Towards Elastic Memory Allocation of Serverless Functions in Disaggregated Memory Systems

Achilleas Tzenetopoulos
National Technical University of Athens
Athens, Greece
atzenetopoulos@microlab.ntua.gr

Dimosthenis Masouros
National Technical University of Athens
Athens, Greece
dmasouros@microlab.ntua.gr

Dimitrios Soudris
National Technical University of Athens
Athens, Greece
dsoudris@microlab.ntua.gr

Sotirios Xydis
National Technical University of Athens
Athens, Greece
sxydis@microlab.ntua.gr

Abstract

Serverless computing is an emerging paradigm adopted by several cloud service providers. Resource underutilization is a major challenge in data centers today that even the fine-grained functions and scaling-out techniques that overcome the physical server bounds cannot resolve. Memory disaggregation has emerged as a solution leveraging remote memory pools to reduce resource fragmentation. However, existing work in serverless computing primarily exploits remote memory only as a low-cost idle state for functions, leaving its full potential untapped.

In this paper, we examine the potential of leveraging disaggregated memory for elastic memory allocations of serverless functions. By exploiting the latest Linux kernel's *weighted interleaving* feature, we first discuss the challenges and opportunities of memory disaggregation, showcasing how remote memory elasticity impacts the latency of different functions and payload sizes. By utilizing Huawei's invocation trace, we show that elastic memory allocation configurations, tailored to functions' behavior, can lead to up to 25% total memory footprint reductions with negligible performance degradation.

Keywords

Memory Disaggregation, Memory Tiering, Serverless Computing, Quality of Service

ACM Reference Format:
Achilleas Tzenetopoulos, Dimosthenis Masouros, Dimitrios Soudris, and Sotirios Xydis. 2025. Towards Elastic Memory Allocation of Serverless Functions in Disaggregated Memory Systems. In *4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25), March 30, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3723851.3723855

1 Introduction

Serverless computing [16] has emerged as a new paradigm in cloud computing, offering fine-grained resource allocation, seamless scalability, and a simplified programming model. Serverless platforms, such as AWS Lambda [5] and Google Cloud Functions [3], avail end-users by abstracting away the burden of infrastructure management, enabling developers to focus entirely on application logic. At the same time, providers benefit from the short-lived and resource-efficient nature of serverless functions, which can be packed into underutilized servers [35] to enhance overall infrastructure utilization and minimize fragmentation within data centers.

In today's serverless paradigm, in environments such as AWS Lambda [5], resource provisioning policies require users to specify static memory requirements for their functions, which are then subjected to rigid CPU-to-memory mappings. This approach limits the flexibility of resource allocation and fails to fully exploit the potential for finer-grained allocations, thereby contributing to inefficiencies and exacerbating memory shortages in modern data centers [21]. Moreover, serverless providers are compelled to pre-provision substantial amounts of costly local DRAM to deliver high performance and uphold the perception of infinite resources [35]. Notably, as shown in Section 2.2, 74% of this provisioned memory often remains unused, leading to increased operational costs and inefficient utilization.

Memory disaggregation represents a promising solution to address these challenges. In disaggregated memory systems, memory is decoupled from compute nodes and treated as a shared, elastic resource pool that can be dynamically allocated and redistributed based on workload demands. With recent advancements in hardware technologies [9, 25] providing remote memory access latencies to levels comparable to local DRAM, disaggregated memory is becoming a practical option for modern cloud infrastructures [22, 23].

Memory disaggregation aligns naturally with the ephemeral and resource-efficient nature of serverless computing. Serverless functions' life cycle typically involves three distinct phases: *sandbox initialization*, where containers are set up and back-end libraries are loaded; *execution*, during which function logic is performed; and the *keep-alive* phase, where containers remain idle until eviction. Notably, much of the memory actually allocated during these phases remains underutilized or not accessed at all, leading to up to 70% of memory inactivity throughout the function's lifetime [32].

Several works leverage remote memory to mitigate inefficiencies in serverless computing. General-purpose solutions [21, 31] provide transparent memory offloading by gradually migrating memory allocations to remote pools until service-level objectives are at risk of being violated. While effective for longer-running workloads, they often lack the fine-grained control needed for short-lived serverless functions. More specific efforts focus on the initialization phase, addressing cold-start challenges by utilizing remote memory pools through CXL and RDMA for sandbox reuse [14] or memory deduplication [26]. Similarly, for the keep-alive phase, prior

works use remote memory to offload inactive pages by dynamically identifying cold pages and balancing access penalties to maintain performance [32].

Despite these advancements, the execution phase remains largely overlooked in leveraging remote memory [32]. Although short-lived, this phase plays a critical role in shaping the infrastructure's overall resource efficiency. During this phase, functions often exhibit fluctuating memory demands, especially for memory-intensive workloads like AI inference, where large models require significant memory for intermediate computations, activations, and temporary data. These transient spikes in memory usage, followed by rapid deallocation, leave behind fragmented and underutilized memory across nodes. Overlooking the execution phase not only exacerbates resource fragmentation but also misses the opportunity to optimize workload bin-packing [16].

Our work. In this paper, we examine the potential of leveraging remote memory in the execution phase of serverless functions. By employing the *weighted interleaving* feature of Linux's latest kernel, we study the impact of elastic remote memory allocations on function latency under different payload sizes. We highlight the opportunities and pitfalls of leveraging remote memory for serverless workloads. Through extensive analysis, we identify critical trade-offs, such as the scale-out effects induced by remote memory overhead under high invocation rates, which can increase sandbox instantiations and temporary spikes in total memory usage. Utilizing Huawei's invocation trace [17], we demonstrate that leveraging remote memory can yield local memory footprint reductions ranging from 6% to 25%, with latency tolerances of 1% to 20%.

2 Background

2.1 The Journey of a Serverless Function

Serverless functions exhibit distinct phases throughout their lifetime, including the preparation of the execution environment, the actual function execution, and the cleanup of resources. This process can be divided into four key phases, as illustrated in Fig. 1, which are analyzed below:

Environment Initialization: This phase establishes the necessary execution environment, including required configurations and dependencies, to enable the function to execute within an isolated context. Functions are deployed inside lightweight *sandboxes*, such as containers or microVMs (e.g., Firecracker [4]) to provide secure, isolated execution. These sandboxes encapsulate the function's specified language runtime (e.g., Python, Node.js) along with any required dependencies. The sandboxes are in turn hosted on virtual machines (e.g., EC2 instances in AWS) managed by bare-metal (Type 1) hypervisors [5]. The memory footprint of this phase is affected by the runtime and libraries loaded into the sandbox, often accounting for a significant baseline memory allocation before function execution begins. When a function is invoked in the absence of a pre-initialized sandbox, a *cold start* occurs, which requires the environment to be set up from scratch, introducing additional latency (as shown in the first function invocation of Fig. 1). To mitigate this, several works have proposed optimizing the initialization phase by leveraging remote memory systems either by keeping shared remote memory pools for rapid restoration of function states [14, 18]

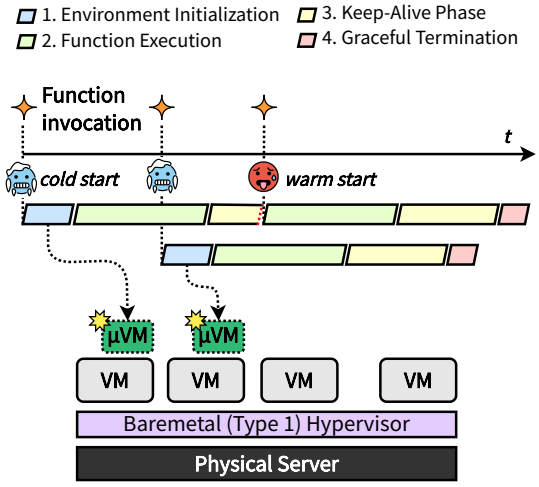


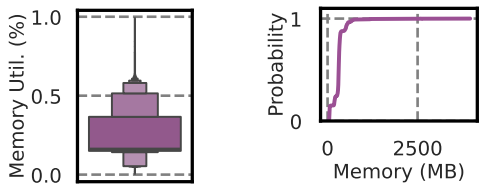
Figure 1: Overview of a serverless function's lifetime.

or by identifying and offloading infrequently accessed pages to the disaggregated memory [32].

Function Execution: This phase involves running the user-specified logic within the isolated environment. The function processes input data from an event source (e.g., object storage service) using the runtime and libraries loaded during initialization. During this phase, memory may be allocated for temporary computations or to interact with external storage. If an invocation arrives while a function instance is busy executing another, a new sandbox is usually instantiated to handle the request (second invocation in Fig. 1). Interestingly, this phase has been largely overlooked in the context of disaggregated memory environments. However, as serverless platforms increasingly host memory-hungry workloads [11, 34], the memory requirements of this phase often far exceed those of environment initialization. For example, in vision models [34], the batch size or the inter-layer calculations can increase the memory footprint significantly. We measured an x2.36 increased footprint during the execution phase for image classification using ResNet50 with batch size 1. Although the memory allocated during execution is de-allocated upon completion [32], the underlying VM must still retain sufficient memory until the keep-alive phase concludes to handle subsequent function invocations without failure. This can lead to resource fragmentation, where memory is reserved but underutilized.

Keep-Alive Phase: In this phase, the initialized environment remains idle but active for a predefined period after the function execution completes. This allows for faster subsequent invocations by avoiding the need for re-initialization, enabling *warm starts*. The memory footprint during this phase is substantial, as the runtime, libraries, and cached data remain loaded in memory [32]. While this improves responsiveness, it may result in underutilized memory when the function is not actively invoked.

Graceful Termination: This phase marks the final stage in the function's lifecycle, where the environment is decommissioned and resources are released. The memory footprint drops to zero as the platform reclaims resources, ensuring they are available for other workloads.



(a) Memory utilization. (b) Memory footprint.

Figure 2: Memory Utilization (%) and cumulative density function of memory footprint from Huawei's Trace [17].

2.2 Resource provisioning in Serverless Computing

Function-as-a-Service (FaaS) platforms, such as AWS Lambda and Azure Functions, require users to specify resource configurations, typically in terms of vCPU and memory allocations. These configurations are bundled by providers [1], where the vCPU quota scales with the requested memory. Underprovisioning memory leads to execution failures, as swap memory is generally disabled in cloud environments. Conversely, memory overprovisioning is common, not because the workload demands it, but as a strategy to gain higher vCPU quotas and enhance function performance. This misalignment results in extensive resource provisioning, fragmentation, and therefore increased costs [7]. The Huawei invocation Trace [17] records the memory provisioned for each function as well as the memory utilized for each request associated with that function. Using this trace, we analyze memory utilization in the most heavily utilized region, as shown in Figure 2a. Memory utilization, defined as the fraction of allocated memory that is actively used, averages just 26%. The median utilization is 16%, while the 90th percentile reaches 54%. Figure 2b demonstrates the cumulative density function (CDF) of memory usage across function invocations, showing that 90% of the invocations have a memory footprint below 500 MB. While this number seems negligible, scaling this to thousands of concurrent function instances, can lead to a substantial aggregated memory footprint.

3 Elastic Memory for Serverless Functions

In modern data centers, disaggregated memory pools are increasingly adopted [22], leveraging technologies such as Compute Express Link (CXL) to enable flexible and efficient memory management. By utilizing disaggregated memory, the system can dynamically allocate and share remote memory across nodes, allowing for elastic provisioning, reducing fragmentation, enhancing utilization, and lowering costs compared to traditional static memory allocation. While explored for different application domains [28, 30], there is no analysis for leveraging disaggregated memory for the execution phase of serverless functions.

In this section, we examine the implications of disaggregated memory on function performance within the serverless computing context, considering factors such as varying payload sizes, horizontal scaling effects, and overall resource efficiency. Due to the lack of commercially available CXL-enabled memory solutions, we emulate remote memory pools utilizing an Intel Optane Persistent Memory, a non-volatile memory technology. Specifically, we conduct our experiments on a virtual machine (VM) configured with 16 vCPUs, 8 GB of DRAM, and 4 GB of NVRAM, running Linux kernel v6.9. The VM is deployed on top of an Intel® Xeon® Gold 5218R CPU running at 2.10 GHz. Similar to CXL [22], the remote memory (NVRAM) is configured as a zero-CPU NUMA node in Direct Access (DAX) mode. We evaluate four serverless functions from the SeBs benchmark suite [8][1]: a website ranking algorithm (*graph pagerank*), thumbnail generation (*thumbnailer*), DNA sequence visualization (*dna visualization*), and Image Classification using ResNet50 (*image recognition*). Data exchange was facilitated by the MinIO key-value store, hosted on a dedicated physical node, and communicated over the network.

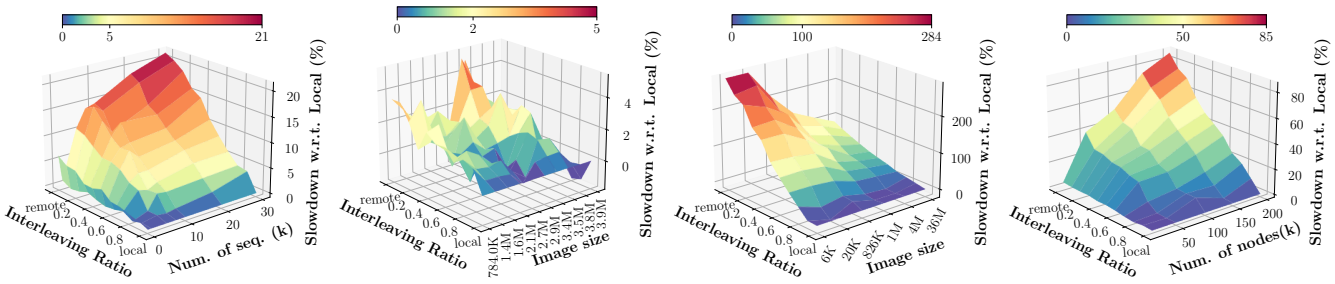
3.1 Impact of remote memory on functions' latency

To assess the potential trade-offs associated with using remote memory for function execution, we analyze the impact on performance degradation—measured in terms of latency—alongside the reduction in local memory footprint. To do so, we utilize the *weighted interleaving* feature of `numactl`, which is available in newer Linux kernel versions. This feature enables the specification of different local-to-remote memory allocation ratios, referred to as the *interleaving ratio*. Specifically, we configure the NUMA allocation policy to span a spectrum of configurations, ranging from exclusive usage of the remote NUMA node (denoted as 0), backed by Intel Optane, to exclusive usage of the local NUMA node (denoted as 1), which utilizes DRAM. To capture intermediate allocation scenarios, we explore interleaved configurations by incrementally adjusting the interleaving ratio in steps of 0.1 across the range.

Figure 3 presents the results of our analysis. The 3D plots illustrate the performance slowdown relative to local execution (z-axis), across varying interleaving ratios (y-axis) and payload sizes (x-axis). In this analysis, we focus exclusively on processing latency by excluding I/O-related latencies, such as input/output download and upload times [20]. The processing latency of *dna visualization* increases by up to 21% for the largest examined payload size. In contrast, *graph pagerank* and *image recognition* exhibit more pronounced sensitivity to remote memory allocations, with latency increases of approximately 85% and 390%, respectively. The *thumbnailer*, however, demonstrates significant performance variability, making it difficult to attribute the observed noise directly to remote memory overhead, as illustrated in the figure. In general, we observe that memory allocations closely follow the interleaving ratio, reflecting a strong correlation between the two. For example, an interleaving ratio of *0.5* reduces the local memory footprint to approximately half that of a local-only allocation.

◇ ***Takeaway:*** *The impact of remote memory latency varies across functions, presenting an opportunity to optimize the total local memory footprint by leveraging functions that are more tolerant to remote memory allocations.*

[1]Without loss of generality, for the rest of the paper, we consider up to four vCPUs available per function.



(a) dna visualization (b) thumbnailer (c) image recognition (d) graph pagerank

Figure 3: Execution latency slowdown (z-axis) of different serverless functions from SeBs [8], for different interleaving ratios (y-axis), and payload sizes (x-axis). Units of measurements for payload sizes: number of DNA sequences for *dna visualization*, image size for *thumbnailer*, and *image recognition*, and number of nodes for *graph pagerank*.

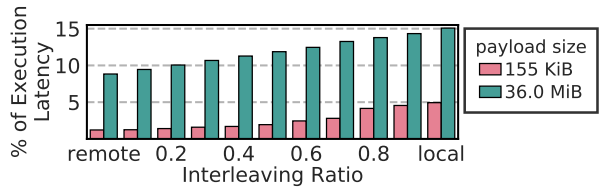


Figure 4: Impact of payload (image) size on end-to-end execution latency for different interleaving ratios.

3.2 Impact of payload size

Serverless functions are inherently stateless, with their input and output typically exchanged via remote storage systems such as AWS S3 [2]. As noted in prior studies [29, 36], payload size can significantly impact function latency, depending on the workload. The unit of measurement for the payload size is benchmark-specific, i.e., image size for *image recognition*, and *thumbnailer*, number of nodes for *graph pagerank*, and number of sequences for *dna visualization*. As shown in Figure 3, payload size influences the processing latency slowdown (% increase in latency when the function utilizes remote memory). For *graph pagerank*, and *dna visualization* the slowdown ranges from 4-85%, and 6-21%, respectively, across the examined payload sizes. In contrast, as observed previously, no clear correlation exists between payload size and slowdown for the *thumbnailer*. Finally, *image recognition* exhibits a unique behavior: as payload size increases, the extent of slowdown decreases. We attribute this behavior to the three following observations:

(1) **Execution phase stages:** In vision model inference, the function first preprocesses the downloaded image, to satisfy the inference model's specific requirements. This may include changing the color format (e.g., RGB) or resizing to meet the model's required image size dimensions (e.g., 224x224 for *Resnet50*).
(2) **Stages contribution to execution latency:** Given the previous point, as the payload (image) size increases, the portion of time spent in the preprocessing stage increases, because larger images require more computational resources for decoding and transformation. On the other hand, the respective time for inference remains the same, since this step always receives the same image dimension.
(3) **Sensitivity to remote memory:** Different stages are affected by remote memory usage in different volumes. E.g., for the 36MiB image, we measured the impact on preprocessing and inference at 100% and 385% respectively.

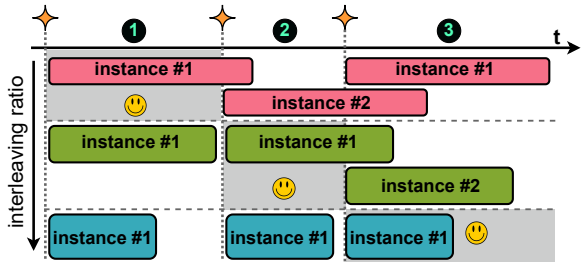
To showcase the impact of other phases, e.g., input download, we further analyze the end-to-end latency of the *image recognition* function. Figure 4 illustrates the proportion of image download time relative to the total latency for two image sizes and various interleaving ratios. As expected, downloading larger images contributes more significantly to the overall latency. It accounts for up to 15%/5% of the execution time for a 36 MiB and 155 KiB image respectively. The absolute download latency is not influenced by the interleaving ratio. However, its relative contribution to the total latency increases as larger portions of local memory are utilized, since other stages, such as inference, have shorter execution durations in these cases.

◇ ***Takeaway:*** *Payload size must be carefully considered when selecting configurations, as it can have a substantial and sometimes counterintuitive impact on execution time slowdown.*

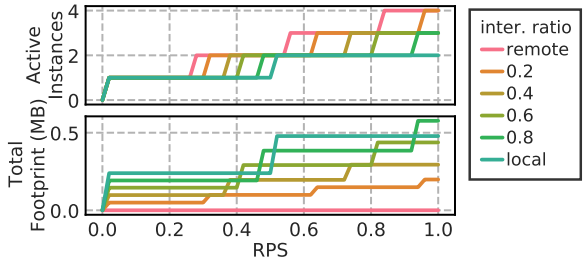
3.3 Pitfall of Remote Memory Offloading

While offloading memory to a remote pool can reduce the local memory footprint for individual function executions, (albeit at the cost of increased latency), this strategy can backfire under high request arrival rates. The increased execution time caused by remote memory access latency can cause increased overlapping invocations in certain scenarios, e.g., high function invocation ratios. Therefore, to preserve throughput, platforms must spawn additional sandboxes to handle new requests. Paradoxically, this can also increase the overall local memory footprint, as more sandboxes are instantiated, each imposing its baseline memory requirements.

We demonstrate this behavior with a simple indicative example. Figure 5a shows a scenario of three consecutive function invocations over time (stars), served by three configurations with different interleaving ratios. Each such configuration exhibits variations in execution latency (width), memory footprint (height), and, consequently, scaling behavior. We observe that across the 3 different intervals, a different configuration ratio results in the best overall (aggregated) memory footprint. In the first interval,



(a) Function instantiation example. Gray areas denote the minimum footprint for each time interval. The height of the boxes denotes the functions' local memory footprint, while their width denotes their duration.



(b) Active instances, and total memory footprint for image recognition function for different request per second values.

Figure 5: An example of overlapping invocations. Arrival times affect the total memory footprint for different interleaving ratios, requiring additional active function instances.

the configuration with the lowest interleaving ratio achieves the smallest total local memory consumption. However, in the second and third intervals, where additional instances are required to sustain throughput (§ 2.1), different configurations emerge as the most memory-efficient.

In Figure 5b we simulate the scale-out behavior for the *image recognition* function. We plot the active instances required (top) and the total memory footprint (bottom) for various interleaving ratios and request rates (request per second, rps). For the local-only (ratio = 1), scale-out begins at approximately 0.5 rps. In contrast, configurations with lower ratios scale out earlier as the request rate increases. This effect becomes more pronounced at higher rps values. However, since the memory footprint per instance varies across interleaving ratio configurations, the total memory footprint does not directly correlate with the number of active instances. In some scenarios, the total memory footprint for certain interleaving ratios can temporarily surpass that of the local-only configuration, such as for rps values between 0.4 and 0.5. Additionally, the increase in active instances due to overlapping invocations also leads to up to twice as many cold starts when comparing local with remote.

◇ ***Takeaway:*** *Offloading memory can reduce local footprint but may backfire under high request rates, increasing invocation overlap and instance spawns. This can paradoxically raise overall memory usage, making the optimal interleaving ratio workload-dependent.*

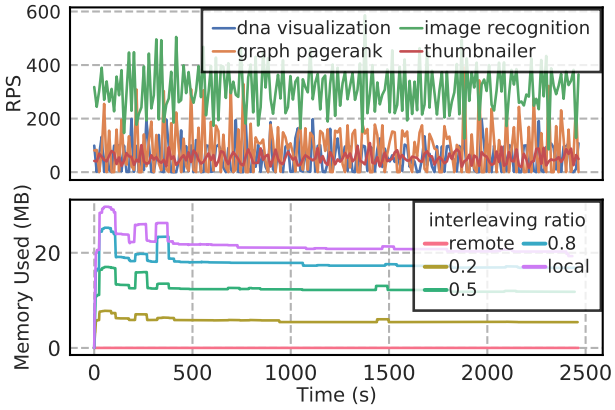


Figure 6: RPS for the four functions over time (top) and memory footprint over time for different interleaving ratios (bottom), both based on the Huawei Trace [17].

4 Trace-driven Analysis

4.1 Simulation Setup

To evaluate memory interleaving policies in serverless, we develop an in-house Function-as-a-Service simulator. It models the complete function lifecycle, including instantiation, execution, auto-scaling, and scale-to-zero mechanisms. The keep-alive time for functions is set to 60 seconds, while a new instance is provisioned for each function invocation if no idle instance is available, similar to [5]. We analyze Huawei's trace [17] and extract the first one million invocation requests for the four most frequently invoked functions. We then mapped these functions to the benchmarks outlined in Section 3 to ensure compatibility with our evaluation framework. To align the trace's payload distributions with that of our benchmarks, we rescale the payload sizes and assign them to the nearest predefined bins. We assume ideal memory provisioning, i.e., equal to the memory footprint measured during profiling. For the analysis, we process the first one million requests from the trace for the four selected functions, logging the memory usage at 100 ms intervals.

4.2 Results

Memory Footprint Analysis Over Time: To assess the impact of disaggregated memory on resource efficiency, using the serverless trace, our first experiment evaluates the aggregated local memory footprint over time across different interleaving ratios. The upper plot in Figure 6 illustrates the requests per second (RPS) for the four selected functions, while the lower plot depicts the local memory footprint over time under different interleaving ratio configurations. At the start of the trace, we observe a spike in local memory usage as new instances are instantiated to handle incoming requests. As execution progresses, the footprint stabilizes, benefiting from a sufficient number of keep-alive instances that sustain the invocation rate. Fluctuations in local memory footprint arise from the dynamic spawning and termination of function instances as part of the scale-up/down mechanism. Notably, while the expected memory footprint reduction of the interleaving ratio equal to 0.8 would

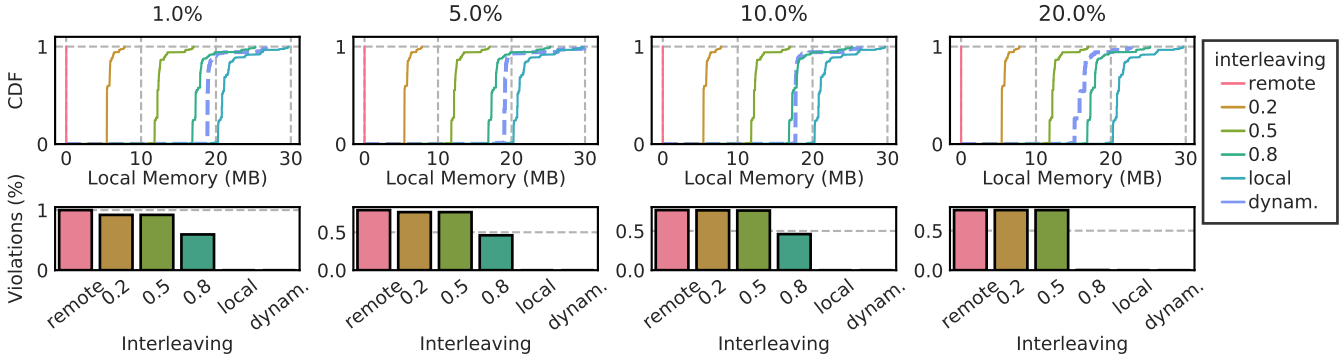


Figure 7: (Top): CDF of the memory footprint (MB), and (bottom): violations % for different interleaving configurations (*"dynam."* denotes the dynamic ratio assignment) and different SLO strictness levels (1, 5, 10, and 20% higher than local-only).

be 20% (w.r.t. local-only), the average and maximum percentage differences are 17% and 15% respectively. This supports our previous finding (§3.3) that local memory reduction is limited by the increased overlap of invocations, leading to more active instances.

Managed interleaving ratio selection: Next, we compare the impact of fixed vs. dynamic interleaving settings on local memory footprint under latency constraints, while maintaining throughput. In the "fixed" case, interleaving settings persist across functions and payload sizes. Dynamic interleaving (denoted as "*dynam.*") selects for each function and payload size combination, the minimum ratio that satisfies the latency constraint. We define latency constraints as thresholds that encompass a range of possible overheads, allowing execution latencies of up to 1%, 5%, 10%, and 20% above the local-only baseline for each benchmark and payload combination. Notably, small differences, such as 1%, could fall within the margin of error and be attributed to noise rather than meaningful performance variations. Figure 7 presents the cumulative probability distributions (CDF) for local memory usage (top) and the number of latency violations (bottom) under different constraints (columns). The dynamic strategy reduces the local memory footprint by 6% to 25% (median of the distribution) compared to the local-only configuration, depending on the latency degradation tolerance (from 1% to 20%). This occurs because relaxing latency constraints creates more opportunities for remote memory offloading. Additionally, the 0.8 interleaving ratio, despite having a local memory footprint distribution similar to the dynamic strategy, results in significantly higher violation rates – 60% and 46% – for the 1% and 10% latency tolerance levels. This is due to the workload-specific memory interleaving applied by "*dynam.*", which adapts to function sensitivity, ensuring efficient offloading for tolerant functions while mitigating violations for more sensitive ones.

5 Discussion & Future Directions

Limitations of weighted memory interleaving: While our approach leverages memory disaggregation to improve serverless function elasticity, it currently allocates pages randomly across NUMA nodes. This provides load-balancing benefits by distributing memory accesses, which can help mitigate bandwidth limitations. However, this strategy does not account for data locality, potentially leading to frequent accesses to remote (slower) memory, impacting latency-sensitive workloads. A more sophisticated, data-access-driven memory allocation strategy could dynamically assign pages based on memory access patterns [10, 13, 15, 19, 24]. For example, hot pages could be allocated to local (fast) memory, while cold pages could be migrated to remote memory pools. Implementing such an adaptive allocation strategy could significantly reduce the performance overhead of remote memory while preserving its cost benefits. Nevertheless, the overhead introduced by monitoring and classifying pages as hot or cold must be carefully considered, especially in the context of short-lived serverless functions, where such overheads can be significant relative to function execution latency.
Further optimization strategies: Additionally, future work can explore predictive strategies to further optimize remote memory performance. For example, memory prefetching [6, 12, 27, 33] can proactively fetch data before it is accessed, reducing remote memory latency. Integrating such predictive mechanisms with serverless memory disaggregation could improve performance by minimizing cache misses and reducing memory stall cycles.

6 Conclusion

This paper examines the challenges and opportunities of leveraging memory disaggregation in serverless computing, focusing on latency and memory footprint optimization. Using weighted interleaving, we demonstrate up to 25% local memory reductions while addressing latency constraints. Our findings highlight the importance of workload-specific remote memory configurations, with dynamic interleaving strategies balancing resource efficiency and performance.

Acknowledgments

This research was partially funded by the European Union's Horizon 2020 Research and Innovation programme CONVOLVE under Grant Agreement No 101070374 and the Hellenic Foundation for Research and Innovation (HFRI) under the 3rd Call for HFRI Ph.D. Fellowships (Fellowship Number: 5349).

The authors wish to acknowledge the use of ChatGPT in the writing of this paper. This tool was used only to assist with polishing and improving the language of the final version of the text. The paper remains an accurate representation of the authors' underlying work and novel intellectual contributions.

References

[1] [n. d.]. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing. Accessed: January 10th, 2025.

[2] [n. d.]. AWS S3. https://aws.amazon.com/s3/. Accessed: January 9th, 2025.

[3] [n. d.]. Google Cloud Funtions. https://cloud.google.com/functions. Accessed: February 4th, 2025.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[5] AWS. [n. d.]. Lambda Executions. https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html. Accessed: January 15th, 2025.

[6] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1121–1137.

[7] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 381–397.

[8] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*. 64–78.

[9] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *Comput. Surveys* 56, 11 (2024), 1–37.

[10] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. 2021. Cori: Dancing to the right beat of periodic data movements over hybrid memory systems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 350–359.

[11] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[12] Gerasimos Gerogiannis and Josep Torrellas. 2023. Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 698–713. https://doi.org/10.1145/3613424.3623780

[13] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. In *International Conference on Machine Learning*. PMLR, 1919–1928.

[14] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, et al. 2024. TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 421–437.

[15] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. 2024. {ExtMem}: Enabling {Application-Aware} Virtual Memory Management for {Data-Intensive} Applications. In *2024 USENIX annual technical conference (USENIX ATC 24)*. 397–408.

[16] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[17] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. 2024. Serverless Cold Starts and Where to Find Them. *arXiv preprint arXiv:2410.06145* (2024).

[18] Christos Katsakioris, Chloe Alverti, Vasileios Karakostas, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2022. Faas in the age of (sub-) μs i/o: a performance analysis of snapshotting. In *Proceedings of the 15th ACM International Conference on Systems and Storage*. 13–25.

[19] Manolis Katsaragakis, Konstantinos Stavrakakis, Dimosthenis Masouros, Lazaros Papadopoulos, and Dimitrios Soudris. 2023. Adjacent lstm-based page scheduling for hybrid dram/nvm memory systems. In *14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 7–1.

[20] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 81–92.

[21] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[22] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[23] Dimosthenis Masouros, Christian Pinto, Michele Gazzetti, Sotirios Xydis, and Dimitrios Soudris. 2023. Adrias: Interference-aware memory orchestration for disaggregated cloud infrastructures. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 855–869.

[24] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. Daos: Data access-aware operating system. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 4–15.

[25] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. 2020. Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 868–880.

[26] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 714–729.

[27] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. 2022. Sibyl: Adaptive and extensible data placement in hybrid storage systems using online reinforcement learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 320–336.

[28] Achilleas Tzenetopoulos, Michele Gazzetti, Dimosthenis Masouros, Christian Pinto, Sotirios Xydis, and Dimitrios Soudris. 2024. Disaggregated RDDs: Extending and Analyzing Apache Spark for Memory Disaggregated Infrastructures. In *2024 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 107–117.

[29] Achilleas Tzenetopoulos, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. 2024. Leveraging Core and Uncore Frequency Scaling for Power-Efficient Serverless Workflows. *arXiv preprint arXiv:2407.18386* (2024).

[30] Jacob Wahlgren, Gabin Schieffer, Maya Gokhale, and Ivy Peng. 2023. A quantitative approach for adopting disaggregated memory in HPC systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[31] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. 2022. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 609–621.

[32] Chuhao Xu, Yiyu Liu, Zijun Li, Quan Chen, Han Zhao, Deze Zeng, Qian Peng, Xueqi Wu, Haifeng Zhao, Senbo Fu, et al. 2024. FaaSMem: Improving Memory Efficiency of Serverless Computing with Memory Pool Architecture. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 331–348.

[33] Huijing Yang, Juan Fang, Xing Su, Zhi Cai, and Yuening Wang. 2024. RL-CoPref: a reinforcement learning-based coordinated prefetching controller for multiple prefetchers. *J. Supercomput.* 80, 9 (Feb. 2024), 13001–13026. https://doi.org/10.1007/s11227-024-05938-9

[34] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 768–781.

[35] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 724–739.

[36] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1–14.