FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture

Chuhao Xu
Shanghai Jiao Tong University
Shanghai, China
barrin@sjtu.edu.cn

Yiyu Liu
Shanghai Jiao Tong University
Shanghai, China
liu_yiyu@sjtu.edu.cn

Zijun Li
Shanghai Jiao Tong University
Shanghai, China
lzjzx1122@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Han Zhao
Shanghai Jiao Tong University
Shanghai, China
zhao-han@cs.sjtu.edu.cn

Deze Zeng
China University of Geosciences
Wuhan, China
deze@cug.edu.cn

Qian Peng
Huawei Cloud
Shenzhen, China
pengqian19@huawei.com

Xueqi Wu
Huawei Cloud
Shenzhen, China
wuxueqi4@huawei.com

Haifeng Zhao
Huawei Cloud
Shenzhen, China
zhaohaifeng4@huawei.com

Senbo Fu
Huawei Cloud
Shenzhen, China
fusenbo@huawei.com

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

Abstract

In serverless computing, an idle container is not recycled directly, in order to mitigate time-consuming cold container startup. These idle containers still occupy the memory, exasperating the memory shortage of today's data centers. By offloading their cold memory to remote memory pool could potentially resolve this problem. However, existing offloading policies either hurt the Quality of Service (QoS) or are too coarse-grained in serverless computing scenarios.

We therefore propose **FaaSMem**, a dedicated memory offloading mechanism tailored for serverless computing with memory poor architecture. It is proposed based on our finding that the memory of a serverless container allocated in different stages has different usage patterns. Specifically, FaaSMem proposes Page Bucket (Pucket) to segregate the memory pages in different segments, and applies segment-wise offloading policies for them. FaaSMem also proposes a semi-warm period during keep-alive stage, to seek a sweet spot between the offloading effort and the remote access penalty. Experimental results show that FaaSMem reduces the average local memory footprint by 9.9% - 79.8% and improves the container deployment density to 108% - 218%, with negligible 95%-ile latency increase.

CCS Concepts: • **Software and its engineering** → **Cloud computing**; **Memory management**; • **Computer systems organization** → **Cloud computing**; • **Hardware** → *Emerging technologies*; *Memory and dense storage.*

Keywords: FaaS, Serverless Computing, Memory Pool architecture, Memory Offloading

ACM Reference Format:
Chuhao Xu, Yiyu Liu, Zijun Li, Quan Chen, Han Zhao, Deze Zeng, Qian Peng, Xueqi Wu, Haifeng Zhao, Senbo Fu, and Minyi Guo. 2024. FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620666.3651355

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04…$15.00
https://doi.org/10.1145/3620666.3651355

1 Introduction

Serverless computing [18, 31, 32, 43, 56, 60, 63], also known as Function-as-a-Service (FaaS), has gained significant popularity in the cloud-native era. It decouples a monolithic application into fine-grained functions, and uses lightweight containers to execute requests when triggered [3, 41, 48]. Popular serverless platforms like AWS Lambda [11], Azure

Functions [12], and Google Cloud Functions [5] will manage the infrastructure, dynamically scaling-out containers as workload changes for each function independently.

Serverless containers usually provide a keep-alive stage upon the completion of function execution [25, 42, 57], waiting for potential future requests, to avoid the time-consuming cold-starts. By analyzing the Azure trace [69], we find the keep-alive stage (or strategy) has resulted in a significant amount of idle memory footprint for each container. Consequently, the memory inefficiency of serverless computing has emerged as one of the major concerns for cloud providers, as the number of serverless containers within a node increases into thousands [16, 41], exasperating the growing memory shortage [29, 37, 40, 65] issue of today's data center.

Many prior works [22, 26–28, 30, 47] have shown that the emerging memory pool architecture has the potential to address the memory inefficiency and shortage in data centers. For instance, Pond [40] utilizes memory pooling to reduce DRAM cost for the virtual machine scenarios, which manages to harvest the stranded memory from the load-unbalanced nodes. Given that serverless containers are idle during keep-alive, they could be a good fit for the memory pool architecture and benefit from the increased deployment density. However, even though the memory pool enables the compute nodes to offload some pages to the remote to save local memory space, the offloaded part would suffer the remote access latency penalty. Therefore, to avoid severe performance degradation, only the infrequently accessed pages, i.e., cold pages, are commonly appropriate candidates for offloading [39, 65].

When employing existing memory offloading policies, we find they are not plug-and-play solutions for serverless containers. Most importantly, these policies are agnostic to the container stage, and perform sampling constantly during the keep-alive stage. In this case, hot pages required for subsequent requests are still considered cold and then offloaded, resulting in subsequent requests suffering the remote access latency penalty. It motivates us that in order to efficiently and safely offload memory in serverless architecture, we should explore the memory layout of serverless containers to seek opportunities for memory offloading.

Drawing inspiration from different stages during the container's lifetime, we propose a segment-based memory layout for serverless containers, namely the runtime, init, and execution segments. In the runtime segment, we find numerous memory pages have been allocated during runtime loading, but hardly accessed again during function execution. These cold pages can be offloaded efficiently and safely with negligible performance degradation. In the init segment, many containers handle few requests across lifetime, leading to a high risk of misjudgement of cold pages. Meanwhile, numerous hot pages still persist during keep-alive, overwhelming scarce memory resources. In the last execution segment, memory allocations during function execution often exhibit short-lived pattern, as they will be freed immediately once the execution completes. Considering the relatively short execution time, the benefit of memory offloading in execution segment is marginal.

To seize the opportunities to embrace the memory pool architecture in serverless, we encounter three primary challenges. First of all, the memory pages allocated during different container stages have been managed in a unified manner by the current kernel memory subsystem [8]. A method is required to segregate different pages towards different segments. Secondly, as each segment exhibits unique memory characteristics with different access patterns, the offloading polices should be carefully tailored for different segments, avoiding severe performance degradation. Thirdly, numerous hot pages reside during keep-alive stage, while the conventional wisdom is that these hot pages are not suitable for offloading. Therefore, the keep-alive strategy under the memory pool architecture deserves reconsideration.

Therefore, we propose **FaaSMem**, a dedicated memory pooling mechanism for serverless architecture. FaaSMem introduces the Page Bucket (Pucket) to segregate pages from different segments into different buckets of a serverless container and applies a series of segment-wise offloading policies towards them. For the Runtime Pucket, FaaSMem offloads the cold pages upon the first request completes. For the Initialization Pucket, FaaSMem lazy-offloads the cold pages by a request-window, which is determined by a real-time profiling. Further, to deal with numerous hot pages in Runtime and Init Puckets during container keep-alive, a shared hot page pool in also introduced to manage them. FaaSMem gradually offloads the hot pages in this pool, making the container semi-warm. It guarantees the tail latency by selecting an appropriate semi-warm start timing based on real-time profiling. Semi-warm helps FaaSMem seek a sweet spot between the offloading effort and the remote access penalty.

To the best of our knowledge, FaaSMem is the first work that exploits the memory pool architecture for serverless computing. The main contributions are as follows.

- **Comprehensive analysis of serverless memory layout.** The proposed segment-based memory layout reveals opportunities of memory offloading, and motivates the Pucket design in FaaSMem.
- **The segment-wise cold page offloading policies for different Puckets.** FaaSMem introduces Pucket to segregate memory pages among three segments, and applies segment-wise offloading policies corresponding to unique memory patterns.
- **The proposal of semi-warm period for serverless containers under memory pool architecture.** We reframe the keep-alive stage under the memory pool architecture with semi-warm period, and provide the methodology to adopt the semi-warm.

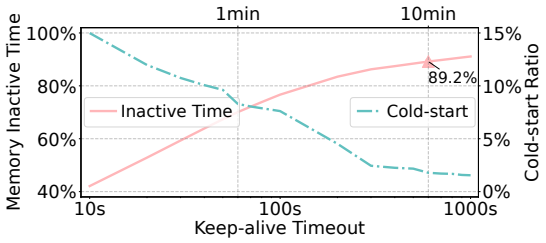


Figure 1. The containers memory inactive time and the requests cold-start ratio when keep-alive timeout varies.

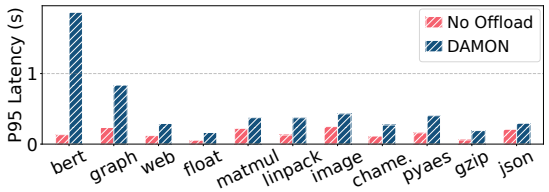


Figure 2. Response latency when offload via DAMON.

We evaluate FaaSMem using real-world applications and Azure traces. Experimental results show that compared to serverless systems without memory pool architecture, FaaSMem reduces the average local memory footprint by 9.9% - 79.8% and improves the container deployment density to 108% - 218%, with negligible 95%-ile end-to-end latency increase. FaaSMem's source code is publicly available at https://github.com/BarrinXu/FaaSMem.

2 Background and Motivation

In this section, we analyze the memory inefficiency problem in serverless computing, and problems of leveraging memory pool to resolve the inefficiency.

2.1 The Memory Inefficiency Problem

To alleviate frequent cold startups [19, 41–43, 50, 57], serverless platforms usually implement a keep-alive strategy to delay the timing of container recycling. This approach preserves the container in a warm state for a certain duration after the function container processes any request.

However, the keep-alive strategy results in severe memory inefficiency. We conduct a simulation based on the Azure Functions Invocation Trace 2021 [69] that involves 424 functions with 1,980,951 invocations. By setting the keep-alive threshold, we can count the incidence of cold-start, the total container keep-alive time, and the total container lifetime. Figure 1 illustrates the metrics of "memory inactive time" and "cold-start ratio" under varying keep-alive timeouts in the trace. Higher inactive time suggests that the system expends more time retaining container memory.

As observed, a superior user experience, denoted by a lower cold-start ratio, leads to a larger number of inactive memory pages occupying host space. When cloud providers commonly set the maximal keep-alive timeout at around 10 minutes, container memory remains inactive for 89.2% of its lifetime. Furthermore, even with a 1-minute keep-alive timeout, memory stays inactive for 70.1% of its lifetime, indicating inefficient memory utilization.

The memory inactive time is high because a keep-alive container retains memory resources even when it is free. Meanwhile, there is a growing demand for memory capacity in today's data centers, leading to severe memory shortage[29, 37, 40, 65]. In this case, memory becomes a bottleneck resource that limits container deployment density in serverless.

2.2 Inefficiency of Leveraging Memory Pool

In the face of growing memory demand, simply upgrading memory for compute nodes would have to replace the old ones, resulting in additional waste [15]. Therefore, a feasible approach to improve the memory efficiency is leveraging the emerging memory pool design in data centers [27, 28, 30, 37, 38, 46, 58, 62, 65], which allows for the reuse of older or retired memory with enhanced cost-efficiency.

To this end, a data center could have some memory nodes, and these memory nodes are accessible through high bandwidth connection (e.g., InfiniBand and CXL). In this way, compute nodes are able to offload some cold pages to the memory pool to save local memory space. Such design has been proved to be efficient in many application scenarios [15, 17, 40].

There are generally two types of policies for managing the memory offloading: *feedback-based* [37, 65] and *sample-based* [7, 39, 47, 51]. With feedback-based policy, such as TMO [65], the memory is often slowly offloaded to avoid severe performance degradation, and the offloading immediately stops once the performance drop reaches a pre-defined threshold. With sampling-based policy, such as DAMON [7, 51], the access hotness of all the memory pages are monitored, and all the cold pages are offloaded immediately.

TMO tries to offload only 0.05% of the total memory every 6 seconds, and the offloading ratio of a 10-minute period is within 3.0%. This is impractical for short-lived serverless containers, while most of them only lives for tens of minutes due to temporary load and keep-alive timeout. Our detailed experiments in Section 8 also show the poor performance of TMO in serverless computing scenario.

We also show the performance of DAMON in handling memory offloading in serverless computing scenario. In this experiment, we use all 11 benchmarks from evaluation section with invocation traces that are randomly selected from the Azure Functions trace [69]. Figure 2 shows the 95%-ile latency of the benchmarks when DAMON is used to offload cold pages. As observed, the response latency of the benchmarks increase by up to 14X, when the memory is offloaded by DAMON. This is mainly because DAMON performs sampling constantly during the keep-alive stage, and a large

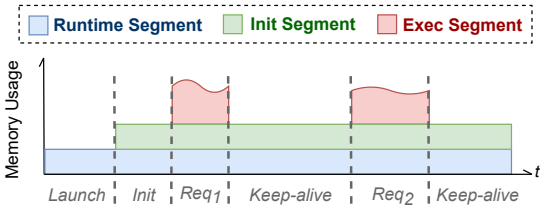


Figure 3. The memory layout of a serverless container.

number of hot pages required for subsequent requests are all considered cold. As a result, memory offloading is heavily influenced by these sampling misidentification, leading to fatal decisions.

Existing solutions are not suitable in serverless computing scenario, because they are agnostic to the changes of memory access patterns in different container stages. We therefore dig into the memory usage pattern of a serverless container, in order to explore the appropriate memory offloading without hurting user experience.

3 Exploring FaaS Memory Usage

Inspired by the stages during the container's lifetime, we analyze the memory usage behaviors according to the stages, and identify different characteristics in different stages. Specifically, Figure 3 illustrates three segments we identified based on different stages of a serverless container.

Specifically, in the beginning of a container's lifetime, it experiences a cold-start container launching, where the memory footprint depends on the container runtime provided by FaaS platforms, mostly not associated with user-uploaded function code. This runtime-related memory is identified as the runtime segment (*Segment-1*). Subsequently, the container proceeds to the initialization of the function code, loading user-defined packages and libraries into memory (*Segment-2*). The memory of the runtime segment and init segment constitutes the base memory footprint of the container. Following container launch and initialization, the container is prepared to handle requests. Additional memory beyond the base footprint is allocated for the function's temporary variables and states during each execution (*Segment-3*), which is subsequently reclaimed upon completion.

Based on the proposed memory layout, we dig into the memory characteristics of each segment to discuss the opportunities of memory offloading respectively.

3.1 Segment-1: Memory of Container Runtime

A container for a serverless function contains not only the function code itself, but also the runtime environment on which it depends. As the statistics of Azure Functions [57] show that 99.9% of applications' containers take up higher than 76 MB of memory, the practical explanation is that the runtime segment takes a large amount of memory footprint.

We collect the memory footprint on real serverless systems by executing a hello-world function. In this experiment, the containers are all created from the official-build images of Apache OpenWhisk [2] and Azure Functions [12]. We identify the inactive memory pages after a hello-world function execution based on whether the *Access bit* of a page is changed. Figure 4 shows the inactive memory of the hello-world container. Since a hello-world function itself takes up little memory, the result can be recognized as the memory usage of container runtime.

As shown in Figure 4, the inactive runtime memory of a container alone takes up considerable amounts and is correlated with the type of programming languages. For Azure, all three take up more than 100 MB of memory, while for OpenWhisk, the Python and Java based runtimes also take up 24 MB and 57 MB, respectively. Using Java as runtime has the largest amount of inactive memory due to Java JVM.

Takeaway. The inactive memory of runtime segment significantly reduces the memory efficiency. A large number of cold memory pages that are allocated during runtime loading but hardly ever accessed later.

3.2 Segment-2: Memory of Function Initialization

After loading the runtime environment, the container proceeds to the function initialization stage. This stage is user-defined, leading to diverse memory footprints for different functions. Generally, this segment is designed to fulfill the requirements of subsequent requests, including dependent packages, global variables, datasets, and caches.

Cold memory pages that are suitable for offloading, can be identified by analyzing the memory access patterns of historical traces. However, this method encounters challenges with serverless functions that experience infrequent invocations. Figure 5, derived from Azure Trace simulations, illustrates the distribution of the number of requests handled by each container. Nearly 60% of containers invoke at most two requests throughout their lifetime. In this case, we cannot collect adequate data to infer the memory access pattern.

Meanwhile, an increasing number of real-world applications, such as ML inference [68], web services [32], and graph processing [1], have been migrated to the serverless architecture. These applications often exhibit a substantial number of hot pages in the initialization segment, accessed during each request. Figure 6 illustrates the memory footprint of a ML inference (BERT) function over time. The first 5 seconds is the initialization stage, during which enormous memory is allocated and accessed (from 0 to 1000 MB). Some of the memory is released afterwards. During the execution stage, about 610 MB of memory is accessed. The strips after 8s in the figure illustrate the memory access pattern, showing that a significant amount (about 400 MB) of hot pages generated during initialization is accessed in each request.

Takeaway. The limited number of requests per-container handles makes the cold-page identification in initialization

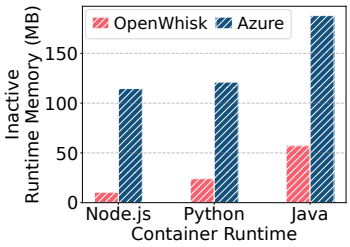


Figure 4. The inactive memory of the runtime segment.

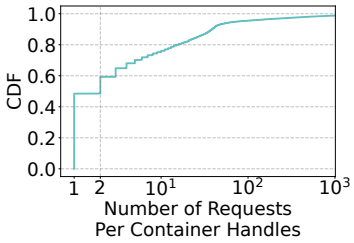


Figure 5. CDF of the number of requests per container handles.

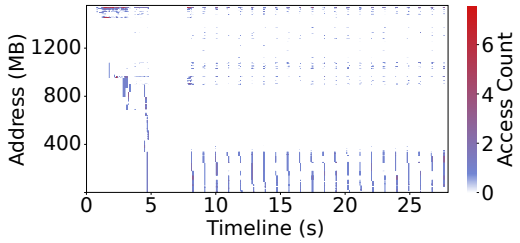


Figure 6. The *Access bit* scans of memory pages of ML inference benchmark BERT.

segment arduous. Besides, numerous hot pages persist during keep-alive, significantly reducing the memory efficiency.

3.3 Segment-3: Memory of Function Execution

Memory in this segment consists of temporary variables and states during function execution. Since each invocation request usually relies on the outside inputs, the memory footprint of this segment varies as requests change.

We select FunctionBench [34] and conduct an empirical study. We run these functions in independent processes, and then check the memory usage of each process via Linux Control Group (Cgroup) to trace the memory footprint during function execution. We find this segment only exists during the execution stage. Any temporary variables allocated are freed as soon as the execution completes, i.e., these pages do not reside during the keep-alive stage.

Takeaway. Memory pages allocated at the execution segment are immediately released after each execution completes. Given that function execution in serverless scenario typically takes only a few hundred milliseconds [57, 69], the benefit of page offloading in this segment is marginal.

3.4 Implications and Challenges

The study of the three segments implies that there exist memory offloading opportunities. To exploit our proposed segment-based memory layout to offload memory in serverless, we encounters three primary challenges.

Challenge-1: Reorganizing the memory structure of serverless containers. To efficiently exploit the memory pool architecture, we should take the segment-based memory structure characteristics of FaaS into consideration. Nevertheless, the current memory subsystem is not expected to identify the different pages attributed to the corresponding segment, and the pages generated across different segments are also not discriminated in the operating system.

Challenge-2: Tailoring cold page offloading mechanisms for different segments. In accordance with the aforementioned page identification and segregation, different mechanisms for identifying cold and hot pages, along with corresponding policies for offloading cold pages, need to be tailored for each segment. Crucially, these mechanisms and policies should be carefully designed to minimize the penalties linked with the revisiting of remote memory pages.

Challenge-3: A methodology of hot pages management during keep-alive to reduce memory footprint. After offloading cold pages, a high proportion of hot ones are still residing during the long-lasting keep-alive and waiting for future requests, leading to a significant reduction in local memory efficiency. Under memory pool architecture, the keep-alive strategy for serverless computing should be redesigned, and the hot pages could also be offloaded to the remote memory.

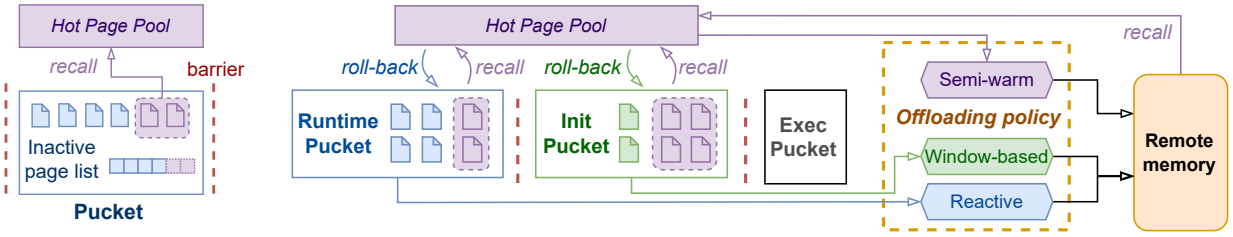
4 Overview of FaaSMem Mechanism

Given the unique offloading preferences for the memory in different segments, it is imperative to identify the cold memory pages in different segments and apply distinct offloading policies for each. However, the kernel memory subsystem treats the entire container as a single Cgroup, where many pages in different segments share the same virtual address space and are being managed in a unified manner. Since current sample-based cold-page identification methods are actually recording the virtual memory addresses, we cannot trace-back from these pages to correspond segments.

FaaSMem introduces the Pucket to segregate memory pages in different segments, with each Pucket being partitioned by the Time Barrier. Figure 7(a) shows the design of Pucket. There is an inactive page list maintained in each Pucket, and a shared hot page pool among different Puckets.

Under serverless paradigm, FaaSMem creates three Puckets to segregate memory pages into three different segments. Figure 7(b) shows the organization of Puckets in serverless. Specifically, upon the successful loading of the container runtime, FaaSMem inserts a Runtime-Init time barrier into the LRU page list of its Cgroup. Pages generated before this Runtime-Init time barrier are segregated and assigned to the *Runtime Pucket*. Similarly, after the initialization of a function container, an Init-Execution time barrier is introduced, which produces *Init Pucket* and *Execution Pucket*.

All pages generated within a Pucket are initially included in its inactive list. As explained in the memory layout, the offload benefit of the execution segment is marginal. Therefore,



(a) Pucket design. **(b)** Segment-based Pucket in FaaSMem and tailored offloading policies for different Puckets.

Figure 7. Overview of FaaSMem. FaaSMem uses time barrier to segregate pages into different Puckets. It reactively offloads the Runtime Pucket, applies window-based policy to offload Init Pucket, and proposes the semi-warm period to manage hot pages.

promote once accessed

FaaSMem does not monitor the page access of the Execution Pucket. To identify cold pages for Runtime Pucket and Init Pucket, when a page is revisited, it is moved to the hot page pool. As illustrated in Figure 7(a), when purple-colored pages initially assigned to the Pucket are revisited, they are removed from the inactive page list and added to the hot page pool.

The remaining inactive pages within these Puckets are considered as candidates for cold pages that can be safely offloaded. A reactive offloading policy is employed for the Runtime Pucket, and a request window-based offloading policy is applied to the Init Pucket. FaaSMem fetches the remote pages once accessed, and periodically re-evaluates the activity of pages in the hot page pool through page rollback. Moreover, to manage numerous hot pages during keep-alive, FaaSMem introduces a semi-warm period to gradually offload them while ensuring the 95%-ile latency. Subsequent sections discuss these customized policies, addressing when to offload inactive pages of different Puckets and how to handle recalled pages in the hot page pool.

5 Offloading Cold Pages for Puckets

The inactive cold pages of the Puckets and the hot page pool change as time goes by, while pages are moved among them. FaaSMem proposes a segment-wise offloading policies towards different memory Puckets.

5.1 Reactive Offload for Runtime Pucket

The runtime part of a container interacts with the controller and provides the environment for functions. Therefore, the memory of the runtime segment is independent of the function code executed. It motivates us that if a page created at the Runtime Pucket is not accessed during the init stage and the first request execution, it will hardly be accessed during later request execution.

Experiment in Figure 8 verifies that few pages are recalled during subsequent requests after the first execution. In the experiment, we use the runtime of OpenWhisk, which consists a Flask-based action proxy, and we select all 11 benchmarks from the evaluation section. After the offloading of the Runtime Pucket once the first request completes, we find that

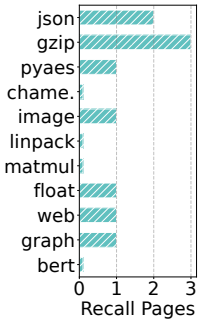


Figure 8. Recalls of Runtime Pucket.

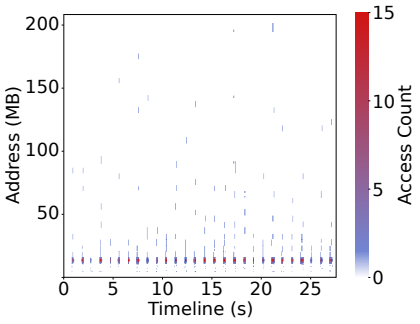


Figure 9. The *Access bit* scans of memory pages of benchmark web.

the subsequent requests hardly results in the page recalling from the Runtime Pucket.

Therefore, when the container completes the first request, pages left in the inactive list of the Runtime Pucket can be offloaded to remote memory. Once the the first request of a launching container is completed, an message is generated and passed to the serverless controller. FaaSMem will then offload all inactive pages of the Runtime Pucket.

5.2 Window-based Offload for Init Pucket

For pages created in the Init Pucket, we can still keep track of pages that remain unaccessed by subsequent requests, during which any access to init segment's page will move it to the hot page pool. However, aggressive offloading for these inactive cold pages like Runtime Pucket does is of high risk because we cannot confidently guarantee that a page unaccessed by one request will not be accessed by further ones. This means that the serverless system should lazy-offload inactive pages of the Init Pucket until more requests have been executed.

Given that the preference for offloading the Init Pucket depends on the characteristics of the function, determining an adaptive request-window for different functions while reliably identifying cold pages is another non-trivial issue. For example, in ML model inference of Figure 6, the number of pages accessed per request tends to stable, so it is sufficient

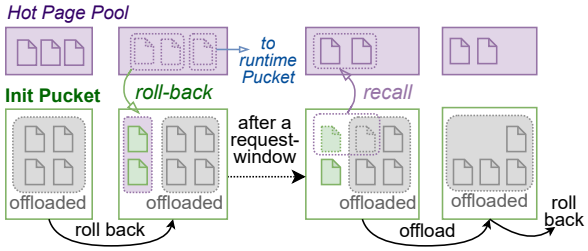


Figure 10. Periodic pages rollback of Pucket.

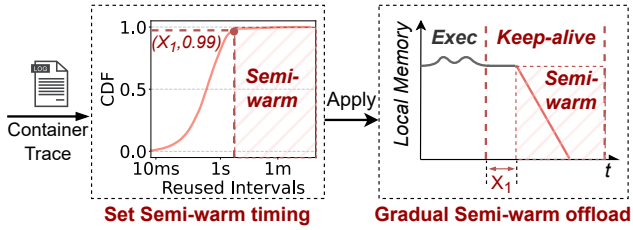


Figure 11. Overview of the semi-warm design, including the determination of semi-warm timing and the gradual memory offloading process upon a container entering the semi-warm.

to offload the Init Pucket after one request-window. Conversely, web services may involve different cached HTML pages accessed by each request. Figure 9 shows the access scans of web, while each vertical column represents a request. Within a request, there consists of multiple bars, representing different cached HTML pages. Therefore, it is necessary to offer a more prudent choice of a larger request-window, such as 20.

To determine the appropriate request window size, FaaSMem monitors the descent gradient of remaining pages within the Init Pucket relative to the number of executed requests. As more function requests have been completed, the number of pages in the Init Pucket decreases. When the descent gradient tends to zero, which indicates the number of pages in the Init Pucket will probably not change dramatically, FaaSMem identifies this corresponding point as the final request-window size [36, 45, 49]. At this stage, FaaSMem systematically offloads all pages in the Init Pucket. Employing this approach enables FaaSMem to lazy-offload the Init Pucket after a carefully determined request-window.

5.3 Periodic Page Rolling Back to Puckets

As one container executes more requests, pages of the Runtime Pucket and the Init Pucket that were previously offloaded may be gradually and slowly recalled to the hot page pool. FaaSMem needs to periodically re-evaluate the activeness of the pages in the hot page pool.

Figure 10 illustrates the re-evaluation procedure. FaaSMem utilizes insights gained from profiling the request-window through the Init Pucket. Following the execution of a request-window of the same size, FaaSMem rolls back all pages in the hot page pool to their respective original Puckets. Subsequently, FaaSMem awaits another request-window, during which any access to a page in the inactive list prompts its return to the hot page pool. Pages that remain in the inactive lists of the Runtime Pucket and the Init Pucket after this request-window are then offloaded. Following this, FaaSMem initiates a new cycle of rollback. To prevent frequent rollbacks, FaaSMem introduces a time parameter, denoted as t, requiring an interval of more than t from the last rollback before another can be performed. The rollback is triggered by the system only when the time-window and request-window are both satisfied. The evaluation section will discuss how to select an appropriate t to control the overhead of rollbacks within a manageable range.

In summary, FaaSMem adopts a reactive offloading approach for the Runtime Pucket upon completion of the first request. The determination of the request-window size for delaying the offload of the Init Pucket is made through real-time monitoring. Subsequently, FaaSMem continually re-evaluates the hot page pool and offloads inactive pages.

6 Offloading Hot Pages with Semi-warm

The segment-wise memory offloading policies for Puckets effectively identify and offload cold pages to the remote memory pool. However, large-scale real-world applications, such as web services and model inference, have numerous hot pages during container keep-alive. This motivates us to design an exclusive offload policy tailored for hot pages during container keep-alive.

6.1 The Semi-warm Period

As for the hot memory pages, the offloading behavior is no longer non-destructive, and the pages to-be-offloaded are exactly the ones to be accessed by subsequent requests. However, as discussed in Section 2.1, under the serverless context, the memory will be idle for a long time while taking up enormous resources, let alone the fact that the container might not be used after a request. Therefore, although it is evident that further offloading hot memory pages leads to a significant increase in response latency for the next request, we still needs to do this. The previous current page offloading policy only focus on the historical page access pattern, without the consideration of near-future request coming.

FaaSMem introduces a new period for keep-alive containers under memory pool architecture, the **semi-warm** period. It is designed to ensure that offloading hot pages has minimal impact on future requests. Figure 11 shows the overview of semi-warm. FaaSMem sets the semi-warm start timing based on the invocation characteristics of each function. It gathers the historical invocation trace of each function, and then analyzes the distribution of container reused intervals.

Here, the container reused intervals is a collection of how long each container stays keep-alive until receiving the next request. Based on it, FaaSMem generates a CDF depicting the frequency of container reused intervals. For example, the left part shows the corresponding CDF graph of one anonymous function of Azure Trace.

To guarantee the 95%-ile of end-to-end latency, FaaSMem uses pessimistic estimation that adopt the 99%-ile of container reused interval distribution as the semi-warm start timing. During semi-warm period, hot pages are further offloaded to the remote pool. In this way, serverless containers keep all hot pages for 95% of the requests. The rest 5% will experience a semi-warm start with hot page recalling.

6.2 Gradual Offloading and Bandwidth Control

FaaSMem adopts a gradual memory offload policy for semi-warm containers with bandwidth control. Specifically, FaaSMem would not tend to offload all memory immediately when entering the semi-warm period, but perform a gradual offloading as time goes by. Once a new request arrives, the offloading procedure will stop. The right part of Figure 11 shows the local memory changes during semi-warm. A gradual offloading approach is particularly effective for fluctuating request invocation patterns.

If a compute node generates numerous containers in response to a sudden surge, these containers will become semi-warm simultaneously. However, offloading simultaneously with a significant amount of memory could result in bandwidth contention. The gradual approach can also help distribute the offloading pressure evenly over time.

Specifically, FaaSMem provides two approaches to gradually offload memory during semi-warm: percentile-based (e.g., 1%/s) and amount-based (e.g., 1MB/s). Cloud providers can profile each function and then select an appropriate offloading approach. For example, large functions adopt the percentile-based approach to complete the offloading, while small functions follow the amount-based approach to offload memory faster. Besides, FaaSMem also monitors the global remote bandwidth in real-time, and uniformly reduces the offload speed of all containers when the bandwidth approaches the limit under extreme cases.

7 Implementation of FaaSMem

We leverage the Multi-gen LRU (MGLRU)[1] [9] feature in the Linux kernel to implement FaaSMem. MGLRU tracks pages in each LRU generation, and move them from older to newer generations when they are accessed [10]. We then implement a Pucket as a generation of the MGLRU. Meanwhile, MGLRU offers a system interface to allow for creating new generations, and inserting a time barrier or rolling back hot pages corresponds to creating a new generation.

[1]The MGLRU feature has been integrated into the kernel since Linux 6.1.

To support transparent memory pooling, we have ported Fastswap[2] [15], which modifies the page-out and page-in data paths of the Linux kernel swap mechanism to support remote memory offloading and retrieval. When offloading a page, the kernel swaps the page to the remote memory through Fastswap's RDMA interface; when accessing a remote page, a pagefault is triggered, invoking Fastswap to fetch the remote page.

8 Experimental Evaluation

In this section, we first evaluate FaaSMem in reducing local memory footprint under real-world traces. Then, we break down the effectiveness of the mechanisms in FaaSMem, and analyze the applicability and the overhead. Last, we study the production deployment density improvement by FaaSMem.

8.1 Experimental Setup

Hardware and Software Setup. We evaluate FaaSMem on a 2-node cluster of the CloudLab [23]. We use one node as the compute node, and the other node as the memory pool. Each node has 16 cores, 64 GB memory, and is equipped with a Mellanox FDR CX3 card. The nodes are connected through a Mellanox SX6036G switch that offers 56 Gbps Infiniband network. FaaSMem runs on Ubuntu 22.04 with a modified Linux 6.1 kernel. The container runtime is based on OpenWhisk [2], and is set with a 10-minute keep-alive timeout.

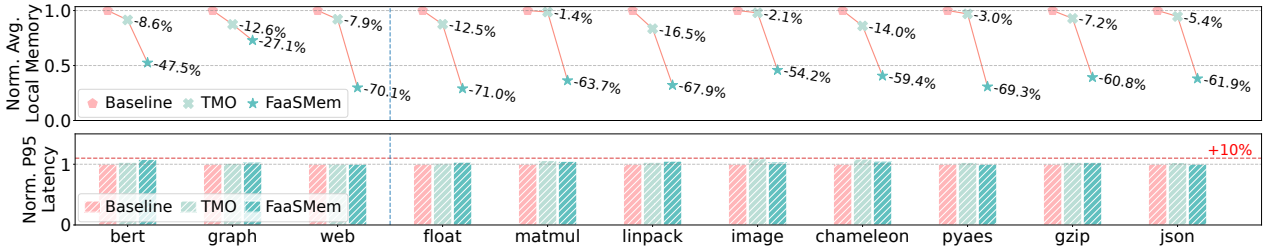
Benchmarks. We use 11 benchmarks in total, including eight micro-benchmarks from FunctionBench[3] [34]: float, matmul, linpack, image, chameleon, pyaes, gzip, and json. Besides, we select three real-world applications: *BERT-based ML Inference* (*Bert*) [20], *Graph Breadth-first Search* (*Graph*) [1], and *HTML Web Service* (*Web*) [4].

For eight micro-benchmarks, we assign the popular 0.1-core settings [13]. To fulfill the user-facing latency requirements of the real-world applications (around 200 ms), we assign 1-core, 0.5-core, 0.2-core for Bert, Graph, and Web, respectively.

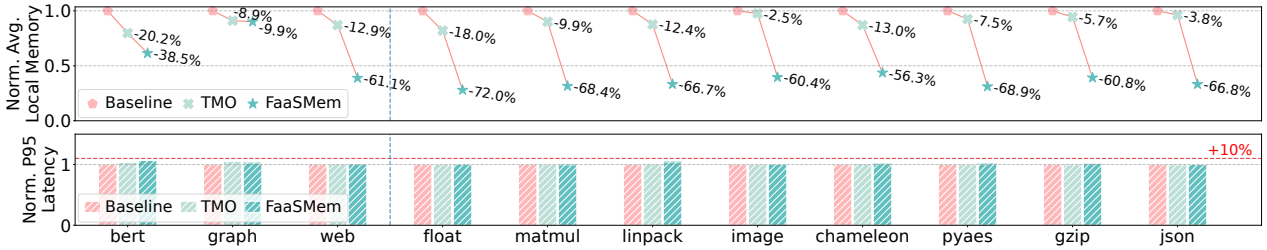
Invocation Patterns. We initiate all invocation requests on the compute node to avoid the latency effects between the client and the server. For the micro-benchmarks, we follow the pre-defined patterns from FunctionBench. For Bert, the input text is a random sentence from a long novel, which makes it more challenging as different requests may access different nodes in the neural network. For Graph and Web, the user's request includes a specific number *idx* to specify the start node for Graph or the specific HTML page for Web. The *idx* obeys the Pareto distribution, which can better represent the real-world scenario.

[2]Fastswap was originally developed on Linux 4.11, and we have ported it to Linux 6.1 to make it compatible with the MGLRU.

[3]We have replaced the model-related benchmarks of FunctionBench with BERT to better represent the real-world scenario. Meanwhile, a few benchmarks are omitted due to workflow requirements or runtime conflicts.



(a) High load



(b) Low load

Figure 12. The response latency and memory usage of the benchmarks under the high-load trace and the low-load trace.

Baselines. We treat one of the variants of FaaSMem without memory offloading as the baseline. In addition, we also compare FaaSMem against TMO [65], which is a state-of-the-art memory offloading system. As discussed in section 2.2, we do not consider DAMON [7] since it incurs severe performance degradation.

8.2 Azure Trace Evaluation

We use the Azure Functions Trace 2021 [69] to demonstrate the memory savings and the corresponding performance impact of FaaSMem. We divide the anonymous function traces into two types, the high-load and the low-load, according to their average daily invocations. As each trace includes a detailed firing timestamp of each invocation, we map them to our benchmarks, and compare FaaSMem's performance towards the baseline and TMO among different traces.

8.2.1 Performance of Diverse Benchmarks. To investigate the performance of FaaSMem under diverse benchmarks, we randomly choose a high-load trace and a low-load trace of a 1-hour-period.

Figure 12 shows the normalized average local memory usage and the 95%-ile latency of all benchmarks with the high-load and the low-load trace. Overall, FaaSMem reduces a significant amount of local memory footprint while keeping the same level of function performance. It reduces the local memory usage by 27.1% to 71.0% under high load, and by 9.9% to 72.0% under low load. Compared with TMO, FaaSMem's offloading effort is up to 45.5X higher, while the tail latency maintains the same level. This is because TMO offloads memory in a conservative step-by-step manner, whereas FaaSMem quickly identifies cold pages based on Pucket and makes containers semi-warm during keep-alive.

For eight micro-benchmarks, FaaSMem offloads at least 50% of memory, since they all have very little memory in the init segment and the cold pages in the runtime segment take up a significant portion of their total memory footprint, which can be offloaded by Pucket efficiently. Conversely, for three applications, their init segment has much more memory than the runtime segment. And the page access pattern in this segment also varies from these benchmarks. Benchmark Web gains the highest offloading ratio, as the pages accessed obeys the Pareto distribution, while Graph exhibits poor offloading ratio, because each request performs a complete traversal of the entire graph.

Meanwhile, under the high-load trace, FaaSMem usually offloads more memory for three applications, compared to the low-load trace. This is because the invocation pattern of high-load trace often exhibits a sudden increase and decrease, and it is likely to have more keep-alive containers stranded in memory due to burst, where FaaSMem's semi-warm period is tailored precisely for this situation. As for TMO, it reacts slowly to these temporary bursty containers. Take Bert in Figure 12a as an example, during each request, there are average 1.08 MB data offloaded from local and 0.65 MB data recalled from remote in FaaSMem. The corresponding results for TMO are only 0.05 MB and 0.0004 MB. This also indicates that FaaSMem performs better when the load is higher and is more helpful for situations when nodes are strapped for memory resources due to high load.

8.2.2 Performance of Diverse Workloads. To investigate whether FaaSMem is applicable to various workloads,

Table 1. 95%-ile response latency and average memory usage of applications under diverse traces ("Lat." stands for latency).

(a) Bert

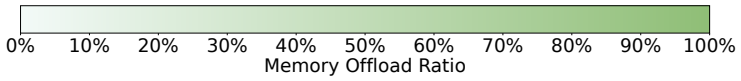
ID	Baseline		TMO		FaaSMem	
	Lat.	Mem	Lat.	Mem	Lat.	Mem
1	0.14s	2.64G	0.16s	2.56G	0.15s	1.62G
2	0.14s	1.55G	0.16s	1.51G	0.15s	0.92G
3	0.14s	2.09G	0.15s	1.90G	0.15s	1.43G
4	0.14s	2.04G	0.17s	1.96G	0.15s	1.43G
5	9.24s	8.81G	10.4s	8.58G	10.1s	5.60G
6	0.13s	1.30G	0.16s	1.26G	0.15s	0.81G

(b) Graph

Baseline		TMO		FaaSMem	
Lat.	Mem	Lat.	Mem	Lat.	Mem
0.26s	0.77G	0.27s	0.73G	0.27s	0.59G
0.25s	0.44G	0.25s	0.41G	0.25s	0.33G
0.25s	0.58G	0.25s	0.55G	0.26s	0.38G
0.24s	0.67G	0.26s	0.62G	0.25s	0.58G
9.95s	2.70G	10.8s	2.47G	10.6s	2.31G
0.23s	0.35G	0.25s	0.33G	0.25s	0.28G

(c) Web

Baseline		TMO		FaaSMem	
Lat.	Mem	Lat.	Mem	Lat.	Mem
0.16s	0.83G	0.17s	0.75G	0.17s	0.19G
0.14s	0.57G	0.15s	0.56G	0.15s	0.13G
0.12s	0.71G	0.14s	0.69G	0.13s	0.15G
0.12s	0.81G	0.12s	0.73G	0.13s	0.38G
0.45s	3.62G	0.49s	3.25G	0.46s	1.16G
0.12s	0.48G	0.12s	0.42G	0.12s	0.10G



we select six high-load traces and apply them to each of the applications. Table 1 shows the result with diverse traces.

In the table, FaaSMem's corresponding color blocks are typically much darker than TMO, indicating that more memory is offloaded. FaaSMem works best for Web because its init segment has remarkable cold pages, allowing for efficient offloading via Pucket, as previously discussed in Section 8.2.1.

Among these traces, there are some anomalies. For ID-5, the tail latency of three applications severely increases in baseline, especially for Bert and Graph. This is because ID-5 exhibits an extreme short-term surge, resulting in a large number of congested requests suffering from cold startup. However, even in this case, FaaSMem is still able to reduce memory usage by 14.4% to 68.0% and keep the tail-latency degradation consistent with the baseline.

FaaSMem is effective while maintaining the same level of performance under diverse traces.

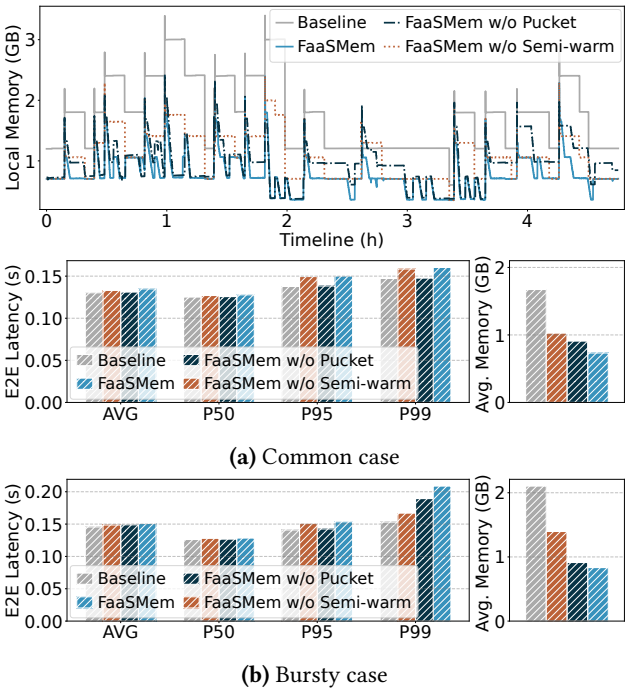
8.3 Ablation Experiments

In this subsection, we conduct experiments to demonstrate the effectiveness of each component in FaaSMem. Figure 13 shows the results when disabling each component of FaaSMem of benchmark Bert under two high-load traces.

8.3.1 Common Case. We first focus on the common case trace of Figure 13a to study the effect of each component by tracing the memory footprint timeline.

The effectiveness of Pucket. Without Pucket, the memory footprint is significantly higher than that with Pucket. This is because all pages won't be offloaded until semi-warm. For containers that never enter the semi-warm period, their cold pages reside in local, while Pucket can efficiently offload them at an early stage, resulting in higher memory usage.

Specifically, the average memory usage of FaaSMem decreases by 19.3% compared to that with Pucket disabled. However, without the Pucket, the 95%-ile latency of FaaSMem decreases by 9.2%, which indicates that Pucket causes the latency increase. But such increase is within our expectation, as the Pucket is designed to suffer a small number of remote memory accesses as a tradeoff for offloading cold pages.



(a) Common case

(b) Bursty case

Figure 13. The latency and memory usage on whether to disable Pucket and Semi-warm of benchmark Bert.

The effectiveness of Semi-warm. Without the semi-warm period, the memory footprint timeline is higher than that with semi-warm period, but parallel to the baseline's curve and exhibiting the same characteristics, i.e., it will suddenly decrease at some time. This arises from the fact that the container is removed from memory only when it exceeds the 10-minute keep-alive timeout, which results in a long waste. Instead, FaaSMem gradually offloads memory during semi-warm, rather than wait for timeout. This approach reduces the memory by 28.6%.

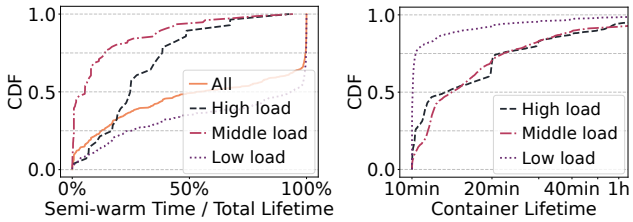


Figure 14. The semi-warm benefits across container lifetime.

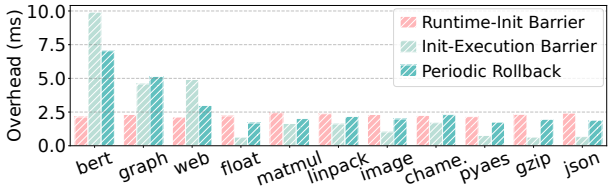


Figure 15. Overhead of time barrier and rollback in Pucket.

8.3.2 Bursty Case.

Figure 13b demonstrates the latency and memory usage of Bert under a much more bursty trace.

Effect of Semi-warm on Pucket. In this case, we find that the effect of semi-warm partly outweights the Pucket, i.e., the local memory without Pucket is close to the enabled. This is because the bursty case results in numerous keep-alive containers whose cold pages can also be offloaded during the semi-warm, just at a later time compared to Pucket.

The QoS guarantee via pessimistic estimation. Theoretically, by choosing a 99%-ile container reused intervals, FaaSMem can guarantee the e2e latency of 99% requests. However, under the bursty case, the 99%-ile latency of FaaSMem increases by 25.0% compared to that without semi-warm, while this phenomenon does not appear under the common case. This exception comes from the fact the data of container reused intervals does not take the cold-start impact into account. Under the bursty load, the cold-start incidents increase, resulting in the collected 99%-ile container reused intervals being smaller than the ideal value, making the semi-warm policy misestimate the 99%-ile latency.

Therefore, in order to guarantee 95% of the requests, we should pessimistically choose the a higher SLA, i.e., the 99%-ile. Thus, there is also an opportunity to select a more precise timing, by taking the cold-start incidents into account, in order to further reduce the keep-alive memory footprint.

8.4 Applicability of Semi-warm

To study the applicability of semi-warm across diverse workloads, we categorize the total 424 functions in Azure trace into three categories - high ($\geq$ 512), medium, and low ($\leq$ 64) - based on the average number of invocations per day, and investigate the time proportion of semi-warm period.

Figure 14 shows the distribution of the time proportion of semi-warm and the container lifetime across diverse functions. A higher proportion of semi-warm indicates that this function will save more memory with the help of semi-warm.

As a result, semi-warm period takes up more than one-half of container lifetime among 50% functions. This indicates that the mechanism of semi-warm is widely applicable for different invocation patterns. On top of that, we can find that it is more effective for both high-load and low-load functions. This is because the proportion of short-lived containers is high under both condition and short-lived containers will magnify the effect of semi-warm. For the low-load functions, a large scale of containers cannot be reused after created, while for the high-load functions, short-term surge creates a large number of short-lived containers.

Conversely, the middle-load functions tend to have a stable invocation pattern that results in most containers being lived for a long time, and therefore have less chance of enjoying semi-warm period.

8.5 Overhead of Puckets

Figure 15 shows the procedure overhead of Pucket.

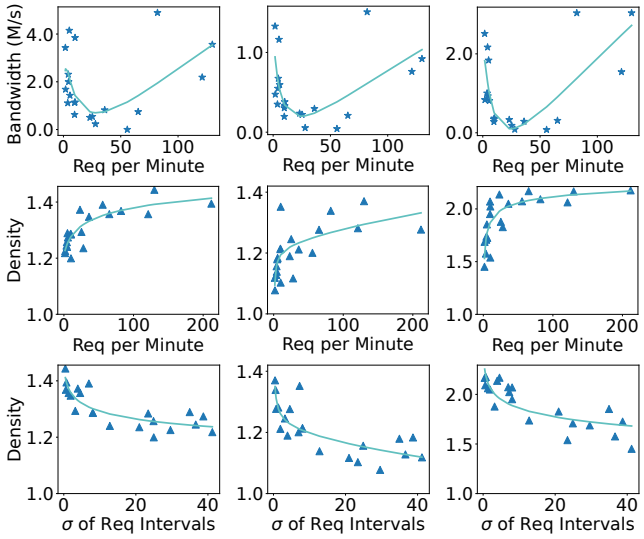
Time barrier. The process of the insertion of time barrier is blocked to accurately segregate the memory pages into multiple segments. For eight micro-benchmarks, the insertion of the runtime-init barrier and init-execution barrier takes less than 2.5 ms. For three applications, the runtime-init barrier also takes less than 2.5 ms, but it takes 10, 5, and 5 ms for Bert, Graph, and Web to insert the init-execution barrier, respectively. The overhead increases due to the increased memory footprint of these applications during initialization. Overall, as the insertions only occur during cold-start, the overhead is negligible compared to the total cold-start time.

Periodic rollback. For all benchmarks, the time of a rollback is less than 7.5 ms, and we recommend that each round should be performed at the interval of at least 10 s, which ensures the overall overhead is less than 0.1%.

8.6 Production Density Evaluation

In production environment, each container has a predefined memory quota, which is used to schedule and deploy. Therefore, we consider the offload amount as a reducible amount of the quota. For example, if a 128 MB-quota container can offload 28 MB via FaaSMem, we then treat the new quota under FaaSMem is 100 MB. Considering the actual memory usage, we assign 1280, 256, 384 MB as the original memory quota for the three applications, Bert, Graph, and Web, respectively. Then, we randomly select 20 Azure traces to investigate the relevant factors affecting the deployment density. Figure 16 shows the remote bandwidth consumption and the production density under different traces of the three applications.

First, we take a look at the bandwidth consumption. As can be seen from the figure, the remote bandwidth increases near linearly as the request load increases. However, the



(a) Bert **(b)** Graph **(c)** Web

Figure 16. The remote bandwidth consumption and the estimated density improvement under diverse traces.

bandwidth also increases when the load is extremely low. This is because the low-frequency invocation characteristics could lead to an earlier start timing of semi-warm period during keep-alive.

We then focus on the density improvement. For Bert, Graph, and Web, FaaSMem offers up to 1.4X, 1.4X, and 2.2X density improvement, respectively. Each scatter in the figure represents a particular trace. We then investigate two potentially relevant factors, request loads and request intervals.

We find that the density improvement is positively correlated with the request loads, which matches the trend in Section 8.2.1. As the number of requests per minute increases, FaaSMem gains higher density, since Pucket can gather more requests within a shorter time to perform the offloading. Meanwhile, higher loads are more likely to incur burst with more semi-warm benefits, as discussed previously.

Besides, we also find a negative corelation with the standard deviation (σ) of request intervals. This is because when σ of the request intervals increases, the distribution of request intervals is more scattered. In this case, the semi-warm period starts later, leading to more pages to reside for a longer time in the hot page pool outside the Pucket.

9 Discussion

Bandwidth provision and latency analysis under large-scale deployment. In all our experiments, each container consumes no more than 0.82 MB/s remote bandwidth in average. According to the production data [41], a single compute node with 384 GB memory holds up to 2500 containers. As FaaSMem offers up to 2X density improvement, the bandwidth required of a node holding 5000 containers is around 32 Gbps. Given that memory pools are often configured at rack level [40] to avoid the time-consuming cross-rack communication, only the compute nodes in a rack (around 10 nodes) can access the memory pool in the same rack. In this way, the memory pools work individually, and each requires an aggregated bandwidth of 320 Gbps for serving the 10 compute nodes within a rack. To this end, today's network adapter offers up to 400 Gbps RDMA network, and the bandwidth capacity can be further extended by using multiple network adapters. Meanwhile, there is little communication latency increase until the bandwidth is saturated [33, 61, 64]. Therefore, neither bandwidth nor latency will become a bottleneck under large-scale deployment.

Building memory pool. Considering that most of the existing serverless workloads are web applications [32, 70], the ratio of local to remote memory usage for web is between 1:0.5 and 1:1.1 under different traces. Therefore, we recommend a local-remote ratio around 1:0.8. For a rack consisting 10 compute nodes with 384 GB each, the correspond memory node should offer 3 TB memory. As discussed in Section 2.2, the memory node can reuse the old memory with negligible cost. In this way, the memory pool architecture could lead to a 44% reduction in DRAM cost.

CXL-based memory pool. CXL is an emerging approach that offers higher bandwidth with lower latency compared to the RDMA-based memory pool [27], which FaaSMem's mechanism can also be applied, as it is not limited to a certain implementation of memory pool. However, there is no production-available hardware support for CXL-based memory pool currently. Offloading to SSDs is another possible solution. However, SSDs have limited write durability, e.g. Meta needs to limit their write speeds to less than 1 MB/s [65]. It does not meet the bandwidth demands of FaaSMem.

Hardware-based page sampling method. Many works have focused on reducing the overhead of cold-page identification by using hardware-level sampling [24, 39, 47], e.g., Intel's Precise Event Based Sampling [14], and this approach can also be integrated into FaaSMem. Although FaaSMem's Pucket is based on page table tracing, it already has negligible overhead by the window-based rollback and offloading.

Memory sharing in serverless. To reduce the memory usage, many works [41, 55] have also explored to share memory in serverless computing. For instance, FAASM [59] shares the runtime across different containers of one function. However, as for the init segment and the exec segment, it can only mitigate the duplicate state variables and requires an intrusive user-code reorganization. By combining these techniques, FaaSMem can further reduce memory footprint.

Limitations and future work. Our current evaluations are conducted on a 2-node cluster, which may not be able to cover all situations that would occur in the large-scale deployment. Considering that different nodes may have different loads, memory pooling could potentially yield further benefits for nodes that are memory stranded. Meanwhile,

under extreme load-imbalance, there may not be enough local memory space to call back the semi-warm containers from remote and requires the rescheduling, which we treat as future work.

10 Related Work

Remote Memory Pool. There are two popular memory pool architectures. The first is swap-based, which memory is swapped to the remote node via Remote Direct Memory Access (RDMA). Infiniswap [28] maps the remote memory pool to a local file to support the remote memory swapping. Fastswap [15] modifies the Linux kernel to speed up the swapping process. Besides, many works [30, 46, 52, 62] have also explored and optimized this approach. The second is through Compute Express Link (CXL) [6], which requries specific hardware support, and some works [27, 67] have also explored to design the correspond hardware. These researches are orthogonal to FaaSMem, as FaaSMem focuses on tailoring the serverless paradigm under the memory pool architecture, while they focused on constructing the memory pool and improving the underlying performance.

Memory Offloading. Many works have explored the opportunity to offload memory. TMO [65] offloads memory step-by-step and proposes the Pressure Stall Information to monitor the corresponding slowdown. TPP [47] offers an OS-level mechanism to identify and place hot and cold pages between local and remote, while Pond [40] uses machine learning to develop the offload policy for VMs. Meanwhile, many other works [7, 24, 35, 39, 51, 53, 66] also focus on developing the offload policies. These efforts offer memory offloading solutions to long-term tasks, but their design is not applicable to serverless scenarios. Since serverless containers are transient, the periodic keep-alive stage can disrupt the traditional policy making.

Keep-alive Strategy. Many prior researches focused on adjusting the keep-alive strategy to save resources and alleviate cold-start in serverless computing [21, 44, 54, 57, 68]. For example, a hybrid histogram policy is proposed [57] to proactively pre-warm containers and set a lower keep-alive threshold. FaaSMem introduces a semi-warm period under memory pool architecture to reduce local memory usage, combining the above works can gain more benefits.

11 Conclusion

We propose FaaSMem, a memory offloading mechanism to improve the memory efficiency of serverless computing with memory pool architecture. FaaSMem brings a new solution to serverless memory inefficiencies. Specifically, it introduces Pucket to segregate different pages into three segments and develop the segment-wise offloading policies. The semi-warm period proposed in FaaSMem further efficiently offloads the hot pages during keep-alive stage. Experimental results through real-world traces show that FaaSMem can efficiently utilize the memory pool architecture to reduce local memory footprint with negligible performance degradation.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Rodrigo Fonseca, for their helpful comments and suggestions. This work is partially sponsored by the National Key Research and Development Program of China (2022YFB4501400) and National Natural Science Foundation of China (62232011, 62302302, 61832006). Quan Chen is the corresponding author.

A Artifact Appendix

A.1 Abstract

Our artifact includes the prototype implementation of FaaSMem, the Fastswap [15] ported to Linux 6.1, the 11 benchmarks, and the experiment workflow to run these benchmarks on CloudLab c6220 instances.

A.2 Artifact check-list (meta-information)

- **Program:** FaaSMem, Fastswap, Docker runtime, Python.
- **Data set:** Azure Functions Invocation Trace 2021 [69].
- **Run-time environment:** Ubuntu 22.04 with a modified Linux 6.1 kernel. Detailed dependencies and installations are listed and scripted in the artifact.
- **Hardware:** CloudLab c6220 instances (16 cores, 64 GB memory) connected through InfiniBand network (via Mellanox CX3 NIC).
- **Metrics:** End-to-end latency, memory usage, throughput.
- **Output:** JSON files and PDF graph.
- **Experiments:** Python scripts.
- **How much disk space required (approximately)?:** 80 GB.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours.
- **How much time is needed to complete experiments (approximately)?:** 11 days.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** 10.5281/zenodo.10791101

A.3 Description

A.3.1 How to access. The source code of FaaSMem are available and maintained on GitHub. Visit https://github.com/BarrinXu/FaaSMem for more information. You can also follow the `README.md` in the repository to perform the installation and the evaluation.

A.3.2 Hardware dependencies. FaaSMem requires two nodes, and we STRONGLY RECOMMEND to use the CloudLab c6220 instances, which have already satisfied the following requirements. Each node should have at least 16 cores, 64 GB memory and 80 GB free disk space. Besides, nodes should be connected through a RDMA-compatible network (e.g., InfiniBand).

A.3.3 Software dependencies. The artifact requires the experiment running Ubuntu 22.04 with a modifed Linux 6.1 kernel. FaaSMem is required to run in a Python 3.8 virtual environment. The benchmarks are required to run in a Docker. We provide an installation script to prepare all the software dependencies.

A.4 Installation

From now on, you have two nodes, a *compute node* and a *memory node*. Both nodes are running Ubuntu 22.04. MAKE SURE you are operating under root at anytime.

A.4.1 Install the modified Linux 6.1 kernel on the *compute node*. Enter `linux-6.1.55-fastswap` directory. Execute `bash install_kernel.sh`.

This script first installs the dependent tools needed to compile the kernel, then compiles and installs the kernel, and finally REBOOTS the machine.

Check: After rebooting, please check that the currently running kernel is `6.1.55-fastswap`, by executing `uname -r` in a terminal. If not, make sure the kernel is successfully installed and listed, and then specify this kernel version to boot the machine.

Attention: This script uses a kernel config file that is compatible with CloudLab c6220 instances, and is likely incompatible with other hardware instances, which may have to reconfigure the config file and modify the script.

A.4.2 Install the software dependencies on the *compute node*. Enter `FaaSMem-core` directory. Execute `bash install_software.sh`.

This script first installs Python 3.8 and creates a Python-3.8 virtual environment `venv` under `FaaSMem-core`. It then unzip the Azure invocation trace. Finally, it installs Docker, builds images for 11 benchmarks.

Check: After the script is complete, there should be a `venv` dir under `FaaSMem-core`. You can also find 12 new images by executing `docker images`.

A.4.3 Create a swapfile on the *compute node*. Enter `Fastswap` directory. Execute `bash create_swapfile.sh`.

This script will require an interactive input, which is the directory to create the swapfile. The directory MUST be under an ext4 partition, and has at least 32 GB free space.

Check: After the script is complete, there should be a 32 GB swapfile under the directory you specified.

A.4.4 Install the RDMA driver and compile Fastswap on the *compute node* and the *memory node*. First, on both nodes, enter `Fastswap` directory and execute `bash install_rdma.sh`. This script installs the RDMA driver and REBOOTS the machine.

Second, after rebooting, enter `Fastswap` directory. On the *compute node*, execute `bash build_fastswap_driver.sh`. On the *memory node*, execute `bash build_fastswap_server.sh`.

Check: Execute `ibstat` on each node. You should see from the output that the network adapter is up.

A.5 Experiment workflow

MAKE SURE you are operating under root at anytime.

A.5.1 Pre-experiment operations. The operations in this subsection need to be executed only once. DO NOT execute it multiple times UNLESS you have rebooted both the *compute node* and the *memory node*.

1. On the *compute node*, enter `Fastswap` directory, and execute `bash pre_experiment.sh`. This script will require an interactive input for the swapfile directory. It first disables the transparent huge page and the numa balancing. It then enable the swapfile created early before.
   Check: Execute `swapon -s`, you will see the only one swapfile, which is exactly the one created before.
2. Configure the IP address for the *compute node* and the *memory node*.
   First, you need to obtain the adapter interface name, by executing `ibstat` in the terminal. In CloudLab c6220 instance, the name should be `ibp130s0`.
   Second, check the interface name exists in the output by executing `ifconfig -a`.
   Third, assign IP address for the interface of each node. For instance, execute `ifconfig ibp130s0 192.168.125.x/24` for the *compute node*, execute `ifconfig ibp130s0 192.168.125.y/24` for the *memory node*. Here, x and y MUST be different, and are RECOMMENDED to follow the suffix of node ID in the CloudLab to avoid duplication with other users.
   Check: Execute `ifconfig ibp130s0` on each node to verify the IP address you configured. Execute `ping 192.168.125.y` on the compute node.
3. On the *memory node*, open a window using screen or tmux. Then enter `Fastswap` directory. Execute `bash run_fastswap_server.sh` to start the server for memory offloading. You can detach the window, DO NOT terminate it.
   Check: The output of the script contains "listening on port 50000".
4. On the *compute node*, enter `Fastswap` directory. Then execute `bash run_fastswap_driver.sh` to connect the *memory node*. The script will require two interactive input, the first is the compute node IP, and the second is the memory node IP.
   Check: On the compute node, execute `dmesg | grep ctrl`, you can see "ctrl is ready for reqs".
5. On the *compute node*, open three windows (Window-1, Window-2, Window-3) using screen or tmux. All of them MUST source the virtual environment `venv` under FaaSMem-core. DO NOT terminate them.

6. In Window-1, enter `FaaSMem-core/src/workflow_manager`, execute `python gateway.py`, which starts the gateway of FaaSMem to receive requests. In Window-2, enter `FaaSMem-core/src/workflow_manager`, execute `python test_server.py 127.0.0.1`, which starts the server of FaaSMem to process requests. DO NOT terminate them.

Attention: If any node suffers a reboot during the experiment, the remote connection is broken. You need to reboot both nodes and perform the operations listed in this subsection again.

A.5.2 Conduct each experiment. Each directory within `FaaSMem-core/test/AE` corresponds to one experiment. You should use Window-3, enter each directory, and execute `python test.py` to conduct the experiment. Once the experiment ends, a series of JSON files will be saved to the `result` subdirectory. Detailed scripts usage is introduced in the `README.md` under the repository.

A.6 Evaluation and expected results

Within each experiment directory, we prepare some draw scripts `draw*.py`. You should also source the virtual environment `venv` under `FaaSMem-core` before executing them. Then, you can enter each experiment directory, and execute `python draw*.py` to visualize the results. Each script will save a PDF format graph in the current directory, except that one script will output a table in the terminal. The graphs and tables generated can be compared with those in the paper.

References

[1] Amazon neptune serverless – amazon web services. https://aws.amazon.com/neptune/serverless/, 2023.

[2] Apache openwhisk is a serverless, open source cloud platform. https://openwhisk.apache.org/, 2023.

[3] Aws lambda documentation. https://docs.aws.amazon.com/lambda/index.html, 2023.

[4] Build your first serverless web app – amazon web services. https://aws.amazon.com/serverless/build-a-web-app/, 2023.

[5] Cloud functions. https://cloud.google.com/functions, 2023.

[6] Compute express link. https://www.computeexpresslink.org/, 2023.

[7] Damon: Data access monitor. https://www.kernel.org/doc/html/latest/admin-guide/mm/damon/, 2023.

[8] Memory management — the linux kernel documentation. https://docs.kernel.org/admin-guide/mm/index.html, 2023.

[9] Multi-gen lru — the linux kernel documentation. https://docs.kernel.org/admin-guide/mm/multigen_lru.html, 2023.

[10] Multi-generational lru: the next generation. https://lwn.net/Articles/856931/, 2023.

[11] Serverless computing - aws lambda - amazon web services. https://aws.amazon.com/lambda/, 2023.

[12] Serverless functions in computing - microsoft azure. https://azure.microsoft.com/en-us/products/functions/, 2023.

[13] The state of serverless 2020 | datadog. https://www.datadoghq.com/state-of-serverless-2020/, 2023.

[14] Soramichi Akiyama and Takahiro Hirofuchi. Quantitative evaluation of intel PEBS overhead for online system-noise analysis. In Torsten Hoefler and Kamil Iskra, editors, *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS@HPDC 2017, Washingon, DC, DC, USA, June 27 - 27, 2017*, pages 3:1–3:8. ACM, 2017.

[15] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.

[16] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In Julia Lawall and Dan Williams, editors, *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 315–328. USENIX Association, 2023.

[17] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. Polardb serverless: A cloud native database for disaggregated data centers. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2477–2489. ACM, 2021.

[18] Paul C. Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, 2019.

[19] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In Dilma Da Silva and Rüdiger Kapitza, editors, *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, pages 356–370. ACM, 2020.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[21] 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 James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 467–481. ACM, 2020.

[22] Zhuohui Duan, Haikun Liu, Haodi Lu, Xiaofei Liao, Hai Jin, Yu Zhang, and Bingsheng He. Gengar: An rdma-based distributed hybrid memory pool. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 92–103. IEEE, 2021.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.

[24] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David E. 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 Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 727–741. ACM,

2023.

[25] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 386–400. ACM, 2021.

[26] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[27] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, high-performance memory disaggregation with directcxl. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 287–294. USENIX Association, 2022.

[28] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[29] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: an analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service, IWQoS 2019, Phoenix, AZ, USA, June 24-25, 2019*, pages 39:1–39:10. ACM, 2019.

[30] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: a hardware-software co-designed disaggregated memory system. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 417–433. ACM, 2022.

[31] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 152–166. ACM, 2021.

[32] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[33] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[34] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, page 477. ACM, 2019.

[35] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 715–728. USENIX Association, 2021.

[36] Mohit Kumar, Subhash Chander Sharma, Anubhav Goel, and Santar Pal Singh. A comprehensive survey for scheduling techniques in cloud computing. *J. Netw. Comput. Appl.*, 143:1–33, 2019.

[37] H. Andrés Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 317–330. ACM, 2019.

[38] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: in-network memory management for disaggregated data centers. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.

[39] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. MEMTIS: efficient memory tiering with dynamic page classification and page size determination. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 17–34. ACM, 2023.

[40] 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 Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 574–587. ACM, 2023.

[41] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 53–68. USENIX Association, 2022.

[42] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 69–84. USENIX Association, 2022.

[43] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, 54(10s):220:1–220:34, 2022.

[44] Wes Lloyd, Minh Vu, Baojia Zhang, Olaf David, and George H. Leavesley. Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads. In Alan Sill and Josef Spillner, editors, *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018, Zurich, Switzerland, December 17-20, 2018*, pages 195–200. IEEE, 2018.

[45] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless dags. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 303–320. USENIX Association, 2022.

[46] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 843–857. USENIX Association, 2020.

[47] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. TPP: transparent page placement for cxl-enabled tiered-memory. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 742–755. ACM, 2023.

[48] M. Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino, editors, *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 405–410. IEEE Computer Society, 2017.

[49] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F. Wenisch. Parslo: A gradient descent-based approach for near-optimal partial SLO allotment in microservices. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 442–457. ACM, 2021.

[50] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In Christina Delimitrou and Dan R. K. Ports, editors, *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. USENIX Association, 2019.

[51] Seongjae Park, Yunjae Lee, and Heon Y. Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In Dejan S. Milojicic and Vinod Muthusamy, editors, *Proceedings of the 20th International Middleware Conference Industrial Track, Davis, CA, USA, December 9-13, 2019*, pages 1–7. ACM, 2019.

[52] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In Mahesh Balakrishnan and Manya Ghobadi, editors, *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 181–198. USENIX Association, 2023.

[53] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real NVM. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 392–407. ACM, 2021.

[54] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 753–767. ACM, 2022.

[55] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 714–729. ACM, 2022.

[56] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: A survey of opportunities, challenges, and applications. *ACM Comput. Surv.*, 54(11s):239:1–239:32, 2022.

[57] Mohammad Shahrad, Rodrigo Fonseca, Iñigo 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 Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020.

[58] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

[59] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 419–433. USENIX Association, 2020.

[60] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: enabling quality-of-service in serverless computing. In Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 311–327. ACM, 2020.

[61] Robert Underwood, Jason Anderson, and Amy W. Apon. Measuring network latency variation impacts to high performance computing application performance. In Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar, editors, *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, pages 68–79. ACM, 2018.

[62] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In Mahesh Balakrishnan and Manya Ghobadi, editors, *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 161–179. USENIX Association, 2023.

[63] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. Peeking behind the curtains of serverless platforms. In Haryadi S. Gunawi and Benjamin C. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146. USENIX Association, 2018.

[64] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[65] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: transparent memory offloading in datacenters. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 609–621. ACM, 2022.

[66] Zi Yan, Daniel Lustig, David W. Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 331–345. ACM, 2019.

[67] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-Yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with cxl-enabled ssds. In Julia Lawall and Dan Williams, editors, *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 601–617. USENIX Association, 2023.

[68] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS*

'22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022, pages 768–781. ACM, 2022.

[69] Yanqi Zhang, Iñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 724–739. ACM, 2021.

[70] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. Beehive: Sub-second elasticity for web services with semi-faas execution. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 74–87. ACM, 2023.