TOSS: Tiering of Serverless Snapshots for Memory-Efficient Serverless Computing

Theodore Michailidis*, Juno Kim†, Linsong Guo*, Steven Swanson*, Jishen Zhao*
*University of California, San Diego
{tmichail, l1guo, sjswanson, jzhao}@ucsd.edu
†Oracle
{juno.k.kim}@oracle.com

***Abstract*—Serverless computing is an emerging cloud computing paradigm where users offload functions to serverless platforms that manage their own execution environments. Despite recent advancements on efficient serverless management, we find that cloud providers' solutions lead to unnecessary memory overheads, and significant memory cost by assuming a single tier of memory (DRAM). In this paper, we evaluate previous serverless works and offer insights about enabling memory tiering for serverless. Based on our insights, we introduce Tiering of Serverless Snapshots (TOSS), a heterogeneous memory mechanism that aims to reduce the total memory cost on serverless platforms, while achieving comparable performance to single-tier solutions. We show that TOSS achieves near optimal memory cost for most functions, while offloading on average 92% of the memory to the slow tier. In addition, TOSS achieves 52× lower setup time and up to 4.2× lower invocation time than the state-of-the-art DRAM-only mechanism.**

***Index Terms*—memory systems, serverless, tiered memory**

I. Introduction

Serverless computing [28], [34], [4], [10], [30], [9], [46] is a cloud computing paradigm that allows developers to focus on building and deploying applications on infrastructure that is solely managed by cloud providers. Developers only pay for the resources they use, while it is easier to scale applications, as the infrastructure adjusts automatically to meet demands. When a function invocation request is made, the system has to setup the function's environment. This means loading the code, library and data that are needed, namely the startup time.

In the serverless world, most invocations are short-running. A previous study [30] claimed that 50% of the functions run for less than 0.67 s, while 75% for less than 10 s, while another study [35] showed that the setup time takes between 160 ms and 3.6 s. Consequently, the startup time dominates the total invocation time for most functions, which is known as the *cold start* problem. This is a burden for cloud providers, since clients only pay for the invocation time and resources used during the function execution.

A few techniques have been proposed to accelerate the startup time, with checkpoint restoration using Virtual Machine (VM) snapshots [34], [4], [10] gaining a lot of popularity. The use of VM snapshots minimizes the startup latency and loads all required information and data to re-invoke functions. This has gained a lot of attraction from industry too, with Amazon recently introducing Lambda SnapStart [5].

We identify two key pathologies in previous snapshot-based approaches: One is the assumption that memory access patterns remain consistent across different invocations. Another is that modern serverless systems utilize a single tier of expensive memory (DRAM), which drives up costs, as memory contributes to 40% of total server expenses [1]. While memory tiering could help with the last issue, existing approaches are designed for long-running applications, contrary to short-running and randomly invoked serverless functions.

In this paper, we offer a memory analysis for serverless functions, identifying shortcomings from previous approaches. Based on our observations, we introduce **T**iering **o**f **S**erverless **S**napshots (TOSS), the first memory tiering mechanism for serverless functions. TOSS provides insights about different functions' memory patterns, aiming to reduce both the main memory usage and the serverless memory cost. In this paper, we make the following contributions:

- We identify shortcomings in previous snapshot-based serverless works, and provide insights for enabling memory tiering for serverless.
- We introduce TOSS, the first serverless memory tiering system that automatically minimizes memory cost.
- We present a simple memory cost formula that evaluates the memory cost of any memory tiering configuration.
- We add snapshot tiering support on Firecracker, a widely-used open-source Virtual Machine Monitor.

The remainder of the paper is organized as follows. Section II contains the associated background of our work. Section III includes our analysis of past approaches our motivation. Section IV includes a high-level view of TOSS' design, while Section V contains our implementation details. Section VI includes the evaluation of our work, in Section VII we describe the related work and Section VIII concludes our paper.

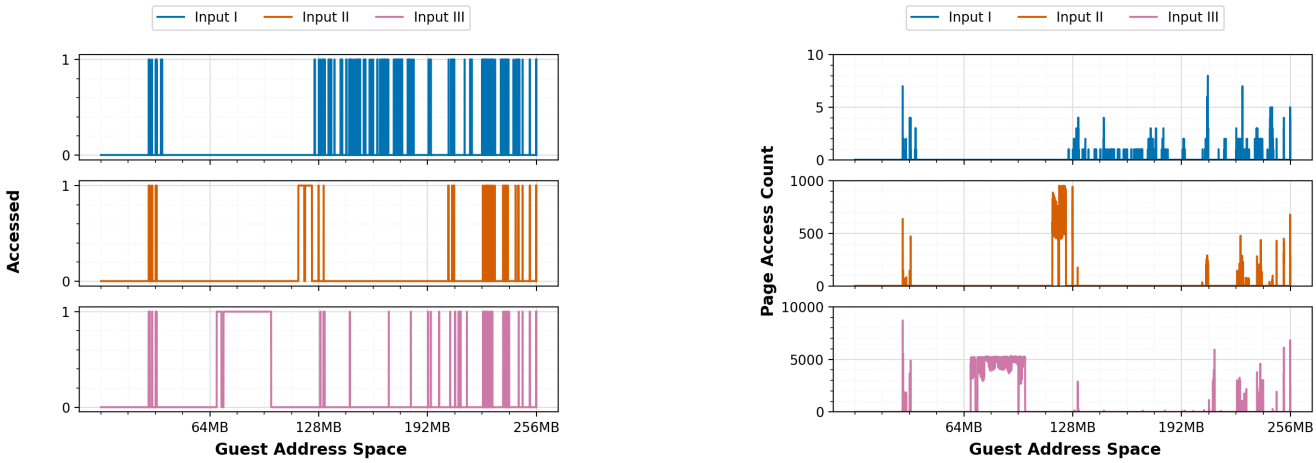
II. Background

Serverless computing offers a pay-as-you-go pricing model, with cloud providers solely managing the infrastructure, optimizing resource usage by automatically allocating resources based on demand. This section focuses on the associated background information.

A. Firecracker as Function Instance

Firecracker [1] is an open source micro-virtualization technology by AWS Lambda. It manages lightweight VMs (microVMs) that host function-based services with a focus on running short-lived, stateless functions in a scalable, secure and cost-effective manner.

†Work done while at UCSD.



(a) `userfaultfd()` **(b)** DAMON

Fig. 1: Visualized working set characterization from `userfaultfd()` and DAMON

Firecracker offers a snapshotting feature [5] to allow users to create, save, and restore microVMs, as a solution to the cold start problem. Snapshots are created using copy-on-write on the root filesystem, while they capture both the VM's and virtual machine monitor's states, including all emulated hardware devices. To restore a snapshot, Firecracker initially loads the VM state and memory maps the guest memory file, while guest memory pages are loaded on-demand.

B. Memory Tiering

Memory tiering techniques [27], [2], [13], [22], [16] leverage the heterogeneous characteristics of different memory tiers. For two memory tiers, we expect to have a smaller, expensive and fast tier (such as DRAM) and a denser, cheaper, inexpensive and slow tier (such as CXL-attached [38] or persistent memory [40]). The goal is to maintain the best performance and expand the total memory capacity, while minimizing the overhead. This is achieved by profiling the memory and dynamically placing hot and cold pages to the appropriate tier.

Traditional memory tiering techniques cannot be utilized for serverless functions, since they are designed for long-running applications. In contrast, serverless functions are short-lived, with execution times ranging from milliseconds to a few seconds [30], rendering memory profiling ineffectual. Additionally, serverless functions exhibit highly variable invocation patterns, from completely random to fixed invocations. These dynamic behaviors make it difficult for conventional tiering systems to effectively capture and optimize memory usage for serverless environments.

C. Memory Profiling

This section includes existing approaches on memory profiling for snapshotting and memory tiering.

Snapshot-based Memory Profiling. REAP [34] and FaaSnap [4] are snapshot-based serverless approaches that use the following profiling method: After the first function invocation, they capture a VM snapshot and identify the function's working set (WS), i.e. pages that have been accessed at least once during the first invocation. REAP uses `userfaultfd()` [20] and FaaSnap uses `mincore()` [19] for the working set characterization. For subsequent invocations, the WS is loaded on memory and the rest of the memory is loaded on demand from the disk.

PEBS. Recent memory tiering systems utilized Intel's Processor Event Based Sampling (PEBS) for memory profiling, an extension to Intel's processors original performance counters. PEBS captures hardware events (e.g. LLC misses) along with the associated memory addresses allowing for more sophisticated decision-making about page placement on different memory tiers.

DAMON. Another promising memory monitoring profiling tool is Data Access MONitor (DAMON) [29], which is designed to be accurate, light-weight and scalable. DAMON reduces the overhead by using adaptive region-based sampling, where memory is split between different-sized regions. Adjacent memory regions are periodically merged and split, based on the similarity between their access frequency.

DAMON is already used on real-world systems from companies like Amazon [17] and Alibaba [7]. In addition, Linux has recently added support for DAMON-based reclamation and huge page management [31]

D. Memory Pricing

Serverless computing platforms offer different memory pricing tiers, which are measured in $/unit of storage/unit of time (e.g. $/MB/ms). For instance, AWS Lambda charges the usage per 1 millisecond, while Cloud Functions per 100 milliseconds for fixed size memory configurations, i.e. 128 MB, 256 MB etc. While these prices refer to single-tier systems, modern heterogeneous systems consist of different memory technologies with varying costs per MB and attributes.

These vendors offer vCPU and memory bundles, which necessitates that clients find the best bundle that covers both

the memory and CPU requirements for their functions. For CPU-bound functions, clients choose bundles with significantly more memory, to avoid the slowdown due to insufficient CPU time. Consequently, clients have to choose higher cost tiers that lower the execution time and increase the cost per unit of time; choosing the best bundle can be cumbersome.

In addition, all this excess memory is a good use case for memory tiering, where we can offload unused memory to the slower tier and keep frequently used memory in the fast tier. Automating the process of finding the best memory configuration alleviates clients from profiling the function.

III. Analysis & Motivation

Despite recent progress on providing fast startup of VMs, recent works have failed to identify and resolve serverless functions' memory idiosyncrasies. At the same time, previous solutions work on a single tier of memory, which incurs high memory costs and limits memory scalability. While memory tiering is an active research area, it has not been explored in serverless computing environments. In addition, the literature is missing the means to evaluate such memory tiering solution for serverless functions.

Industry publications [18], [1] claim that DRAM dominates the total cost of servers, taking up to 40-50%. Our main motivation is to reduce the memory cost of serverless execution by utilizing a slower, cheaper tier. The cost reduction derives from replacing DRAM with a cheaper alternative (price per GiB), while maintaining an acceptable performance loss.

Therefore, the motivation of this work is threefold: (i) identify shortcomings in previous snapshot-based serverless works, (ii) provide insights about memory tiering for serverless, and (iii) propose a cost-efficient memory tiering approach for serverless. Last, we also introduce a memory cost formula that can be adopted by cloud vendors to evaluate tiered memory configurations and enhance their current memory pricing plans.

While we use for our analysis Intel® Optane™ Persistent Memory as the slow tier, we design TOSS to use with any memory technology as fast and slow tiers. For instance, TOSS can be utilized by using DDR5 as the fast tier and CXL-attached DDR4 as the slower, cheaper tier and adapting the memory cost formula. Additionally, despite Optane's discontinuation, the persistent memory research space remains relevant due to emerging alternatives like MRAM and ReRAM. Recent works from industry and academia are using these non-volatile memory technologies to accelerate high-performance computing and AI workloads, while CXL-attached persistent memory is explored for cloud and data center environments [41], [43], [42], [24], [33], [32], [21].

A. Memory Tiering for Serverless

Existing memory tiering techniques are ill-suited for short-running, arbitrarily invoked serverless applications. The contiguous profiling and costly page migrations between tiers can lead to suboptimal and unfair performance between serverless functions, while unnecessarily increasing the system overhead.

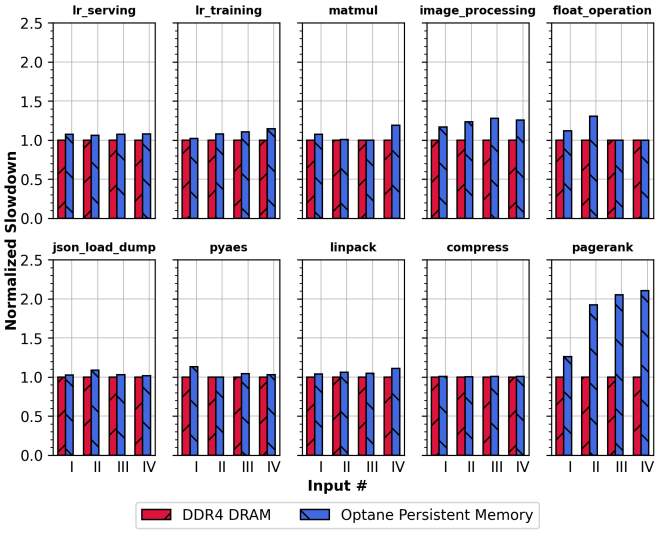


Fig. 2: Normalized slowdown when offloading functions fully on Intel® Optane™ Persistent Memory. We offload each function for every input depicted in Table I, and show normalized arithmetic mean over 10 iterations.

Instead, we propose to profile each function individually, which leads to optimal memory tiering for each function and significant profiling and system overhead reduction.

To enable memory tiering for serverless, we have to first identify how a slower tier can affect the functions' performance. To this end, we deploy the functions in Table I with their respective inputs to Intel® Optane™ Persistent Memory and show the slowdown normalized to the DRAM case in Figure 2.

`Observations #1 & #2`: **First**, some non-memory intensive and short-running functions can run entirely in the slower tier, with no or negligible performance degradation. **Second**, some functions display a varying slowdown between different inputs. This is due to the fact that the memory footprint varies widely between different inputs for these functions. In this case, some inputs still display negligible slowdown. Correctly identifying these cases and offloading them to the slower tier can lead to large memory cost savings.

B. Function Input Impact

Since REAP and FaaSnap capture a snapshot during the first function execution, all subsequent executions will use the same snapshot. However, divergent inputs lead to varying memory access patterns and working set files. In order to test how different inputs can affect the execution performance, we run REAP with different inputs for the first execution (*snapshot input*) and a subsequent invocation (*execution input*). We use the functions in Table I for every combination of snapshot and execution input, and show the mean and maximum invocation time in Figure 3. The invocation time includes the time it takes to setup the snapshot and execute the function. Each bar represents the mean execution time between different snapshot inputs, normalized to the case where the snapshot and

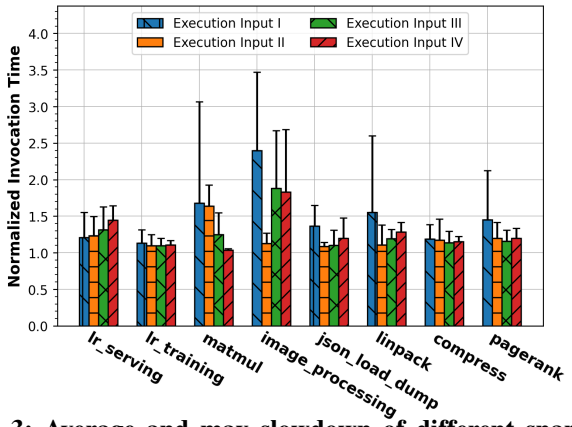


Fig. 3: Average and max slowdown of different snapshot inputs with REAP for each execution input. Normalized to using the same snapshot and execution input.

execution input is the same. The average execution slowdown over all cases is 26% and up to 3.47×. During our study we also found that invocations with the same input can lead to slightly different memory access patterns and execution times. We attribute this instability to the non-deterministic memory allocation in the guest OS.

Observation #3: The snapshot input can heavily affect the execution performance. Additionally, different executions with the same input can lead to different memory access patterns. To reduce functions' execution overhead, we need a representative snapshot by capturing the memory access pattern over multiple function invocations with divergent inputs.

C. Memory Access Profiling

We find two issues with previous approaches' dual-accessed memory profiling mechanism: (a) memory access patterns are not stable across invocations, and (b) the classification scheme is not nuanced enough. i.e. a single-accessed page is handled the same as one that has thousands of accesses. In addition, `userfaultfd()` and `mincore()` incur a high overhead, which limits REAP and FaaSnap to use them only during the initial function invocation. Additionally, `mincore()` inflates the memory working set by taking into account prefetched pages in the host page cache.

In another aspect, while PEBS is designed to be more accurate with less overhead, it still suffers from significant overheads [2], [3], including increased context switches, memory IO, and cache misses, and also leads to inconsistent results [44]. Previous works have achieved low overhead with PEBS only at a single-application granularity and by lowering its profiling frequency, which renders it unsuitable for profiling multiple short-running functions.

To avoid the aforementioned issues, we propose using DAMON as a sweet spot for functions' memory profiling in a memory tiering serverless solution. DAMON provides a low-cost and fine-grained memory access pattern view, while also displays reduced overhead and better scalability.

Figure 1 visualizes how `userfaultfd()` and DAMON capture a function's memory access pattern for different inputs. We notice that the pages' access counts grow as we increase the input, and each input leads to significantly different memory access pattern. The former is the result of increased memory allocation to the guest, which leads to increased memory accesses.

Observation #4: Memory tiering for serverless requires a more nuanced view of the memory access patterns. Thus, dual-accessed memory information is inadequate to drive fine-grained memory tiering decisions for serverless. Most important, due to the high volume and nature of short-running serverless functions, the profiling overhead and duration should be minimized.

D. Pricing for Serverless

Since most functions either use intensively a small part of memory or they are not memory-intensive, using expensive DRAM entirely is wasteful.

Observation #5: Memory tiering for serverless can lead to substantial memory savings, by identifying and offloading underutilized memory to the slower tier.

These memory cost savings directly benefit both serverless platforms and users. Platforms can offer a dynamically calculated and reduced memory price per unit of time (and number of requests), based on the function memory tiering configuration at any given time. In the worst case scenario, where the function is running entirely in DRAM, users pay the same amount with the currently offered plans. On all other cases, the platform benefits from a reduced memory cost of ownership, while offering users to receive a dynamically reduced plan.

IV. TOSS Design

TOSS is the first tiered memory mechanism for serverless functions that aims to reduce the total memory cost. It offers both fast VM startups and fast function execution, while maintaining a low-cost approach.

A. Overview

TOSS supports memory tiering of serverless functions in four steps as depicted in Figure 4. First, TOSS executes a function in a DRAM-only guest VM and takes a VM snapshot after the execution finishes (***Step I***). TOSS uses this single-tier snapshot to later create the tiered snapshot that will reside in the slow and fast tiers. Second, TOSS enters the profiling phase (***Step II***), gathering memory access information for subsequent invocations. This is contrary to previous systems that assume a single memory access pattern per function. TOSS collects this information for multiple invocations to capture their divergent memory access behavior.

The profiling phase is done solely on DRAM and depends on the function's execution time with profiling enabled. Since 75% of functions run for less than 10s [30] the profiling phase usually ends within minutes. Profiling is not affected by the request distribution and TOSS starts profiling after the first invocation, regardless of future requests' distribution.

After TOSS has gathered enough information to create a representative tiered snapshot, it analyzes and processes the

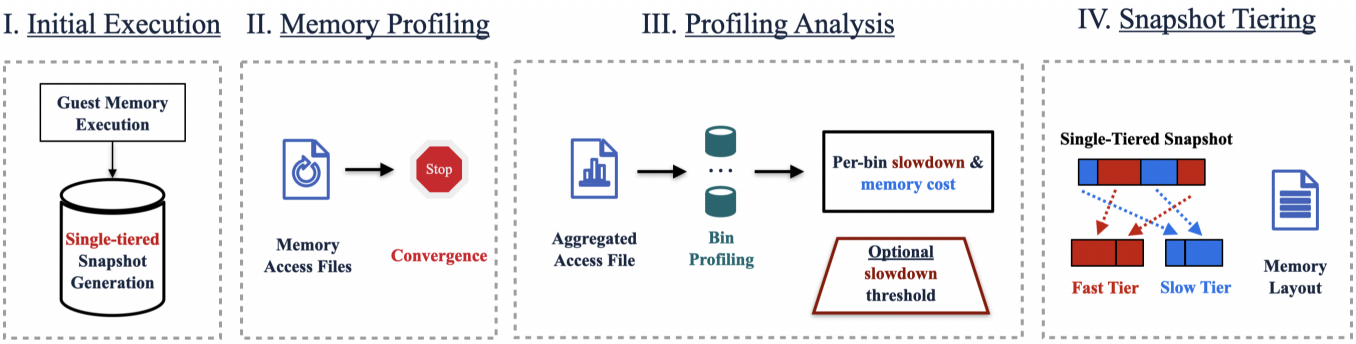


Fig. 4: Overview of TOSS.

profiling information to decide how to partition the guest VM memory between the 2 tiers (***Step III***). Finally, TOSS creates the tiered snapshot that will utilize for future invocations (***Step IV***). Since some functions may have substantially different memory access patterns, we may have to re-profile and enhance our tiered snapshot.

This mechanism addresses the following challenges: First, by capturing a divergent set of memory access patterns, TOSS creates an improved representation of future memory accesses for each function. This reduces or even eliminates overhead from future invocations. Second, it automates the process of choosing the page placement between the two tiers, alleviating clients and vendors from choosing manually the best configuration. Third, TOSS reduces the total memory cost per invocation, as it detects the memory cost of different memory regions, and places them accordingly to each tier.

B. Memory Cost Model

Our goal is to offer a solution to easily evaluate the memory cost of a tiering decision, while enhancing cloud vendors' current pricing structures with heterogeneous memory information. Thus, we introduce a formula that calculates the tiered memory cost, where any memory technology can be leveraged for the two tiers. The formula that we use is the following:

$$SDown * (MB_{Fast} * Cost_{Fast} + MB_{Slow} * Cost_{Slow}) \quad (1)$$

where `SDown` is the slowdown relative to running the function entirely in the fast tier, `MB` is the size in MB for the fast and slow tiers, and `Cost` is the cost per MB for each tier. This formula manages to detect how memory partitioning and the relative slowdown affects the cost.

Since vendors use the $/unit of storage/unit of time (e.g. $/MB/ms) as their pricing model, this formula enhances it in the following ways: If we maintain the same slowdown, but migrate memory from the fast to the slow tier, we reduce the $MB_{Fast} \times Cost_{Fast}$ portion, while increasing the $MB_{Slow} \times Cost_{Slow}$ portion. This reduces the $/MB part, thus reducing the total memory cost. On the contrary, if we maintain the same memory partitioning and increase the slowdown, this increases the number of milliseconds invocations take, increasing proportionally the total memory cost.

V. TOSS Implementation

This section describes the distinct steps that TOSS takes to create the tiered snapshot. We describe in detail the technical challenges that drove our design decisions and how we attempt to solve them.

A. Initial Execution

When TOSS receives the first function invocation, it runs the function solely on DRAM and takes a single-tier snapshot after the execution is complete. This single-tier snapshot is used during the profiling phase, that is initiated from the second invocation until TOSS creates the tiered snapshot.

B. Memory Profiling

The profiling phase lasts for multiple invocations, since TOSS has to account for the variability between different invocations with varying inputs. This phase should be kept short to avoid the profiling overhead and also generate the tiered snapshot promptly. In this phase TOSS uses `DAMON` to gather the access pattern information for each invocation, merging all access patterns in one unified access pattern file. This aids in capturing the different memory access patterns for both significantly different inputs and similar inputs with different guest memory allocations. The unified access pattern file determines whether we should terminate the profiling phase. At the end of each invocation we check if the new profiling information modifies the unified memory access pattern or if it has converged to a stable set. If the access pattern file does not change for N (100 in our prototype) sequential invocations, then we terminate the profiling phase.

We chose DAMON's sampling interval to be 10 µs, so we can collect sufficient profiling data for very short-lived functions. We observed that DAMON's overhead does not depend on the execution time, but mostly on rapid changes in the access pattern. DAMON can easily scale for multiple functions and a huge amount of memory due to its design. It aggregates multiple pages into one region, which significantly reduces the overhead since it only needs to profile a subset of memory. We confirmed this minor performance overhead for peak load in our server.

C. Profiling Analysis

In this phase, TOSS analyzes the information from the unified access pattern file to decide how to split the data between the two tiers. Ideally, we want to keep frequently accessed data in the fast tier, and the rest in the slower tier. However, since our goal is to minimize the memory cost per invocation, we have to explore how different memory parts affect the total memory cost. This stage takes from several hundred milliseconds for a 128 MB snapshot and up to a couple seconds for a 1 GB snapshot.

Initially, we move the zero-accessed pages to the slower tier. Next, we use bin packing [37] to split equally the remaining guest memory regions (different sized contiguous chunks of memory) into bins, by using an open-source heuristic [6] for the bin packing algorithm. Our intuition behind that is to classify memory and identify how each part of the memory affects the performance and memory cost. One potential approach to split memory regions equally is to split the guest VM memory into equally-sized bins. However, this leads to disproportional accesses between bins, since each memory region has different access frequency.

Instead, in order to offer a better understanding regarding the relationship between the overhead and offloaded memory to the slower tier, we split the accesses into N (10 in our prototype) mostly equally accessed bins. While the memory accesses are split into equal bins, from the system's perspective not all accesses are equal, e.g. accessing serially regions performs better than accessing memory randomly. This can lead to slightly variable overheads between accessing different bins. Also, by splitting memory into regions based on the total bin access frequency, we end up with variable bin sizes.

After we split the memory regions to different bins, we have to profile (*bin profiling*) the function for different fast/slow ratios, by varying the number of bins that we offload to the slower tier, i.e. we start with all bins in DRAM and progressively offload more bins to the slower tier. For each one of these configurations we measure the performance and slowdown compared to the first case i.e. all bins in DRAM.

For the bin profiling we need a single representative input that would present good intuition about the overhead and slow tier ratio. We choose the biggest input that we have encountered during the memory profiling phase (***Step II***) for 3 reasons: First, it presents the highest variability and intensity in terms of memory access pattern. Second, it most likely covers the memory access patterns of invocations with smaller inputs. For most functions in our study, we found that the DAMON output from the biggest input execution covers almost completely the patterns of smaller input invocations.

Third, as we see in Figure 2, the longest request (the largest input) suffers the largest slowdown. The only exception are functions that are short-running for every input (less than 10 ms) due to high variability in execution time.

For each one of the bins, we use the slowdown and slow tier ratio to measure the bin's memory cost based on our memory cost formula (Equation 1). We get the memory cost normalized to the case where the function runs fully in DRAM, meaning that any bin that has a memory cost less than 1 will lower the total memory cost. We place all such bins in the slow tier to get the minimum memory cost. Since some functions can be performance critical, TOSS receives as input a slowdown threshold that bounds the slowdown, while optimizing the memory cost. In such case, TOSS sorts the bins with a cost less than 1 based on the slowdown and offload them to the slower tier until the threshold is met.

D. Snapshot Tiering

After TOSS finishes with the analysis phase, it partitions the single-tier memory snapshot file between the slow and fast tiers. First, it creates two snapshot files, one for each tier. Then, it copies serially all memory regions to the corresponding memory file. At the same time, we create a memory layout file that maintains the information about each memory region. This information includes the tier, offset within the snapshot file, offset within the guest VM memory and the size of the memory region. When TOSS receives a function invocation, it reads the memory layout file to load the tiered snapshot and restore the guest VM memory by memory mapping each one of the two tiered files.

E. Snapshot Re-Generation

After creating the tiered snapshot, TOSS has to ensure that the generated snapshot maintains an optimal performance. An example of a non-optimal snapshot generation can occur when a function has mostly short-running requests but received a long-running request during the profiling phase. Thus, TOSS should adapt to changes in future invocations that generate significantly different access patterns. For this purpose, we associate each function with a re-profiling threshold. We calculate this as follows: First, we calculate the **Profiling Overhead** as the sum of the number of invocations running with DAMON enabled and the total slowdown from running the binned profiling part:

$$\#invocations_{DAMON} + \sum_{bin=1}^{10} (1 + Slowdown_{bin}) \quad (2)$$

Second, we calculate a re-profiling **Accelerating Factor**: for each invocation that has a greater execution time than the Longest Running Invocation (LRI) during profiling, we calculate the fraction of these two execution times multiplied by the slowdown encountered during profiling when the function was running entirely on the slow tier.

$$\sum_{inv=1}^{N} \frac{Latency_{inv}}{Latency_{LRI}} * (1 + Slowdown_{Slow}) \quad (3)$$

Last, we need to have a threshold that bounds the profiling overhead for each function, i.e. if we want to bound the profiling overhead to be 0.01% of the total invocations, then this threshold has the value of 0.0001 and we re-profile when the following equation is true:

$$\#iterations * 0.0001 \geq prof\_overhead - accel\_factor \quad (4)$$

Name	Description	Memory	Input Type	Inputs
float_operation	Floating point ops for N numbers	128 MB	N	10, 100, 1000, 10000
pyaes	AES Text encryption	128 MB	Text	64, 256, 1024, 4096 characters
json_load_dump	Read-Modify-Write JSON files	128 MB	JSON File	1, 10, 20, 40 JSON files
compress	File compression	256 MB	File	10 MB, 20 MB, 41 MB, 82 MB
linpack	Solves Ax = b for matrix A	256 MB	Dimension	100, 500, 1000, 2000
matmul	Product of two 2D matrices	256 MB	Dimension	100, 500, 1000, 2000
image_processing	Flips the input image	256 MB	Image	43 kB, 315 kB, 1.8 MB, 4.1 MB
pagerank	Pagerank on a graph	1024 MB	Vertices	90,000, 180,000, 360,000, 720,000
lr_serving	Logistic regression inferencing	1024 MB	Model & Dataset Files	51 kB/10 MB, 83 kB/20 MB, 128 kB/41 MB, 192 kB/82 MB
lr_training	Logistic regression training	1024 MB	Model & Dataset Files	51 kB/10 MB, 83 kB/20 MB, 128 kB/41 MB, 192 kB/82 MB

TABLE I: Functions, memory configurations and inputs that we used in TOSS.

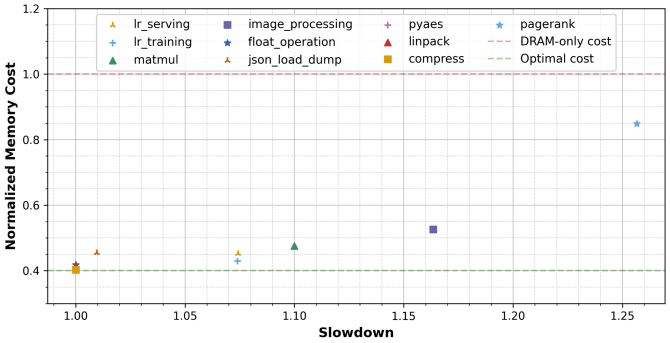


Fig. 5: Normalized cost and slowdown for all functions (Input IV). Lower is better and optimal cost is 0.4.

F. Merging Adjacent Regions

Setup time in TOSS depends on the number of memory regions that are split between the fast and slow tier, since Firecracker uses one memory mapping per memory region. Thus, by minimizing the number of memory mappings, we reduce the overhead and accelerate the startup time. TOSS accomplishes that by merging adjacent regions that have the same attributes, which happens in two cases:

Access count Merging. After combining the access counts from a function's invocations, we merge adjacent regions that have similar access count. We empirically found that merging adjacent regions that vary by less than 100 accesses yields the same results in terms of slowdown.

Bins Merging. After we pack memory regions into bins, we merge adjacent regions that end up in the same tier.

VI. Evaluation

In this section we evaluate TOSS and provide answers to the following questions:

- How much can TOSS reduce the memory cost, compared to the slowdown?
- What is the performance of TOSS, in terms of the startup and function execution time?
- How much memory can TOSS offload to the slow tier?
- Is the longest running invocation representative for bin packing profiling?
- How does placing different bins in the slow tier affect the functions' slowdown and cost?
- How does our bin placement perform for inputs that we have not encountered during profiling?
- How well does TOSS scale for multiple concurrent function invocations?

A. Workloads and Methodology

For the evaluation we use a diverse mix of functions from the FunctionBench benchmarking suite [15] and SeBS [8]. FunctionBench and SeBS include a very diverse set of functions with varying memory intensiveness, needs and patterns that cover a wide range of functions, including cpu-intensive, memory-intensive and storage-intensive functions. For each function we choose the guest VM memory size to be the smallest multiple of 128 MB that executes without guest VM performance issues memory issues. We used multiples of 128 MB, as a common memory size scheme that is used by Lambda, Cloud Functions and Azure.

We drop the host page cache between invocations, which includes clearing all snapshot data that are cached, ensuring the storage accesses for both host and guest. For DAMON, we empirically choose 10 µs as the sampling interval and 16 kB for the minimum memory region size, displaying an 3% overhead on average. We use 100 DAMON files for each input that we include in our snapshots.

We use 2 different snapshots; the first one is based on input IV invocations only, while the other includes memory traces using all inputs. The former is used to evaluate how well TOSS works for smaller inputs/requests that have not been encountered during the profiling phase. The snapshot that includes all inputs represents a more realistic view of a tiered snapshot in TOSS, where each function has encountered divergent inputs during profiling.

Our evaluation excludes caching to provide an objective basis for direct comparison with the SOTA. Caching is orthogonal to our approach, since TOSS can be leveraged in conjunction with caching. Previously proposed techniques are either caching most used functions [12] or predict the request patterns to set up the function before the next invocation [30]. For the former, TOSS can keep the VM alive on both tiers until

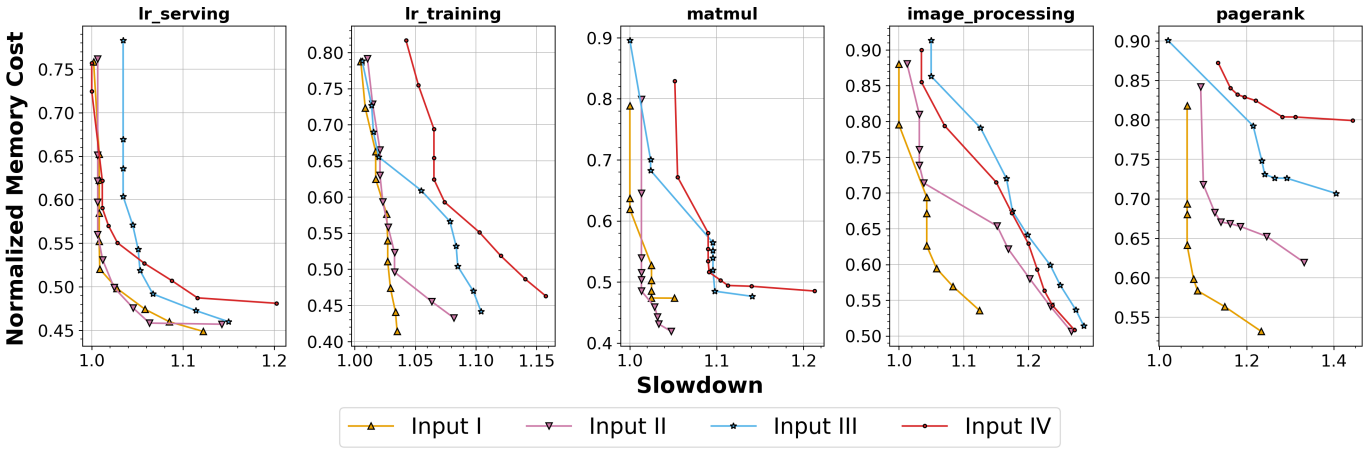


Fig. 6: Slowdown to Memory Cost for each bin. Bins are sorted based on the memory cost efficiency.

evicted, while for the latter TOSS can load the VM before the predicted function execution.

B. Evaluation Platform and SOTA

Our platform includes a 2×20-core Intel® Xeon® Gold 6230 CPU @ 2.10 GHz, with 96 GB of DDR4 DRAM (fast tier), 768 GB of Intel® Optane™ Persistent Memory (slow tier). We also use an Intel® Optane™ DC SSD with sequential read and write of up to 2,500 MB/s and up to 2,200 MB/s, respectively, and random read and write of up to 550,000 IOPS. We use the same 5.18 Linux kernel for both host and guest and disable hyperthreading. Our implementation includes approximately 2.8K LoC and we use 2.5 as the cost ratio between the fast and slow tiers, according to public cost data [23].

We compare TOSS with REAP, the snapshot-based state-of-the-art serverless system. REAPtakes a snapshot during the first function invocation and further optimizes snapshot loading by prefetching the WS pages on memory and populating the respective page table entries. This leads to reduced page fault latency during the first accesses in the working set regions of the guest memory.

C. Memory Cost

In this section we evaluate how memory cost is affected by design decisions on TOSS, and we show its relationship with slowdown and memory offloaded to the slow tier.

1) Minimum Memory Cost

Figure 5 contains the minimum memory cost over the slowdown for all functions running with input IV. We use the snapshot generated by all inputs and compare these costs with the DRAM-only cost and the Optimal cost that can be achieved. As we used 2.5 as the cost ratio between the fast and slow tiers, the normalized optimal cost from our formula is 0.4, considering the case where all memory resides in the slow tier with no slowdown. TOSS displays a slowdown between 0% and 25.6% (average: 6.7%), and a normalized memory cost between 0.4 and 0.87 (average: 0.48). It is able to achieve a near-optimal memory cost, while maintaining a slowdown of less than 10% for 7 out of 10 functions. We note here that the results are similar when using the Input IV-based snapshot.

The functions that suffer the biggest slowdown are *image_processing* and *pagerank*, displaying a slowdown of 17% and 25%, respectively. This might not be acceptable for some latency critical functions, so the client can choose to set a slowdown threshold, which will minimize the slowdown and increase the memory cost. In addition, *pagerank* is a very memory intensive function, which limits how much memory we can offload to the slow tier, limiting the memory cost savings to 15%. In such cases, cloud providers can decide to run these functions fully in DRAM, since the memory cost savings might be insufficient to outweigh the performance profiling overhead.

Other functions in our benchmarks are also memory intensive, with *matmul* being one of the most intensive ones. However, *pagerank* is the only function in our benchmarks that displays such a low memory cost saving. We attribute this to the fact that it displays the same intensity across most of its working set, thus we can only offload a small part of its working set to the slow tier. We use `perf` [39] to measure the memory intensiveness by collecting the hardware counters that measure the fraction of cycles stalled due to outstanding Last-Level-Cache miss demand loads.

Memory Ratio Between Tiers. Table II shows the amount of memory that TOSS offloads to the slower tier for the minimum cost configuration. In average, TOSS offloads 92% of the memory, with 5 functions being entirely offloaded. For *compress* this was expected, since it displayed negligent slowdown when we offloaded it fully in the slower tier (see Figure 2). Last, since *pagerank* is a very memory intensive function TOSS manages to offload only 49.1% of the memory, which we still consider a sufficient main memory usage reduction.

We have 2 important observations: First, TOSS correctly identifies that most functions have a small hot subset, offload-

Function	Slow Tier Percentage
lr_serving	94.8%
lr_training	100%
matmul	92%
image_processing	100%
float_operation	94%
json_load_dump	100%
pyaes	94.7%
linpack	95.9%
compress	100%
pagerank	49.1%

TABLE II: Memory offloaded to slow tier for the minimum cost configuration in TOSS.

ing most memory to the slower tier, showcasing the importance of memory tiering for serverless. Second, focusing entirely on minimizing memory cost, and not the slowdown of the function, can be more beneficial. This is the case for *image processing*, which displays a huge slowdown, but is still fully offloaded to the slower tier due to the maximum memory reduction goal.

2) Incremental Memory Cost

Figure 6 visualizes the bin packing portion (Section V-C) in TOSS for the 5 functions in Figure 2 that display the worst slowdown. We show how incrementally moving bins to the slow tier affects the slowdown and memory cost for all inputs. Bins are sorted based on their individual memory cost and offloaded to the slow tier one bin at a time, with the leftmost point being the first bin and so forth.

We observe that as the input size increases, the accumulated slowdown increases as well; this confirms our decision to use the longest execution request for bin packing profiling. There are some outliers here: First, we notice that for *image_processing* input III slight exceeds input IV in terms of slowdown. This is not surprising, since the function displays a higher latency variability. Nevertheless, memory cost and slowdown are very similar between inputs II, III, IV. Second, regarding *lr_serving*, input III displays a slightly increased slowdown (2%) compared to input IV for the initial bins; we assume that this is within the acceptable variability limits.

Last, we observe that the memory cost is also proportional to the input size, i.e. by using the largest invocation time request we get a more conservative view of the memory cost. This is beneficial for our approach, since our calculated memory costs are the upper bound for each case, meaning that invocations with smaller inputs can improve memory cost even further.

3) Snapshot-based Memory Cost

Achieving the minimum cost heavily depends on the tiered snapshot representability. Below we are discussing our observations with using the two different snapshots, while we omit the figure due to space limitations.

Input IV vs. All Inputs. We evaluate how our tiering mechanism works for smaller inputs that were not encountered during the profiling phase. We compare the minimum costs between the two different snapshots, the one based on input IV invocations and the other based on all inputs invocations. Furthermore, we notice that for most functions/inputs the two snapshots display a negligible difference, with the average cost variance being 7.2%. This variance is increased in two cases: for a few short-running invocations (less than 10 ms) and *pagerank*, which is extremely memory intensive.

The former have a very short invocation time, leading to very volatile results, even when they run completely on DRAM. The latter has an average cost variance of 10% with and a max cost variance of 21.5%, which also is the highest by far for all invocations. We notice that this happens both due to *pagerank*'s memory intensity and that it requires a more diverse set of requests during the profiling phase. Excluding the short-running invocations and *pagerank*, the average cost variance drops to 2.4%.

All things considered, TOSS performs well for smaller inputs that have not been encountered during the profiling phase, while for larger invocations we use the re-profiling mechanism. Overestimating the memory cost provides an upper bound to the memory cost that can be used as a guarantee for the memory savings.

Input IV vs. Individual Input Placement. Next, we evaluate how well the bin placement based on input IV works for the rest of the inputs. We compare the bin placement that minimizes input IV cost with the bin placement that minimizes the cost for each individual input. The average difference is 6.1%, which shows that using just the maximum input for the bin placement can achieve nearly optimal minimized cost. Again, we see a few outliers that include the short-running invocations; excluding those, the average cost variance drops to 3.3%.

D. Invocation time

Our main goal is to reduce the memory cost, while maintaining an acceptable slowdown. We evaluate the setup and total invocation (*setup & execution*) time, and compare TOSS with REAP. For TOSS we use the minimum cost tiered snapshot, while for REAP we use all combinations of snapshot and execution inputs and show the minimum, average and maximum setup time.

Figure 7 compares the setup time between REAP and TOSS, showing that TOSS has a significantly lower setup time in most cases, with REAP displaying up to 52× higher setup time. Because of the unified tiered snapshot in TOSS, the setup time for each function is constant. On the contrary, REAP's setup time increases as the size of the snapshot increases. REAP displays a faster setup time by a small margin only when the working set is very small. Out of the 10 functions, only `pyaes` and `float_operation` have a slightly lower setup time when using REAP. REAP's expensive setup time is the result of having to load in-memory all pages that were accessed during the invocation time. This inflates the setup time, particularly for functions that have mostly a small working set, while facing a larger input during REAPs profiling phase.

Figure 8 compares the normalized total invocation (*setup & execution*) time between REAP and TOSS. Compared to

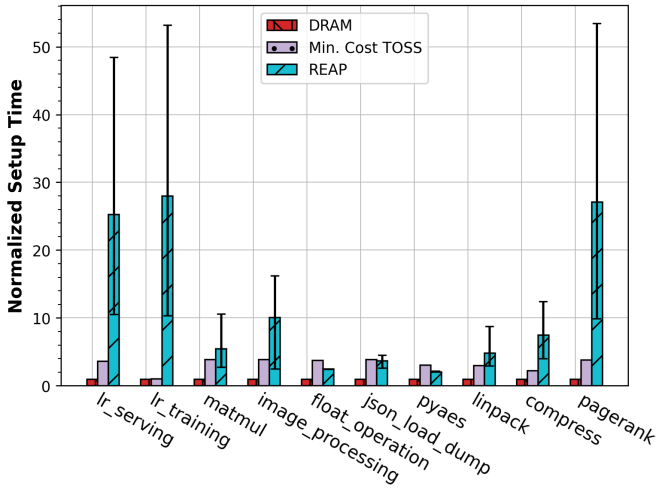


Fig. 7: Setup time for all combinations of execution and snapshot inputs, normalized to DRAM snapshot setup time.

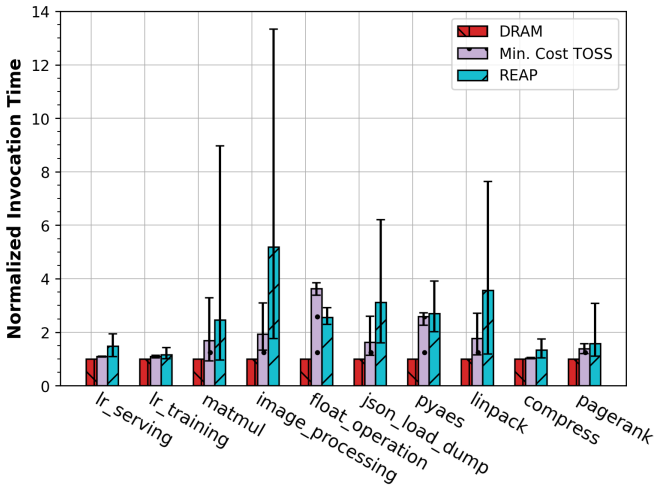


Fig. 8: Total invocation time (setup & execution) for all combinations of execution and snapshot inputs, normalized to DRAM invocation time

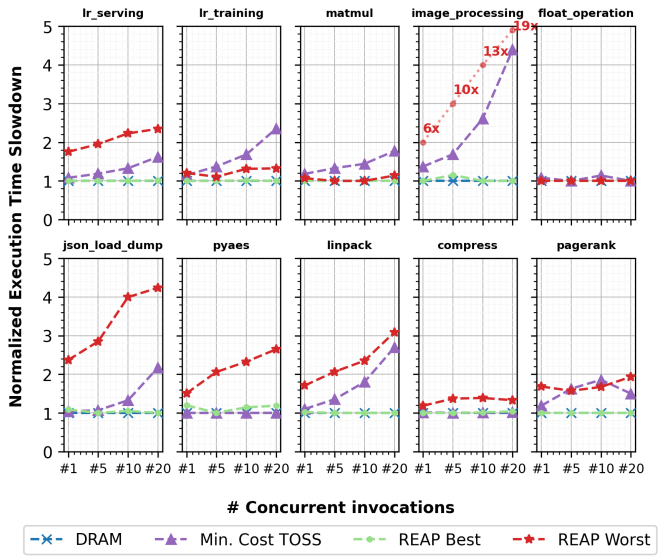


Fig. 9: Execution time slowdown for concurrent invocations, normalized to DRAM execution time. For *image_processing*, REAP Worst surpasses the y axis.

DRAM, TOSS displays 1.78× slowdown on average, and up to 3.8×, while REAP a 2.5× slowdown on average, with up to 13×. In addition, our data showed that *image_processing* displays a high slowdown not only for different snapshot and execution input, but also when using input II for both, displaying a 4× slowdown. This showcases the importance of profiling multiple invocations, to capture the different memory access patterns caused both by different inputs and memory allocation patterns.

E. TOSS Scalability

Modern serverless platforms run multiple functions at a time, so next we test the execution time scalability for concurrent invocations. We use up to 20 concurrent invocations, which is the peak load that our server can achieve, since we have 20 cores and we disable hyperthreading akin to [1], Figure 9 contains the execution time slowdown for TOSS and REAP, normalized to the DRAM case. We use execution input 4, the largest input that causes the most slowdown, and test the scalability for 1, 5, 10 and 20 concurrent invocations of each function. For REAP we test two snapshot cases: same snapshot and execution input (*REAP Best*), and input 1 for snapshot input (*REAP Worst*). The former has similar results with DRAM, since during the profiling phase it marks nearly all memory as the working set. The latter displays significant slowdown for most functions as we increase the number of concurrent invocations, with an average of 3.79× and up to 19× slowdown for 20 concurrent invocations. TOSS outperforms REAP Worst with different execution and snapshot inputs in 8 out of 10 functions, with an average slowdown of 1.95× and up to 4.2×.

For TOSS we notice that 5 out of 10 functions have similar performance to DRAM, showing that our single-function profiling is appropriate to drive platforms' memory cost decisions. Interestingly, *pagerank*, a memory intensive function, displays similar scalability to the DRAM case. This is the result of offloading only 49.1% to the slower tier, keeping the most intensive memory to DRAM. For the remaining functions that display a worse scalability, serverless providers can either use a more conservative ratio between the fast and slow tiers or modify the memory cost policy to reflect the increased slowdown.

F. Limitations in TOSS

There are a few cases where TOSS can potentially lead to reduced effectiveness or performance overhead. Extremely memory intensive functions limit the offloaded memory to the slow tier, thus limiting potential cost reductions while introducing the profiling overhead. However, only *pagerank* among functions that we tried exhibits such memory intensiveness (15% cost reduction in Figure 5). Another potential issue might occur for functions with highly varying inputs that will lead to multiple reprofiling cycles. In practice, we are not aware of functions that display this input variability.

VII. Related Work

This sections contains the related work regarding previous cold start and memory tiering solutions.

A. Cold Start

Solutions for the cold start problem can be broadly classified into two categories: caching and snapshotting. Two representative works on caching are proposed by Oakes et al. [26] and Fuerst et al. [12]. Oakes et al. [26] introduced a concept called Zygotes, which are pre-initialized processes containing the necessary libraries required by applications. This allows application processes to be forked from Zygotes, reducing the library initialization cost. Fuerst et al. [12] extends caching models to the serverless domain by demonstrating the equivalence between caching and function keep-alive. Based on this, they proposed a Greedy-Dual keep-alive policy to mitigate cold start overhead.

Snapshot and restore is a novel approach to the cold start problem. Du et al. [10] proposed to accelerate the restore process by on-demand recovering application and system state. Similarly, AWS Firecracker [1] employs the lazy restore design. While these approaches help reduce cold start overhead, they can result in extra page faults during the function execution.

B. Memory Tiering

In recent years, the emergence of non-DRAM memory technologies, including NVMe SSDs and NVMs, have opened up new opportunities for tiered memory systems. These technologies are cheaper than DRAM and offer the potential to reduce expenses for modern datacenters. For example, a study conducted by Weiner et al. [36] at Meta revealed that even though the cost of DRAM can account for 33% of the server's total cost, a significant portion of memory is not frequently accessed. This insight has led to the development of a mechanism where cold memory can be offloaded to slower tiers, such as SSDs, or compressed in DRAM, reducing the total server cost.

Apart from SSDs, non-volatile memories also has the potential to act as the slow tier in memory tiered systems. Duraisamy at el. [11] built a tiered memory system that consists of DRAM and a variant of Intel Optane Persistent Memory. By dynamically managing the page placement, they managed to reduce the total cost by up to 25% with an up to 5% slowdown.

While there is no limitation to the slower memory's performance characteristics, TOSS would benefit the most from having a comparable performance between the fast and slow memories. Since the existing memory technologies (SSDs, DRAM, CXL-attached DRAM, PMEM, GPU memory etc.) have comparable performance, we expect that TOSS can be utilized for any of these combinations.

TOSS can support accelerator memory, provided that it can be exposed by the system. For instance, TOSS can be used with DRAM as the slow, capacity tier and a GPU's memory as the fast, small tier. A similar GPU unified memory idea is utilized by NVIDIA [25] and past papers [45], [14], where DRAM and flash memory act as a capacity extension of the GPU's memory.

VIII. Conclusion

We present TOSS, the first memory tiering mechanism for serverless snapshots that aims to reduce the memory cost per invocation. TOSS enables efficient memory tiering through enhanced memory access characterization, leading to better accuracy across different function invocations. We also propose a simple memory formula that evaluates the tiered memory cost, maintaining compatibility with cloud vendors such as AWS Lambda, Google Cloud and Azure. Based on this formula we show how TOSS achieves near optimal memory cost by offloading most memory to the slower tier. Last, we show how memory cost reduction is decoupled from slowdown optimization.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This paper is supported in part by the PRISM center within JUMP 2.0, an SRC program sponsored by DARPA

References

[1] 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.

[2] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Soramichi Akiyama and Takahiro Hirofuchi. Quantitative evaluation of intel pebs overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, ROSS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[4] 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.

[5] AWS. Improving startup performance with Lambda SnapStart, Online; Accessed October, 2024. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html.

[6] bfmaier. binpacking 1.5.2, Online; Accessed October, 2024. https://pypi.org/project/binpacking/.

[7] Alibaba Cloud. DATOP, Online; Accessed October, 2024. https://github.com/aliyun/data-profile-tools.

[8] 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.

[9] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 356–370, New York, NY, USA, 2020. Association for Computing Machinery.

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

[11] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[12] 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 2021, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 521–534, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1357–1370, New York, NY, USA, 2020. Association for Computing Machinery.

[15] 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.

[16] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728. USENIX Association, July 2021.

[17] Amazon Web Services Labs. DAMO, Online; Accessed October, 2024. https://github.com/awslabs/damo.

[18] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Linux kernel community. mincore(2) — Linux manual page, Online; Accessed October, 2024. https://man7.org/linux/man-pages/man2/mincore.2.html.

[20] Linux kernel community. userfaultfd(2) — Linux manual page, Online; Accessed October, 2024. https://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[21] Mahesh Natu and Thomas Won Ha Choi,. Compute Express Link™(CXL™): Supporting Persistent Memory, Online; Accessed October, 2024. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL-2.0-Presentation-Persistent-Memory-20210615_FINAL.pdf.

[22] Theodore Michailidis, Steven Swanson, and Jishen Zhao. PMShifter: Enabling Persistent Memory Fluidness in Linux. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 1–8, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Timothy Prickett Morgan. The Era Of Big Memory Is Upon Us, Online; Accessed October, 2024. https://www.nextplatform.com/2020/09/23/the-era-of-big-memory-is-upon-us.

[24] Fabiha Nowshin and Yang Yi. *ReRAM-Based Neuromorphic Computing*, pages 43–65. Springer International Publishing, Cham, 2023.

[25] Nvidia. CUDA UNIFIED MEMORY, Online; Accessed October, 2024. https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf.

[26] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 57–69, USA, 2018. USENIX Association.

[27] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, apr 2021.

[29] SeongJae Park. DAMON: Data Access Monitor, Online; Accessed October, 2024. https://sjp38.github.io/post/damon/.

[30] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[31] The kernel development community. DAMON-based Reclamation, Online; Accessed October, 2024. https://docs.kernel.org/admin-guide/mm/damon/reclaim.html.

[32] TSMC and ITRI. ITRI and TSMC's High-Speed Breakthrough: SOT-MRAM, Online; Accessed October, 2024. https://itritoday.itri.org/116/content/en/unit_03-1.html.

[33] TSMC and ITRI. TSMC's Next-Gen Memory Breakthrough: Seizing Opportunities in AI and High-Performance Computing, Online; Accessed October, 2024. https://www.trendforce.com/news/2024/01/18/news-tsmcs-next-gen-memory-breakthrough-seizing-opportunities-in-ai-and-high-performance-computing/.

[34] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. *Benchmarking, Analysis, and Optimization of Serverless Function Snapshots*, page 559–572. Association for Computing Machinery, New York, NY, USA, 2021.

[35] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 133–145, USA, 2018. USENIX Association.

[36] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Octoberank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[37] Wikipedia contributors. Bin packing problem — Wikipedia, the free encyclopedia, Online; Accessed October, 2024. https://en.wikipedia.org/w/index.php?title=Bin_packing_problem&oldid=1154327033.

[38] Wikipedia contributors. Compute express link — Wikipedia, the free encyclopedia, Online; Accessed October, 2024. https://en.wikipedia.org/w/index.php?title=Compute_Express_Link&oldid=1241521209.

[39] Wikipedia contributors. Perf (linux) — Wikipedia, the free encyclopedia, Online; Accessed October, 2024. https://en.wikipedia.org/w/index.php?title=Perf_(Linux)&oldid=1126307013.

[40] Wikipedia contributors. Persistent memory — Wikipedia, the free encyclopedia, Online; Accessed October, 2024. https://en.wikipedia.org/w/index.php?title=Persistent_memory&oldid=1144357434.

[41] Yuanchao Xu, Chencheng Ye, Yan Solihin, and Xipeng Shen. Ffccd: fence-free crash-consistent concurrent defragmentation for persistent

memory. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 274–288, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Yan Sha, Xiaofei Liao, Hai Jin, and Yan Solihin. Reconciling selective logging and hardware persistent memory transaction. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 664–676, 2023.

[43] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Yan Sha, Xiaofei Liao, Hai Jin, and Yan Solihin. Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 762–777, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 98–105, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 395–410, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Pawel Zuk and Krzysztof Rzadca. Scheduling methods to reduce response latency of function as a service. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 132–140, 2020.