. d.] Intel® Optane™ SSD DC P4801X Series. https://ark.intel.com/content/www/us/en/ark/products/149364/intel-optane-ssd-dc-p4801x-series-200gb-m-2-110mm-pcie-x4-3d-xpoint.html.

[31] Intel. [n. d.] Intel® SSD 520 Series. https://ark.intel.com/content/www/us/en/ark/products/66250/intel-ssd-520-series-240gb-2-5in-sata-6gbs-25nm-mlc.html.

[32] [n. d.] Intel® Optane™ DC SSD Series. https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds/optane-dc-ssd-series.html.

[33] [n. d.] Intel® Optane™ Persistent Memory – Memory Optimized for Data-Centric Workloads. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[34] [n. d.] Intel® SSD 520 Series. https://www.intel.com/content/www/us/en/products/sku/66250/intel-ssd-520-series-240gb-2-5in-sata-6gbs-25nm-mlc/specifications.html.

[35] Joseph Izraelevitz et al. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. (2019). arXiv: 1903.05714 [cs.DC].

[36] Eric Jonas et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. (2019). arXiv: 1902.03383 [cs.OS].

[37] 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)*, 502–504. DOI: 10.1109/CLOUD.2019.00091.

[38] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS. In *Proceedings of the ACM Symposium on Cloud Computing* (SoCC '19). Association for Computing Machinery, Santa Cruz, CA, USA, 477. ISBN: 9781450369732. DOI: 10.1145/3357223.3365439.

[39] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the*

Linux symposium number 8. Vol. 1. Dttawa, Dntorio, Canada, 225–230.

[40] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, (June 2014), 61–72. ISBN: 978-1-931971-10-2. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[41] James Larisch, James Mickens, and Eddie Kohler. 2018. Alto: Lightweight VMs Using Virtualization-Aware Managed Runtimes. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (ManLang '18). Association for Computing Machinery, Linz, Austria. ISBN: 9781450364249. DOI: 10.1145/3237009.3237022.

[42] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '13). Association for Computing Machinery, Houston, Texas, USA, 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167.

[43] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (SOSP '17). Association for Computing Machinery, Shanghai, China, 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763.

[44] 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)*. USENIX Association, Renton, WA, (July 2019). https://www.usenix.org/conference/hotcloud19/presentation/mohan.

[45] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 57–70. ISBN: 978-1-931971-44-7. https://www.usenix.org/conference/atc18/presentation/oakes.

[46] Rusty Russell. 2008. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42, 5, (July 2008), 95–103. DOI: 10.1145/1400097.1400108.

[47] Mohammad Shahrad et al. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, (July 2020), 205–218. ISBN: 978-1-939133-14-4. https://www.usenix.org/conference/atc20/presentation/shahrad.

[48] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '19). Association for Computing Machinery, Providence, RI, USA, 121–135. ISBN: 9781450362405. DOI: 10.1145/3297858.3304016.

[49] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, (July 2020), 419–433. ISBN: 978-1-939133-14-4. https://www.usenix.org/conference/atc20/presentation/shillaker.

[50] Intel Spec. [n. d.] Optane DC Persistent Memory Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.

[51] Yue Tan, David Liu, Nanqinqin Li, and Amit Levy. 2021. How Low Can You Go? Practical cold-start performance limits in FaaS. (2021). arXiv: 2109.13319 [cs.DC].

[52] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (SoCC '20). Association for Computing Machinery, Virtual Event, USA, 16–29. ISBN: 9781450381376. DOI: 10.1145/3419111.3421275.

[53] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. 2021. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 51–62. DOI: 10.1109/IISWC53511.2021.00016.

[54] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM. DOI: 10.1145/3445814.3446714.

[55] Ahmet Uyar, Selahattin Akkas, Jiayu Li, and Judy Fox. 2021. Intel Optane DCPMM and Serverless Computing. (2021). arXiv: 2109.11021 [cs.DC].

[56] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, (July 2021), 443–457. ISBN: 978-1-939133-23-6. https://www.usenix.org/conference/atc21/presentation/wang-ao.

[57] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019* (EuroSys '19). Association for Computing Machinery, Dresden, Germany. ISBN: 9781450362818. DOI: 10.1145/3302424.3303978.

[58] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 133–146. ISBN: ISBN 978-1-939133-01-4. https://www.usenix.org/conference/atc18/presentation/wang-liang.

[59] Dag Wieers. [n. d.] dstat. http://dag.wiee.rs/home-made/dstat/.

[60] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, (Feb. 2021), 307–323. ISBN: 978-1-939133-20-5. https://www.usenix.org/conference/fast21/presentation/wu-kan.

[61] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, (Feb. 2020), 169–182. ISBN: 978-1-939133-12-0. https://www.usenix.org/conference/fast20/presentation/yang.