Memento: Architectural Support for Ephemeral Memory Management in Serverless Environments

Ziqi Wang
Carnegie Mellon University
Pittsburgh, USA
ziqiw@cs.cmu.edu

Kaiyang Zhao
Carnegie Mellon University
Pittsburgh, USA
kaiyang2@cs.cmu.edu

Pei Li
Carnegie Mellon University
Pittsburgh, USA
peili@andrew.cmu.edu

Andrew Jacob
Carnegie Mellon University
Pittsburgh, USA
ajacob@andrew.cmu.edu

Michael Kozuch
Intel Labs
Pittsburgh, USA
michael.a.kozuch@intel.com

Todd C. Mowry
Carnegie Mellon University
Pittsburgh, USA
tcm@cs.cmu.edu

Dimitrios Skarlatos
Carnegie Mellon University
Pittsburgh, USA
dskarlat@cs.cmu.edu

ABSTRACT

Serverless computing is an increasingly attractive paradigm in the cloud due to its ease of use and fine-grained pay-for-what-you-use billing. However, serverless computing poses new challenges to system design due to its short-lived function execution model. Our detailed analysis reveals that memory management is responsible for a major amount of function execution cycles. This is because functions pay the full critical-path costs of memory management in both userspace and the operating system without the opportunity to amortize these costs over their short lifetimes.

To address this problem, we propose Memento, a new hardware-centric memory management design based upon our insights that memory allocations in serverless functions are typically small, and either quickly freed after allocation or freed when the function exits. Memento alleviates the overheads of serverless memory management by introducing two key mechanisms: (i) a hardware object allocator that performs in-cache memory allocation and free operations based on arenas, and (ii) a hardware page allocator that manages a small pool of physical pages used to replenish arenas of the object allocator. Together these mechanisms alleviate memory management overheads and bypass costly userspace and kernel operations. Memento naturally integrates with existing software stacks through a set of ISA extensions that enable seamless integration with multiple languages runtimes. Finally, Memento leverages the newly exposed memory allocation semantics in hardware to introduce a main memory bypass mechanism and avoid unnecessary DRAM accesses for newly allocated objects.

We evaluate Memento with full-system simulations across a diverse set of containerized serverless workloads and language runtimes. The results show that Memento achieves function execution speedups ranging between 8–28% and 16% on average. Furthermore, Memento hardware allocators and main memory bypass mechanisms drastically reduce main memory traffic by 30% on average. The combined effects of Memento reduce the pricing cost of function execution by 29%. Finally, we demonstrate the applicability of Memento beyond functions, to major serverless platform operations and long-running data processing applications.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Memory management**.

KEYWORDS

Cloud computing, Serverless, Function-as-a-Service, Memory Management

ACM Reference Format:
Ziqi Wang, Kaiyang Zhao, Pei Li, Andrew Jacob, Michael Kozuch, Todd C. Mowry, and Dimitrios Skarlatos. 2023. Memento: Architectural Support for Ephemeral Memory Management in Serverless Environments. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3623795

1 INTRODUCTION

Memory management is responsible for a major chunk of datacenter cycles as revealed by Google's internal profiling [20, 25], despite significant efforts in optimizing them in software [16, 32]. Unfortunately, while long-running workloads have some opportunity to amortize part of their memory management costs over their long runtimes, this is not the case for short-lived functions. Prior work on serverless computing [17, 22–24, 29, 43, 45–47, 50, 51, 53, 54, 58, 59], has mostly focused on reducing the cold-start effects of functions [2, 3, 6, 14, 18, 39, 48, 52, 55, 56, 60]. However, memory

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3623795

management still relies on expensive allocation and deallocation paths on the critical path of function execution, leading to major overheads in serverless environments.

In particular, functions pay the full cost of memory allocation and deallocation in both userspace and the OS on the critical path of their short runtime. For example, consider the overheads of typical memory management operations. For starters, applications must pay the cost of memory management in *userspace*, where each allocation and free typically requires tens of instructions in popular high-level languages for serverless environments (e.g., Python). Furthermore, userspace allocator rely on the OS to actually allocate physical memory. However, the allocation path in the OS, either performed at system call time, such as `mmap` or later on at access time through page faults, is inherently expensive. This is because of the creation of page tables and the actual physical memory allocation and bookkeeping, requiring additional thousands of instructions.

In this paper, we focus on reducing the performance overhead of memory management for serverless functions. To better understand the characteristics of memory management in serverless functions and to expose opportunities for optimization, we first start with a detailed investigation of function behavior across several functions and three language runtimes that cover a wide spectrum of memory management behavior. Beyond serverless functions, we further investigate serverless platform operations such as function deployment, and long-running data processing applications. Our analysis reveals three key insights. First, the vast majority of allocations are relatively small, no larger than 512 bytes. This characteristic is partly due to the usage of high-level languages, where small objects are used extensively. Second, the allocation lifetime is bimodal based on the language runtime. In particular, objects are either allocated and freed shortly after, usually within less than 16 other allocations of the same size class. Alternatively, allocations are often not freed due to the short runtime of functions and they are instead batch-freed by the OS when the function exits. Finally, memory management spends a significant amount of cycles in userspace and within the OS. Especially in high-level language runtimes, almost half of memory allocation cycles are spent in the OS.

Based on these insights we propose *Memento*, a holistic hardware-centric design that optimizes memory management by moving most of the work on the software critical path to hardware. Memento introduces two key mechanisms that operate in tandem. The first is a *hardware object allocator* that tracks objects using *arenas* and performs memory allocations and frees in novel per-core metadata cache called the Hardware Object Table (HOT). Due to the small sizes of allocations, a small number of size classes can be efficiently cached. Furthermore, for allocations that are allocated and quickly freed allocation metadata exhibit temporal locality. Therefore, the hardware object allocator can satisfy requests entirely in the metadata cache within only a few cycles.

The second mechanism in Memento is a *hardware page allocator* that manages a small pool of free virtual and physical pages. The hardware page allocator serves the dual purpose of (i) replenishing the hardware object allocator with free virtual pages that constitute arenas; and (ii) backing virtual pages with physical memory on-demand when a virtual page is accessed for the first time. The hardware page allocator removes the kernel path of memory management from the critical path of function execution, thereby avoiding invoking expensive system calls and page fault handlers. Furthermore, for allocations that are batch-freed at the end of function execution, the hardware page allocator can deallocate their memory with low latency.

Overall, these two mechanisms work together in a complementary fashion that resembles the userspace and kernel path in the current software stack. Memento naturally integrates with the existing software stack by adding two new instructions to the ISA for allocation and deallocation of memory. The hardware design and interface of Memento enable seamless integration with both compiled and interpreted languages, including those with and without garbage collection.

Finally, Memento leverages the newly exposed memory allocation semantics in hardware to introduce a main memory bypass mechanism. Memento detects newly allocated objects and avoids unnecessary DRAM accesses. Instead, Memento instantiates lines entirely in the cache hierarchy.

We evaluate Memento with full-system simulations running a diverse set of serverless workloads and three language runtimes in a containerized environment. The results show that Memento achieves function execution speedups ranging between 8–28% and 16% on average. Furthermore, Memento's hardware allocators and memory bypass mechanisms drastically reduces main memory traffic by 30%. The combined effects of Memento reduce the pricing cost of function execution by 29%. Finally, we demonstrate the applicability of Memento beyond functions, to major serverless platform operations responsible for deploying function instances, and long-running data processing applications.

This paper's contributions are:

- A detailed study of the memory management behavior of serverless functions that identifies the costs and optimization opportunities of memory management operations in both userspace and the OS kernel.
- A novel arena-based hardware object allocator and a Hardware Object Table(HOT) that enables hardware to fully manage userspace memory allocations.
- A hardware page allocator that manages virtual and physical address assignment to allocation arenas, replenishes the hardware object allocator, and eliminates the OS costs of memory management.
- A main memory bypass mechanism that leverages the introduction of memory allocation semantics in hardware to avoid unnecessary DRAM accesses.
- The evaluation of Memento across a large set of serverless functions that shows Memento significantly improves their performance, bandwidth usage, and runtime cost. It further shows that Memento is applicable beyond functions to serverless platform operations and data processing applications.

2 BACKGROUND AND OPPORTUNITIES ON MEMORY MANAGEMENT

This section first provides a background on memory management in userspace and the OS. Then we showcase the memory behavior of serverless workloads.

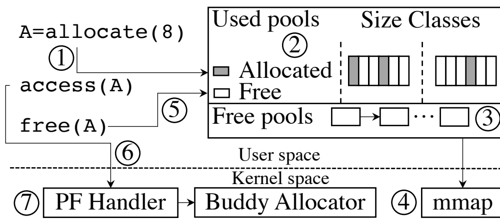

Figure 1: Memory management overview.

2.1 A Day in the Life of a Memory Allocation

Memory management is split between userspace and OS operations. Typically, variable byte-sized allocations are performed in userspace while the OS manages memory resources at the granularity of pages. Referencing Fig. 1, we describe the steps and effects of memory allocation and free, based on a typical example of userspace memory allocation, and then shed light on the steps performed by the OS.

Userspace Operations. We use the CPython implementation of the heap allocator, `pymalloc` [1], as an example to describe the allocator procedure. Allocators in other high-level language runtimes also follow similar steps. The allocator requests memory from the OS at the granularity of 256KB *arenas* and then splits them into smaller 4KB *pools*. Free objects within a pool are organized as linked lists, and each pool serves allocation requests of a particular size class. On an allocation request, the size class is computed by aligning the requested size up to the nearest 8-byte boundary (Step ① in Fig. 1). Then, the allocator checks the per-size class free pool list, and if a pool with free objects is present, the head entry of the free list is returned to the caller (Step ②). However, if no free objects can be found, the allocator attempts to grab a new free pool from the free pool list (Step ③) and tries again. If there are no free pools, `mmap` in the kernel is called to allocate more arenas (Step ④). Allocation requests that are larger than 512 bytes by default are directly serviced by `malloc` in glibc, which eventually calls `mmap` as well.

On a free operation (Step ⑤), the pool header is first obtained by aligning the address down to the nearest 4KB boundary. Then the object is returned to the pool by linking it to the head of the free list. If the free operation turns the pool entirely free, then the pool is returned to the free pool list. Finally, if all pools in an arena become free, the allocator returns its memory by calling `munmap`.

Kernel Space Operations. Userspace allocators request memory allocations from the kernel through system calls. Continuing the above example, when the `mmap` [36] system call is invoked (Step ④), it performs the following. First, `mmap` finds an unused region of addresses in the virtual address space of the process and then sets up mapping metadata describing the allocation. Note that no physical storage backs the virtual address region allocated in this stage. Instead, the system call only returns the region's start address to the userspace without setting up any virtual-to-physical mapping.

As a result, when the software accesses a newly allocated virtual memory page for the first time, a page fault will be raised (Step ⑥) by hardware due to lacking a valid address mapping. The page fault is serviced by a kernel handler, which finds the mapping metadata set up earlier by `mmap` on the faulting address. In this stage, the handler performs physical memory allocation by requesting a free physical page from the kernel's physical page allocator and then setting the corresponding page table entry to point to the physical page (Step ⑦). Eventually, the faulting memory access is retried and will land on the newly allocated physical page.

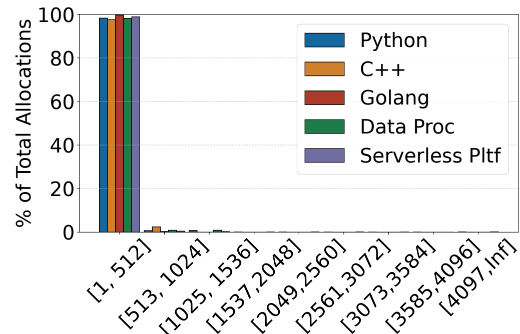

Figure 2: Allocation size (Bytes)

Applications deallocate memory through the `munmap` [36] system call that performs the opposite steps. First, `munmap` tears down the mapping metadata. Then it walks the page table entries describing the mapping, clears page table entries, and returns physical pages to the kernel as needed. Finally, if relevant page tables become empty, they are also freed.

2.2 Memory Management Behavior

Short-lived functions pose new challenges to memory management. In this section, we study the memory behavior of function execution across serverless workloads based on four suites [11, 19, 28, 40] and three languages, Python, C++, and Golang. To characterize the allocation sizes and lifetimes of functions we instrumented the allocators of each language and collected allocation traces. We normalize the number of allocations of each function, then we aggregate across functions, and show the per language breakdown. We further characterize separately four long-running data processing applications (*Data Proc*) written in C++ and three key serverless platform operations (*FaaS Pltf*) written in Golang. We further show a breakdown between userspace and kernel memory management. We discuss our workloads in detail in Section 5.

Allocation Sizes and Object Lifetimes in Userspace . Fig. 2 shows the object size distribution in 512-byte increments. The results show that allocations are small. Specifically, 93% of allocations are smaller than 512 bytes. For several workloads, small allocations can account for more than 98% of all allocations. Large allocations are a rare occurrence. Size distributions within 512 bytes are heavily workload-dependent and we did not observe any consistent patterns for small allocations across the workloads. In data processing applications, small allocations account for 98%, while for the serverless platform, 99% of allocations are smaller than 512 bytes. Overall, small allocations dominate across Python, C++, and Golang.

To characterize object lifetimes, we define a lifetime metric based on the malloc-free distance. In particular, we compute the number of allocations of the same size class before the object is freed. This is a good predictor of allocation metadata locality. Fig. 3 shows the average lifetime distribution of object allocations. The results

show that function allocation lifetimes exhibit a bimodal behavior. Specifically, 71% of allocations are short-lived as they are freed within only 16 allocations of the same size class. On the other hand, 27% of allocations are long-lived and rely on OS deallocation when the function exits. Furthermore, object lifetime is highly dependent on the language runtime. Specifically, for C++ the majority of allocations are short-lived, while for Python they are primarily short-lived except for a few long-lived ones. Furthermore, Golang allocations are long-lived because garbage-collection is not invoked due to the short runtime of functions and allocations are batch-freed at the end of function execution. Finally, in the serverless platform case, most allocations are long-lived due to the Golang garbage collection. Data processing applications written in C++ exhibit primarily short-lived allocations.

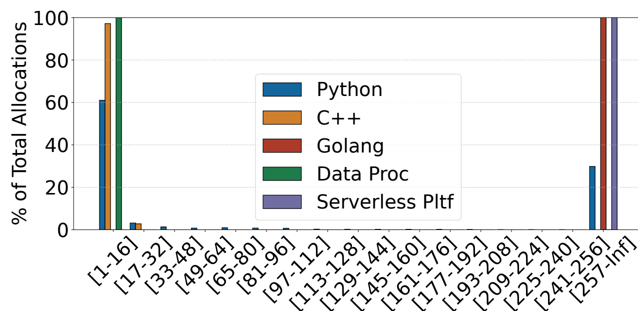

Figure 3: Allocation lifetime (Malloc-Free distance).

In Table 1 we show the joint distribution of allocation size and lifetime for functions. On average, 61% of allocations are *both* small and short-lived. These objects are small in size and freed quickly after allocation. For several workloads, short-lived and small allocations account more than 80% and up to 96% of all allocations. Furthermore, 32% of allocations are long-lived and small. Finally, larger allocations account for 7% of total allocations with the majority being short-lived. Separately from the table, in data processing applications, 97% of the allocations are both small and short-lived. In the case of the serverless platform, 99% of the allocations are small while being long-lived.

	Small	Large
Short-lived	61%	6.55%
Long-lived	32%	0.45%

Table 1: Combined distribution of size and lifetime.

Kernel Impact. We further characterize the impact of kernel memory management in serverless workloads for different languages. Table 2 shows the breakdown. In C++ workloads, the majority of the memory management overhead originates from userspace, accounting for 96% of the total memory management costs. We find that such behavior is due to the smaller heap working set size of C++ applications. On the other hand, for Python and Golang runtimes, the kernel memory management overhead is substantial, accounting for 52% and 46% respectively, due to their relatively larger working sets. For the serverless platform, userspace accounts for 59% while the kernel accounts for 41%. In data processing applications, the split is 38% and 62%. Overall, both userspace and kernel memory management play a critical role in serverless functions.

	Python	C++	Golang	FaaS Platform	Data Proc.
User/Kernel	48%/52%	96%/4%	56%/44%	59%/41%	38%/62%

Table 2: Memory Management Cycles Breakdown.

Insights and Implications. Our detailed study illustrates three prominent memory management behaviors of serverless workloads. First, most objects exhibit varying patterns in small objects under 512 bytes. Therefore, the allocator should optimize for small and varying size classes. Second, most objects are freed shortly after allocation, indicating that they are short-lived. The allocator should prioritize short-lived object handling and take advantage of the strong allocation metadata locality. Third, our detailed study of serverless functions highlights that both userspace and kernel allocations play a substantial role. Especially for Python and Golang runtimes, addressing only the userspace component would leave almost half of the memory management overhead intact. Beyond functions, we observe similar behavior in the serverless platform operations and the data processing applications.

Guiding the Design of Memento. Based on these insights, we design Memento, a holistic hardware-centric approach to eliminate the overheads of memory management. To handle varying patterns of object sizes under 512 bytes, Memento maintains multiple size classes. Allocations in each size class are satisfied by the dedicated metadata for that particular size class. Furthermore, Memento leverages the strong temporal locality of allocation metadata by adding a hardware object allocator that caches allocation metadata near the processor. Hence, Memento fulfills most allocation requests in only a few cycles. Then, Memento introduces a hardware page allocator to fulfill the needs of the object allocator and eliminate the cost of kernel memory management for serverless workloads. Finally, Memento leverages the newly exposed memory allocation semantics in hardware to introduce a main memory bypass mechanism and avoid unnecessary DRAM accesses for newly allocated objects.

3 MEMENTO DESIGN

The design of Memento consists of two mechanisms that operate in tandem to handle userspace and kernel memory management operations. The first mechanism is an arena-based *hardware object allocator* that is located close to the core and it is responsible for object allocation and deallocation. The hardware object allocator interfaces with the software stack through a set of ISA extensions that provide `malloc` and `free` semantics and enable a seamless integration with language runtimes. Next, Memento introduces a *hardware page allocator* at the memory controller that manages physical pages. The hardware page allocator replenishes the object allocator with physical memory-backed arenas on-demand. Finally, Memento builds on top of the newly exposed memory allocation

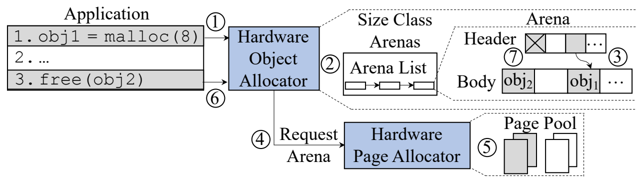

Figure 4: Memento's memory management workflow.

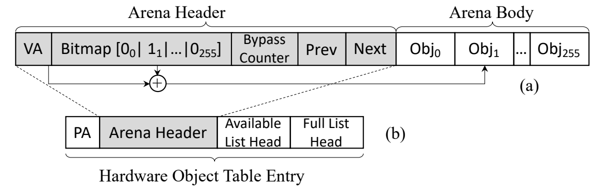

Figure 5: Memento's layout of the (a) arena header and body, and (b) hardware object table entries.

semantics in hardware to design an effective main memory bypass mechanism that enables newly allocated objects to completely avoid expensive DRAM accesses. Through this set of mechanisms, Memento drastically reduces the cost of memory management in userspace and kernel by servicing most memory allocation and deallocation requests with a latency equivalent to a single roundtrip to the L1 cache, improving performance and reducing main memory memory traffic.

Memento's Workflow In Fig. 4 we show a high-level overview of Memento's workflow and how the hardware object and page allocator cooperate to satisfy memory management operations. First, upon an allocation (step ①), the hardware object allocator will identify the size class of the request. In step ②, the corresponding arena from the arena list will be selected. Next, in step ③, the hardware object allocator (which caches arena headers) will identify and mark a free location within the arena's header where allocation metadata is stored. The arena metadata can be used to identify the location of the body of the arena storing the allocated objects. Finally, the allocated virtual address is returned to the caller.

In the scenario that an arena containing free objects is unavailable, the hardware object allocator requests an arena from the hardware page allocator, as shown in step ④. The hardware page allocator then allocates the appropriate number of pages from the page pool (shown in step ⑤) and returns the new arena to the hardware object allocator. Upon a free operation (step ⑥), the hardware object allocator performs the reverse operations that undo object allocation. Specifically, the allocator will clear the appropriate metadata entry of the arena header, as shown in step ⑦. Similarly, the hardware object allocator notifies the hardware page allocator to free arenas and returns pages to the free page pool.

3.1 Hardware Object Allocator

Conceptually, the hardware object allocator performs userspace-level memory management operations. The goal of Memento is to provide a general interface that can seamlessly support language runtimes instead of hardwiring the design to any particular software allocator. To this end, in Memento we opt to provide the abstraction of an object allocator through an interface similar to `malloc` and `free` drastically simplifying integration with software. To realize this interface, we extend the ISA with two new instructions for allocating and freeing objects: `obj-alloc` and `obj-free`.

The `obj-alloc` instruction carries the requested allocation size as an operand. When executed, it returns the virtual address pointing to a memory block that is available for use and satisfies the requested size. The `obj-free` instruction has the virtual address to be freed as its operand, and it deallocates the block so that future allocations can reuse the block.

We next discuss the two main components of the hardware object allocator: (i) *Memento arenas* that track object-level allocations, and (ii) the *hardware object table* (HOT) that facilitates arena management in hardware.

Memento Arenas. The object allocator tracks the allocation status of memory addresses by maintaining bookkeeping information in the unit of *arenas*. An Memento arena is a consecutive range of virtual addresses and it serves allocation and free requests of only one specific size class during its lifetime. Fig. 5(a) shows the arena layout, which includes the header and the body.

An arena header includes four parts: (i) a virtual address (VA) field, (ii) an allocation bitmap, (iii) a bypass counter, and (iv) two pointers to doubly-linked lists of same-size-class arenas. The VA field stores the base virtual address of the arena. The bitmap is used for tracking the allocation status of objects. A set bit in the bitmap indicates that the object at the corresponding offset has been allocated. The *arena body* is an array of objects of the same size. The combination of the VA field with the offset of a bitmap entry points to the allocated object in the arena body. Each arena contains a fixed number of objects. In our experiments, we set this parameter to 256 objects per arena, balancing metadata cost and internal fragmentation.

Arenas are organized into *arena lists* for each size class. Specifically, two lists are maintained per size class. The first is an *available* list tracking arenas with at least one free object, and the second is a *full* list tracking arenas without any available objects. In addition, arenas are connected to the the appropriate list through the *prev* and *next* pointers. The *bypass counter* is used to support main memory bypass of new allocations. We further discuss the main memory bypass design of Memento in Section 3.3.

Hardware Object Table. The core of the hardware object allocator is the *Hardware Object Table (HOT)*. The HOT holds the most recently used arena header for each size class. Based on our analysis in Section 2.2 we opt to support allocations up to 512 bytes in 8-byte increments, resulting in a total of 64 size classes. Fig. 5(b) shows the HOT entry layout. Each entry stores a cached copy of the arena's header fields, which are loaded from memory. Additionally, to support operations on the available and full list, HOT entries also contain the following: (i) the *PA* field storing the physical address of the arena, and (ii) two pointers, the *available list head* and *full list head* pointers, which store the physical addresses of the first arenas in the available and full list of the size class, respectively.

The hardware object allocator manipulates the hardware object table for initialization, allocation, and free operations following the steps in Fig. 6.

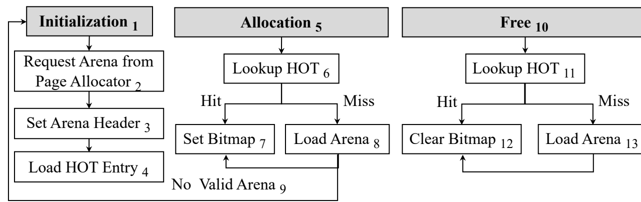

Figure 6: Hardware object allocator steps for initialization, allocation, and free operations.

Initialization. On initialization (①), an arena is allocated for each size class by requesting free pages from the page allocator (②). We further describe the design of the hardware page allocator in Section 3.2. Individual arenas are initialized by preparing the arena header (③) via clearing the bitmap and the linked list prev and next pointers. The arena header's VA field is set to the virtual address assigned to the arena. The PA of the HOT entry is set to the header's physical address returned from the page allocator. The available and full list head pointers are set to indicate that both lists are empty. Finally, the initialized arena header is loaded into the corresponding HOT entry of the allocator (④).

Allocation. On an allocation request (⑤), the size class is computed by rounding the requested size up to the nearest 8-byte boundary. Then the HOT entry is located swiftly using the size class as an index without any associative search (⑥). Next, the bitmap is scanned. Three cases might occur in this step which we discuss next.

In the most common case, a cleared bit is found, and the allocation completes quickly by setting it (⑦). The allocator computes the address of the allocated object based on the entry's VA field and the position of the bit in the bitmap before returning the computed address to the core. We refer to this scenario as an allocation "HOT hit".

In the rare scenario that a zero bit cannot be found, indicating that the arena is currently full, the hardware will then use the available list pointer to load the next arena header into the HOT entry from the memory hierarchy (⑧). Meanwhile, the hardware replaces the existing entry by writing it back to the corresponding memory location using the *PA* field of the replaced entry. The hardware also performs two additional list operations. First, the full arena being replaced is inserted into the full list as the head. Furthermore, the newly loaded arena is removed from the head of the available list.

Finally, if the available list field indicates that no valid arenas exist, a new arena is allocated and loaded into the HOT entry by requesting more pages from the page allocator (⑨). The new arena is initialized following the initialization procedure described earlier. Finally, the current full arena is inserted into the full list. We refer to the last two scenarios as an allocation "HOT miss". As an optimization, the hardware allocator may *eagerly* load an available list arena or requests a new arena at the moment the last valid object of the current HOT entry is allocated. The hardware allocator can hide the potential latency of HOT misses in this manner.

Free. On a free request (⑩), the hardware identifies the appropriate arena by calculating the base virtual address and size class of the arena.This is performed by using the operand of `obj-free`, which is the virtual address of the object. The calculation only requires simple bit operations, as we will discuss shortly in Section 3.2.

After obtaining the size class, the arena's base address is compared to the VA field of the corresponding HOT entry (⑪). In the common case, the VA matches, and the free "hits" in the HOT. The appropriate bit in the bitmap is cleared, and the free operation completes (⑫). If the VA does not match, the free "misses" in the HOT, and the following steps are performed (⑬). First, the hardware allocator translates the arena's base address to the physical address by requesting a translation from the TLB. Next, the hardware allocator fetches the header from the memory hierarchy using the physical address from the previous step. Finally, the hardware clears the appropriate bit in the header's bitmap and writes the header back.

In the case of a HOT miss, if the arena is in the full list (having its all its bitmap set) before the free operation, then the arena will be moved to the available list. The hardware removes the arena from the full list and inserts it into the head of the available list by updating the appropriate pointers.

3.2 Hardware Page Management

As our study in Section 2.2 revealed, the cost of OS memory management operations can be significant in serverless functions. A major challenge to reduced this cost in hardware is how to efficiently manage the virtual and physical page assignments. To address this challenge, Memento introduces a lightweight page allocator that drastically reduce the cost of memory management costs induced by the OS.

The Memento page allocator is a hardware component located on the memory controller, and it interfaces with the object allocator for page-level operations. The two primary responsibilities of the page allocator are: (i) allocating arenas to the object allocator in the virtual address space, and (ii) managing a small pool of physical pages and using them to allocate arenas. Next we discuss how Memento manages virtual and physical addresses for arenas.

Managing Arena Virtual Addresses. For each process that uses Memento, the OS reserves a region of virtual addresses from the process's address space. The OS exposes the begin and end addresses of the region to the hardware through special *region control registers* on the core and the memory controller, which we refer to as *MementoRegion Start (MRS)* and *MementoRegion End (MRE)*. These two registers are managed in a multicore system at a per-address space level, i.e., each executing process maintains its private pair of registers backed by per-process memory locations. The OS is responsible for spilling and loading the region control registers into and from memory on context switches.

Memento divides the reserved virtual address region evenly into 64 size classes. This key design decision enables the hardware object allocator to calculate the size class and the arena's base address by simple bit operations. Given an object's address, the hardware calculates the size class by dividing the address offset in the region by 64. The arena base address is calculated by further rounding the offset within the size class down to the arena size of that particular size class. Note that the rounding can be implemented in hardware efficiently because the arena sizes are known in advance.

The first responsibility of the page allocator is to allocate virtual addresses to arenas. This scenario occurs when the object allocator

runs out of arenas and requests a new one from the page allocator. In Memento, an arena can consist of single or multiple pages depending on the particular size class of the arena. Therefore, the page allocator maintains a per-size-class pointer as the starting address of the next allocation. On receiving an allocation request for a new arena, the page allocator bumps the pointer of the requested size class forward by the arena size of that size class. In addition, a physical page is also eagerly allocated to back the first page of the arena containing the header. Both the virtual and physical addresses are sent back to the object allocator, which then uses them to initialize the VA field of the arena header and the PA field of the HOT entry, respectively.

On a multiprocessor system, the page allocator maintains per-size-class pointers for each core in a reserved memory block, and enables fast access to frequently used entries with a small on-chip cache called the *Arena Allocation Cache (AAC).* The AAC is direct-mapped using core IDs as indexes, and each entry stores the pointers for frequently used size classes. In practice, a small number of size classes per workload is sufficient to cover most of its memory allocation activities.

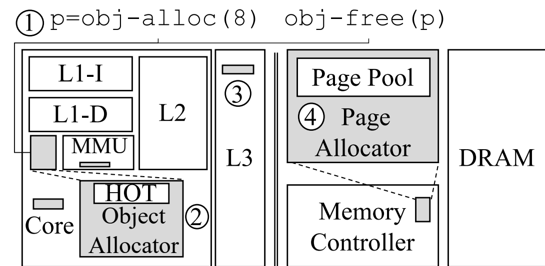

Figure 7: Memento Design Overview

Assigning Physical Pages to Arenas. The second responsibility of the page allocator is to assign physical pages to the arenas handed out to the object allocator. For this purpose, the page allocator implements two additional components. The first is a simple physical page pool consisting of free physical pages replenished by the OS on-demand. The second is a hardware-managed page table that tracks the virtual-to-physical address mapping for arenas. We next describe how Memento's hardware page allocator maps arenas to the physical address space.

As mentioned earlier, when the hardware page allocator gives out new arenas, it physically backs only their first page. This design decision is deliberately made to simplify the operation of the object allocator, as the object allocator will initialize the metadata located on the first page of the arena right after an arena is allocated. However, the page allocator does not back the rest of the arena's virtual addresses. Instead, physical pages are assigned to these virtual addresses only on the first access, reducing potential memory waste.

When arena memory is accessed for the first time, the MMU attempts to translate the accessed virtual address, which will incur a TLB miss and a subsequent page walk. Then the MMU will first check whether the virtual address lies in the process's reserved Memento address region by comparing the requested virtual address against the pair of region control registers. Supposing that the address lies in the Memento address region, the MMU then conducts the page walk from a different page table root address, specified by a *Memento Page Table Root (MPTR)* register (instead of the regular one, e.g., `CR3`). The MMU issues page walk requests marked with a special flag that will be identified by the page allocator. The hardware page allocator manages a Memento page table for each process similarly to the kernel. The only exception is that the page allocator constructs the Memento page table on page walk requests and automatically expands the table when encountering invalid entries.

On receiving the page walk request, the page allocator fetches and returns the Memento page table entry using the physical address in the walk request, if the entry is valid. If, however, the valid bit of the entry is 0, meaning the virtual address is currently not mapped, then one of the following happens before the page allocator returns the entry. In the first case, the walk is on the leaf level. The page allocator will proceed to allocate a new physical page and populate the leaf entry with the page's address. Otherwise, if the entry is higher up in the page table tree (e.g., it could be a PGD, PUD, or PMD entry on x86), the next level of the Memento page table is allocated from the pool and zeroed out (i.e., all entries are invalid). Eventually, all levels on the page walk path will be populated, after which the page walk concludes.

The TLB inserts the mapping after the page walk is complete. Future memory access will proceed as any other memory accesses. Moreover, future page walks on the arena address will not cause new allocation since both the Memento page table and the address are backed by physical memory.

Memento sets the permissions of all arena pages to readable, writeable, and non-executable. The page allocator sets this permission combination in the mapping entry returned to the page walker. While other combinations are also generally valid, in Memento, we choose to solely support heap memory allocation, where this combination alone is sufficient. This design decision dramatically simplifies Memento by avoiding complex use cases such as file-backed allocation.

The page allocator also handles arena frees, which occurs when the object allocator has freed the last live object within an arena. The page allocator walks the Memento page table and reclaims physical pages backing the arena. The corresponding page table entries are also invalidated, with page table pages freed if the last valid entry in it is invalidated.

When freeing an arena, TLB shootdowns are sent to cores that have issued page walk requests on the address space in the past. To identify which cores should receive the shootdown request, the hardware page allocator tracks per-process shootdown information using a hardware bit vector, the length of which equals the number of cores in the system. Note that typical serverless workloads are single-process and single-threaded, and as a result, this procedure's cost is negligible. Overall, the hardware page allocator in Memento enables applications to acquire memory from the Memento address region without incurring any OS costs, saving on both context switch and kernel code execution.

3.3 Main Memory Bypass

By exposing memory management semantics to hardware, Memento can enable previously difficult optimizations. To this end, Memento takes a first stab at this direction by leveraging allocation semantics in hardware to introduce a main memory bypass mechanism. Our main insight is that newly allocated objects do not need to be fetched from main memory and hence can be entirely instantiated in the cache hierarchy. This is safe because software should not have any expectation of data in newly allocated objects, hence Memento can simply provide a zeroed line.

The primary challenge lies in identifying which requests can bypass main memory. Since Memento is responsible for handling allocation and free operations identifying newly allocated objects is feasible. However, tracking which cache lines of an object can bypass main memory is difficult. This is because maintaining a full bitmap for every allocation would be prohibitively expensive.

As we revealed in Section 2.2, function allocations are short-lived. As a result, our insight is that allocations tend to have high reuse and enjoy high cache locality. Hence, it is important to bypass main memory at the first allocation. Furthermore, allocation metadata maintained in the arena header are densely populated and cold allocations are performed the first time an entry in Memento's arena header bitmap is set.

Based on these insights, Memento introduces an effective tracking mechanism that sequentially counts the lines of an allocation that have been accessed. To this end, Memento leverages the *Bypass Counter* in the arena header (Fig. 5) to track accessed cache lines. Specifically, cachelines with indices higher than the counter are guaranteed to have not been accessed before and hence it is safe to bypass main memory. The counter is only 11-bits which is sufficient to track the maximum number of cache lines in an arena. When there is an access to a line of an object allocated by Memento, the corresponding `Bypass Counter` is set to the index of the line if it is higher than the counter value. Similarly, the counter is decremented on a free if the index matches the counter. On an L1 miss, Memento's HOT identifies if this request is a main memory bypass request. To simplify integration with cache coherence, Memento allows the request to propagate regularly to the LLC where the line is instantiated instead of being fetched from DRAM. As a result, costly DRAM access for newly allocated objects are avoided.

3.4 Putting It All Together

Fig. 7 shows overall design of Memento with the main components depicted in gray. The OS exposes Mementomemory regions to the processor through control registers, *MRS* and *MRE*. Applications communicate with Memento through the ISA extensions `obj-alloc` and `obj-free` that allocate and free objects (①). The hardware object allocator (②) is responsible for managing the memory arenas and the corresponding arena lists. The MMU further maintains an additional control register, the *MPTR*, that enables page walks to traverse the page tables that belong to the Mementomemory region. Memento may instantiate a new object in the last level cache upon first access and bypass DRAM, if possible (③) Finally, to allocate new arenas, the hardware object allocator requests free pages from the hardware page allocator that lives in the memory controller (⑤).

4 DISCUSSION

Integrating With Software Memory Allocators. Memento focuses primarily on small object allocations within 512 bytes. Larger allocations, which are rare in serverless workloads, are handled by software. We propose two approaches that enable Memento to integrate with software allocators. The first approach is to let `malloc` check the allocation size. If the size is within 512, the software `malloc` will use Memento to fulfill the allocation. Similarly, `free` will check whether the object pointer to be deallocated lies in the Memento memory region. In this scenario, Memento executes the free. An alternative approach is to expose the address of the software allocation and free function calls and let Memento redirect control accordingly. For simplicity, our design opts for the first approach so that the existing `malloc`/`free` interfaces exposed to the application remain unchanged.

Interaction with Garbage Collection. Memento naturally supports garbage collection (GC). For example, both the Python and Golang frameworks that we use in our experiments have GC runtimes: they use reference counting and mark-and-sweep, respectively. Memento integrates seamlessly with both of these GC runtimes as follows: when the GC algorithm decides to free allocated objects, it uses Memento's `obj-free` interface to perform the free operation, thereby enjoying the same benefits from Memento's hardware support as non-GC'ed languages. Looking ahead toward potential future research, Memento creates interesting opportunities for extending GC algorithms to take advantage of Memento to manage ephemeral objects more efficiently. For example, although Memento does not help with tracking liveness, it could be integrated with an enhanced GC algorithm to help differentiate between ephemeral and non-ephemeral allocations. Once this distinction is made, the GC algorithm could leverage Memento to proactively free dead ephemeral objects before they create too much cache pressure rather that waiting to free objects when there is too much memory pressure. We leave this exploration to future work.

Multi-core Support. Current serverless workloads are single-process, but multiple independent functions can be collocated on the same server. To support multi-tenancy, the OS flushes the HOT table before a process is context switched. As we will see in Section 6.6, this operation is inexpensive as the HOT table is small.

Although today's serverless function workloads are typically single-threaded, Memento does support multi-threaded applications. Existing software allocators rely on locks to atomically modify allocation metadata. To avoid frequent synchronization overheads, modern allocators often leverage per-thread pools. In Memento, we follow a similar design where each thread in Memento manages its own arena whose virtual address range is maintained by hardware. As a result, allocations within each thread's own arena are managed the same way as in a single-threaded case. Note that since allocations are performed within a thread's arena there are no races on the allocation path.

The interesting scenario is when an object that is allocated by one thread is deallocated by a different thread. In practice, we do not expect this to be the common case, because it would require passing pointers through shared memory between threads; while this is possible, allocated objects are more typically deallocated by

the same thread. The first step in handling this case is recognizing that the object was allocated by a different thread, which Memento does by comparing the virtual address of the object with the virtual address range of its own arena (which is maintained by hardware). Once this scenario is recognized, Memento can use either software-assisted or hardware-only approaches to deallocate the object. To amortize the overhead of invoking a software handler, Memento can batch these non-local free operations in a thread-local buffer and only signal the software to perform batch deallocation when the buffer becomes full or when a context switch occurs. The software handler manages such deallocations in the same manner as existing software allocators to perform the deallocations atomically and avoid potential races.

For the hardware-only design, Memento can leverage existing cache-coherence mechanisms to perform deallocations inexpensively without software locks. Instead, Memento performs the read-modify-write operation in the HOT atomically. In particular, if the object lies outside the current thread's arena, a deallocation is performed by the local HOT by first issuing a *BusRdX* to the cache hierarchy to acquire exclusive ownership of the arena's metadata header (Fig. 5). The local HOT then updates the metadata header as in a regular free. Metadata headers are tracked by the existing coherence protocol of the system as normal data. For example, when a HOT receives a coherence invalidation message to one of its entries, the HOT entry will be invalidated and supplied to the request. Overall, this design avoids potential races as the arena header entry is in the *dirty* state in the private cache of the core that performs the free, and it relies on existing hardware cache-coherence mechanisms that provide write serialization. Finally, in the case of page allocations, the hardware page allocator simply serves requests in a first-come first-serve order based on the serialization of requests provided by the interconnect.

In summary, we avoid data races in Memento through the combination of two things. First, because each thread allocates from its own arena, we avoid potential races that might arise from multiple threads attempting to allocate from the same arena. Second, we avoid potential data races in setting bits in the allocation metadata by performing these read-modify-write operations atomically in the HOT on metadata cachelines that are already in the dirty state (leveraging the normal cache coherence protocol). Page allocations in the memory controller are serialized by the interconnect. With this support, both allocations and deallocations avoid race conditions in Memento. Other multi-threaded challenges such as double-free attempts due to application bugs are independent of the allocation implementation in software or in Memento's hardware. In Memento such cases are handled graciously by raising an exception to software.

5 EVALUATION METHODOLOGY

Simulation platform and Hardware Cost. We evaluate Memento with full-system simulation using QEMU [5] integrated with the SST [44] and DRAMSim3 [33]. The simulated architecture is presented in Table 3. We use Linux kernel 5.18. We further show the two main hardware structures of Memento, the hardware object table (HOT) modeled as a 3.4KB direct-mapped cache and the arena allocation cache (AAC) of the hardware page allocator modeled as a 32-entry direct-mapped cache. We evaluate hardware energy and area cost of HOT and ACC using CACTI 6.5 [38] under 22nm technology node. The results are shown inline in Table 3. Overall, the hardware cost of Memento is minimal.

CPU	4-issue OOO, 3 GHz, 256-Entry ROB, 64-Entry LSQ
TLB	L1 64-Entry, 4-Way; L2 2048-Entry, 12-Way
L1d	32KB, 8-Way, 2 Cycle, LRU Replacement
L1i	32KB, 8-Way, 2 Cycle, LRU Replacement
HOT	3.4KB, Direct-Mapped, 2 Cycle, 1.32mW, 0.0084mm2
L2	256KB, 8-Way, 14 Cycle, LRU Replacement
LLC	2MB Slice, 16-Way, 40 Cycle, LRU Replacement
AAC	32-Entry, Direct-Mapped, 1 Cycle, 0.43mW, 0.0023mm2
DRAM	64GB, DDR4 3200, 16 Banks
OS	Ubuntu 20.04

Table 3: Simulation configuration.

We instrumented both userspace and kernel functions to capture memory management routines. For Python workloads, we instrumented CPython 3.8 [42] heap allocation functions. For C/C++ workloads, we link them against an instrumented `jemalloc` [15, 21]. For Golang workloads, we instrumented heap memory allocation and garbage collection functions in go-1.13 and linked them against Golang binaries. To capture page faults, we instrumented the page fault handler functions in Linux. We also instrumented system calls to `mmap` and `munmap`. The simulator replaces calls to memory allocation and free functions with the corresponding ISA extensions of Memento. Allocations of size greater than 512 bytes and addresses outside of the reserved address region are handled by software without Memento's intervention.

Benchmarks. To evaluate Memento, we use a total of fourteen benchmarks. We focus our evaluation on functions that exhibit at least 0.5 MallocPKI (malloc per kilo instructions) on average. All functions executes within a `crun` [12] container. We use function workloads from FunctionBench [28], and the serverless benchmark suite SeBS [11]. `dynamic-html(dh)`, `image-recognition (ir)`, `graph-bfs(bfs)` , `dna-visualisation (dna)` are from SeBS, and `pyaes (aes)`, `feature_reducer (fr)` are from FunctionBench. We further perform experiments with pyperformance [40] using memory management benchmarks `json_loads (jl)`, `json_dumps (jd)`, and `mako (mk)`. In addition, we adapted `UrlShorten (US)`, `UserMentions (UM)`, `ComposeMedia (CM)` and `MovieID (MI)` from DeathStarBench [19], written in C++, into function-like units [49]. We ported the Python `dynamic-html`, `graph-bfs`, and `pyaes` functions to Golang (`dh-go`, `bfs-go`, and `aes-go`). Before simulation begins, we perform a system-level only warm-up and then we simulate functions workloads from the beginning to completion. Beyond function execution, serverless computing involves a significant platform software stack [10, 12, 13, 30, 41], including those providing container runtime, orchestration, event routing, and other functionalities. To this end, we evaluate the OpenFaaS [41] serverless platform when performing three important operations, namely `up` that starts up the serverless platform, `deploy` that registers a function in the function store and prepares it for execution, and `invoke`

that routes a request to an instance of the function. For these operations, we send a request for these operations to OpenFaaS Gateway after the platform has been warmed up and simulate the regions of interest of each operation. We further evaluate four long-running data processing applications, including two key-value stores, Redis and Memcached, and two in-memory databases, Silo and SQLite3. We use the value size distribution from [37] to drive the workloads and perform measurements at the steady state.

All workloads use the default input if possible. Workload execution time ranges from sub-second level to a few seconds, and memory consumption numbers are from low to tens of MBs. Functions communicate via RPCs with a Redis backend to fetch inputs and store results at the beginning and end of function execution. We measure RPC costs to be at between hundreds of microseconds to a few milliseconds across our workloads depending on input/result size. This is a small portion of the total function runtime. RPCs have been the focus of recent works [31].

6 EVALUATION

6.1 Speedup

Fig. 8 shows the execution time reduction of Memento as the speedup over the baseline system. We group results in Fig. 8 by the language runtime (Python, C++, Golang). We show the average of functions as *func-avg*. We further distinguish between data processing applications (*data-avg*) and serverless platform operations (*pltf-avg*).

Overall, Memento achieves substantial speedups between 8–28% and 16% on average for functions. These are substantial gains achieved by Memento for serverless functions. Data processing applications and serverless platform operations also enjoy substantial gains between 4-11%.

To further study the source of speedups, we present in Fig. 9 a breakdown of the performance gains of Memento. The figure shows four main sources of gains that can be achieved by Memento: (i) hardware object allocations (*obj-alloc*), (ii) hardware object frees (*obj-free*), (iii) hardware page management operations (*page-mgmt*), and (iv) main memory bypass (*bypass*). Memento can satisfy memory management operations with latency equivalent to a single cache access in most cases, and hence significantly reduces function execution time. On average for functions, 33% of the gains are attributed to hardware object allocation and 32% to free. The main memory bypass mechanism attains a 2% improvement on average, and it can reach up to 17% improvement depending on the function's sensitivity to bandwidth. The hardware page management component is responsible for 33%. In data processing applications the gains are primarily split between object allocation and page management, that account for 37% and 58% respectively. In the case of the serverless platform operations, the majority of the gains, 71%, come from object allocations with a smaller percentage being attributed to page management. The results are similar within workloads and operations across the two environments hence we show only the average.

Individual function workloads demonstrate different gains from either or both sources, further highlighting the need for both hardware object and page management proposed by Memento. Seven of the nine Python workloads and two out of three Golang workloads get at least 40% of the gains from Memento's hardware page management. This observation is explained by the larger heap size, which causes a higher number of page faults that increases proportionally to the number of pages in the heap memory. The majority of the page management gains come from dynamic page allocations in hardware that eliminate costly OS page fault handling. In the case of `aes` and `jl`, more than 90% of the gains come from hardware object management. This is because their working sets are much smaller, shifting the critical path from page allocation to object allocation.

In the case of data processing workloads Memento provides significant benefits between 5–11%. Redis benefits the most from Memento, achieving an 11% throughput increase on mixed PUT-GET (50% each) workload. We attribute the result to the usage of the SDS string library, which allocates strings on the heap to store keys and values and as temporary buffers. Both Memcached and Silo saw moderate throughput increase of 6.5% and 7.5% respectively. Both applications experience frequent page faults on heap memory, resulting in significant cycle reduction in the kernel with Memento. SQLite3 allocates many small and short-lived objects when parsing SQL queries. Consequently, SQLite3 can also benefit from Memento with 5% improvement when processing SQL `SELECT` statements. The above study reveals that Memento is also applicable to long running workloads that often rely on short-lived allocations. The gains are due the ability of Memento to improve both userspace allocations and kernel memory management that account for 38% and 62% of the memory manangement cycles respectively in the case of data processing applications. Serverless platform operations also enjoy significant speedups with Memento between 4%-7%. The gains are similar across operations, with 59% being attributed to userspace allocations and 41% to kernel memory management.

The main memory bypass mechanism can also be beneficial for certain workloads such as `dh` as it saves 6% of execution cycles leading to a 28% function speedup. The DeathStarBench C++ workloads show significant speedups of 16% on average. The majority of the gains come from hardware object management. The page management gains are smaller because jemalloc pre-maps and pre-faults a small pool of memory during library initialization. This mechanism instead turns object allocation and free operations into a performance bottleneck that Memento addresses. Overall, the results highlight the importance of Memento's mechanisms.

Iso-storage Comparison. Finally, we compare Memento to an iso-storage architecture where the HOT storage is provided to the L1 cache instead. Specifically, we consider a hypothetical 9-way L1D cache that incurs the same SRAM overhead as the HOT while maintaining the same latency. Our results show that while dedicating Memento's HOT storage to the L1D results in an overall speedup of 3%, this is significantly smaller than the 28% speedup provided by Memento.

6.2 Memory Bandwidth Savings.

To study the effect of Memento on main memory traffic, we present results on normalized memory bandwidth savings in Fig. 10. Overall, Memento reduces memory bandwidth usage by 30%. For two workloads, `UM` and `CM`, Memento achieves a 31% and 35% reduction

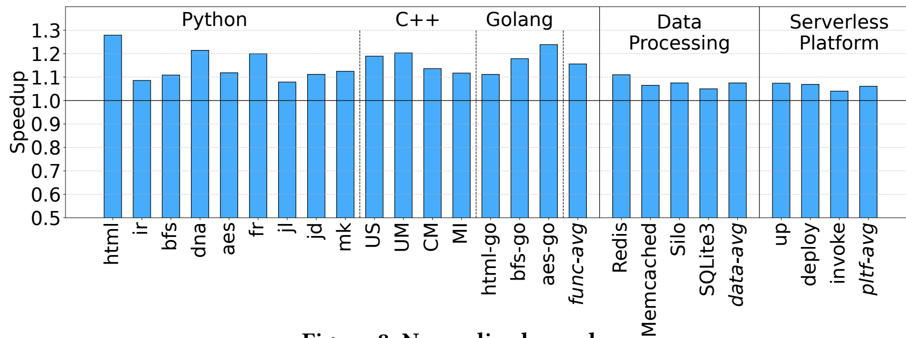

Figure 8: Normalized speedup.

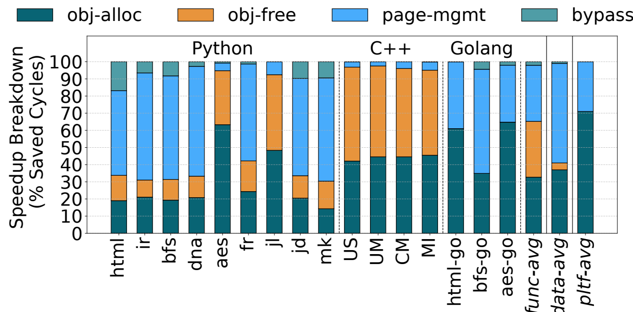

Figure 9: Performance gains breakdown.

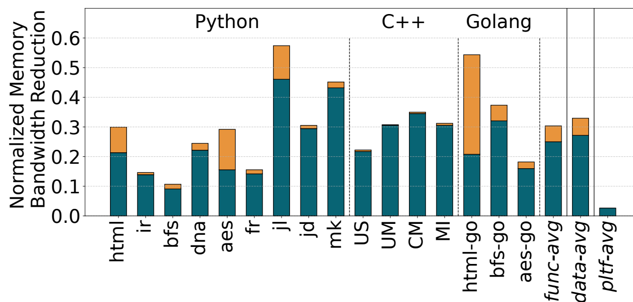

Figure 10: Normalized memory bandwidth usage reduction. Yellow highlights main memory bypass savings.

of DRAM traffic for functions. The gains for the serverless platform are smaller. In the case of data processing applications Memento significantly reduces memory bandwidth usage by 33%. Memento's bandwidth savings are attributed to three sources. First, the hardware object table (HOT) absorbs allocation traffic to memory as objects can be instantiated in the cache without adding unnecessary traffic to the main memory. Furthermore, Memento eliminates the instruction and data movement caused by memory management in userspace and the kernel. Finally, Memento's main memory bypass mechanism provides provides savings of 5% on average and up to 34%.

6.3 Aggregate Main Memory Usage.

Fig. 11 presents normalized aggregated memory usage results. We measure aggregated memory usage as the total number of physical pages allocated during simulated execution. We present normalized user and kernel space memory separately in addition to the total memory usage. Memento achieves an overall 15% reduction in aggregate memory usage for functions. When we look at the userspace and kernel usage, we see that userspace reduction is about 10% and kernel is 28%. In the case of serverless platform operations the memory usage remains almost the same. Furthermore, looking at the aggregate memory usage for data processing applications, we see that Memento achieves significant savings of 5% in userspace, 50% in the kernel, and total savings of 23%.

By examining the behavior of functions, we see that Memento increases userspace memory usage for Python and Golang workloads. A primary reason is that the software allocators for these two languages share free pages among size classes, which reduces external fragmentation. We choose not to include this potential optimization and trade-off userspace memory instead for a less complicated hardware design. Nonetheless, Memento reduces kernel memory usage for these workloads by 29%. In workloads like dh, Memento can achieve major kernel memory usage reductions, more than 60%, due to the reduction of kernel metadata needed to manage memory regions. For C++ workloads from DeathStarBench [19], Memento achieves 41% userspace memory savings. We attribute this to the low utilization of jemalloc's memory pool, which ends up wasting userspace memory resources. Memento instead can dynamically respond to the memory usage of each function. Furthermore, Memento achieves significant kernel savings for DeathStarBench, with an average of 25%.

6.4 Characterizing Memento

Hardware object table hit rate. We present the hit rate of the hardware object table (HOT) in Fig. 12. On allocations, a HOT hit happens when a request finds an available entry in the cached header bitmap. Similarly, on free operations, a HOT hit happens when the cached header can fulfill the free request without incurring additional memory operations. A miss in the HOT causes a cache request to be issued to load the appropriate entry from memory. Hits in the HOT are completed in two cycles without issuing memory requests.

Overall, allocations in Memento enjoy a high hit rate of 99.8% for functions. Furthermore, they exhibit a uniform behavior across functions as well as data processing applications and serverless platform operations. Free operations show an average hit rate of 83%. We observe that Python workloads generally exhibit lower free

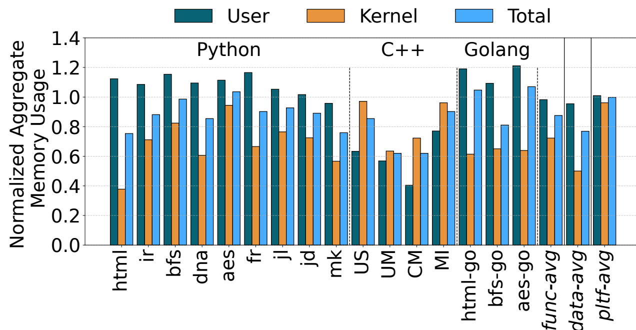

Figure 11: Normalized aggregate memory usage.

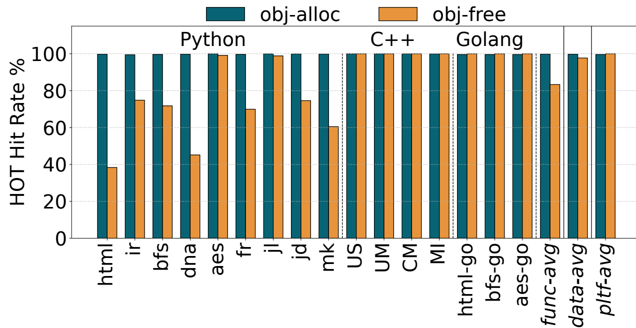

Figure 12: Hardware object table hit rate.

hit rate, while Golang and C++ workloads show a very high free hit rate. Further investigation reveals that C++ workloads operate on tight loops with objects being allocated at the beginning of the loop and freed at the end. Golang workloads, on the other hand, rely on garbage collection to batch-free objects. Therefore, they never need to call free on individual objects. For Python workloads, some of the allocations are made by the interpreter to store global information that tends to live much longer than other local objects (e.g., those created in tight loops). These long-lived objects cause low HOT hit rates. Nevertheless, Memento handles them correctly by performing the free operation out of the execution critical path. As a result, Memento is still able to eliminate their overheads successfully, despite low HOT hit rates.

The results further corroborate the insights from Section 2.2 as function workloads are short-lived in nature with short malloc-free distances. When an object is allocated from the HOT and freed shortly afterward, the free request will very likely result in a hit. Consequently, Memento achieves a high HOT hit rate even with a small direct-mapped structure.

While not shown in the figure, Memento's arena allocation cache (AAC) enjoys uniformly high hit rates as only a few size classes are utilized in each of the workloads.

Arena list operations. Fig. 13 presents the frequency of linked list operations for arena management during allocation and free. These operations occur when the object allocator puts arenas on/off the full/available arena lists (Section 3.1). We characterize them by calculating the percentage of allocations or frees that include arena list operations. For all workloads, less than 1% of allocations

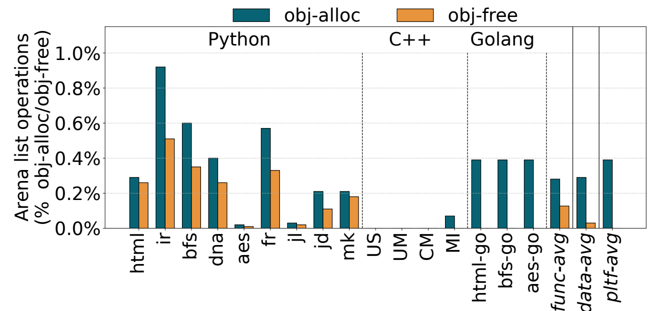

Figure 13: Arena list operation frequency.

and 0.6% of frees need to operate on linked lists. Workloads with smaller working sets require fewer linked list operations because there are fewer arenas to manage, and the locality of allocation and free is high. Compared with the total number of instructions, the linked list operations only constitute a tiny fraction of total memory accesses ($\leq 0.01\%$). We also evaluated an ideal environment without the linked list operations and saw minimal changes in the result. Overall, the performance implications of arena list operations are negligible.

6.5 Function Pricing

We compute the cost of running functions on both the baseline system and Memento according to AWS pricing [4]. Fig. 14 shows the runtime cost based on execution time and memory usage. The current AWS pricing policy computes the cost of functions in the granularity of milliseconds for runtime and MB for consumed memory. On average, we can see from the results that Memento can achieve a significant runtime cost saving of 29%. Function providers add a fixed per-invocation cost that accounts for the infrastructure management and deployment costs outside the function costs. While this cost is outside the scope of Memento, when included for end-to-end pricing cost, Memento is able to achieve cost savings up to 31%, and 11% on average.

6.6 Sensitivity Studies

Populating pages on mmap. `mmap` supports a flag that will force the OS to eagerly populate virtual pages with physical memory (`MAP_POPULATE`) , hence reducing the page fault overhead. However, eagerly populating pages may cause an increased physical memory footprint. To evaluate the effect, we instrumented the software allocators under comparison to pass the flag to `mmap` and evaluated Memento on this setting. For Golang workloads we observed that on average performance improved by 3% from the baseline but physical memory footprint increased by 8.6×, due to the large `mmap` size of Golang allocator. For Python and C++ workloads, we did not observe any significant change in speedup, with physical memory increasing by an average of 9.6%. We conclude that eagerly populating allocations is not cost-efficient for serverless functions based on the AWS pricing model in Section 6.5 due to the increased physical memory footprint.

Multi-process environments. To evaluate Memento in multi-process environments, where a single core is over-subscribed by

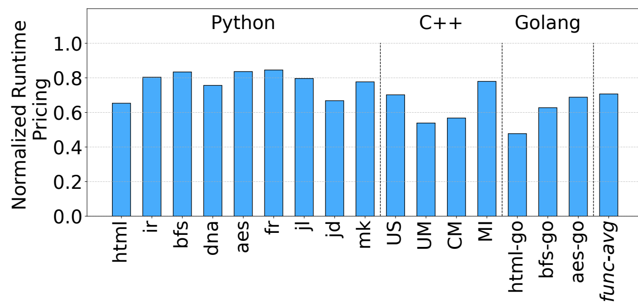

Figure 14: Normalized function runtime pricing.

several time-sharing function instances, we ran the simulation by starting four randomly selected function instances and pinning them to the same core. We repeated the experiment ten times, with different workloads each time. The main potential overhead of Memento is the flushing of the HOT table. Specifically, the core issues flush operations for every entry in the HOT table. Compared with the typical context-switch time that is in the order of microseconds, and the frequency of context switches that is in the order of a few milliseconds, the overall performance effect is negligible.

Tuning software allocators. We manually tuned the available configuration knobs of software allocators to study their effects on Memento. Our results show that while changing most of the knobs does not affect Memento's speedup, enlarging the arena size of software allocators causes a noticeable but less than 1% speedup. Further investigation revealed that using larger arenas reduces the frequency of `mmap` at the potential cost of fragmentation. Physical memory footprint is unaffected as `mmap` reserves physical pages lazily.

Fragmentation. To identify potential fragmentation induced by Memento, we measured fragmentation as the percentage of actual amount of memory allocated to small objects and the total amount of memory given to HOT. Our results show that on average, only 3.68% of the slots in the arena headers are not active. (This result is the combination of fragmentation and free memory, since it is difficult to disambiguate between the two.) We further compared these fragmentation results to the software-only allocators, and observed similar results within a ± 2% of hardware design across applications. Meanwhile, Memento reduces the overall memory usage as we discussed in Section 6.3.

Warm-start versus cold-start. Our earlier experiments warm-started functions, i.e., functions are executed by an existing container process and skip the container set-up stage. However, in practice, functions may also experience cold-starts where the latency of setting up containers are added to their execution latency. To study the effect of cold-started functions on Memento, we ran experiments where containers are set up before executing the function. Our results show that, even with cold-starts, Memento can still gain a speedup of 7%–22% compared with the baseline.

6.7 Comparison with Related Work

The mostly closely related work to Memento is Mallacc [26], which introduces hardware extensions to accelerate certain userspace malloc operations. In contrast, Memento is a holistic hardware design that accelerates both userspace *and* kernel memory management, both of which are very important for performance (as shown in Fig. 9). By exposing memory allocation semantics to the hardware, Memento is able to explore previously hard-to-perform optimizations such as main memory bypass. In addition, while Mallacc is hardwired to TCMalloc and only supports C++ workloads, Memento is agnostic to the language runtime, as demonstrated by our integration of Memento with Python, C++, and Golang.

To provide a quantitative comparison, we simulate an *idealized* version of Mallacc where the Mallacc cache has zero latency and always hits. Because Mallacc only supports C++ workloads, we only compare it to Memento on DeathStarBench. Whereas this idealized Mallacc configuration achieves speedups ranging from 5-10% (8% on average), Memento *doubles* these performance gains with speedups ranging from 12-20% (16% on average). At the same time Memento provides substantial gains (between 8-28%) for other language runtimes (Python and Golang) that Mallacc cannot support. Memento's design for kernel memory management in hardware is especially crucial for high-level languages such as Python and Golang (as shown in Section 6.1), and this is not supported by Mallacc. Memento successfully eliminates the kernel overheads that account for 33% on average and up to 68% of function memory management overheads.

7 RELATED WORK

Reducing Cold Starts for Serverless Functions A body of work has focused on reducing the cold start effects of serverless workloads by focusing on system-level effects such as creating VMs and containers. These works seek to reduce the cold start latency using process snapshots [6, 14, 39, 57], lightweight kernels [2], record-and-replay [56], application-level sandboxing [3], low-latency serverless frameworks [23], and live container caching [18, 48, 52]. These approaches motivated by the short-lived function execution and are orthogonal to Memento and can be adopted together.

Hardware Memory Management Prior work has explored a simplified version of page management with hardware buddy allocators [7–9, 34, 35]. These works aim to provide physical memory management for embedded platforms with limited OS support or simple runtimes. Instead, Memento relies on a fully-fledged OS and only manages a small pool of physical pages used for arena allocations in tandem with the hardware object allocator. We have discussed Mallac [26] extensively in Section 6.7.

Memory Management in Datacenters The cost of memory management in datacenters has been under active investigation [20, 25]. Recent work by Google [20] has shown that the cost of memory allocation remains high in data processing applications. In this work, we have shown that Memento can be beneficial for data processing applications by eliminating wasted cycles in userspace memory allocations and offloading expensive kernel operations to hardware. By eliminating the compound effects of memory management across the stack, Memento can achieve major efficiency gains for datacenter workloads. Finally, by exposing memory management semantics to hardware, future work on accelerator design for other prevalent datacenter operations such as compression, croptography, protocol buffers [27], and RPC [31] can benefit significantly by chaining

multiple accelerator operations together while performing memory management with Memento.

8 CONCLUSION

This paper presented Memento, a novel hardware-centric design that alleviates the overheads of serverless memory management. Memento introduces two key mechanisms in hardware for object allocation and page management that operate in tandem and alleviate both userspace and kernel memory management overheads. It further leverages the exposed memory management semantics in hardware to introduce a main memory bypass mechanism for newly allocated objects. Our evaluation shows that Memento reduces main memory traffic by 30%, speeds up function execution by 8-28%, and further reduces the runtime pricing cost of functions by 29%. Finally, we demonstrated the applicability of Memento beyond functions, to major serverless platform operations and long-running data processing applications.

ACKNOWLEDGMENTS

This work was funded in part by NSF grants CNS-2107307, CNS-2239311 and a Meta Faculty Award. We thank Gennady Pekhimenko and the anonymous reviewers for all of their valuable feedback.

REFERENCES

[1] 2023. Python Memory Management and pymalloc, home page: https://docs.python.org/3/c-api/memory.html.

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

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, USA, 923–935.

[4] Amazon. 2023. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 10–5555.

[6] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[7] Hasan Cam, Mostafa Abd-El-Barr, and Sadiq M Sait. 1999. A high-performance hardware-efficient memory allocation technique and design. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 274–276.

[8] J. Morris Chang and Edward F. Gehringer. 1996. A high performance memory allocator for object-oriented systems. *IEEE Trans. Comput.* 45, 3 (1996), 357–366.

[9] J Morris Chang, Witawas Srisa-An, and C-TD Lo. 2000. Architectural support for dynamic memory management. In *Proceedings 2000 International Conference on Computer Design*. IEEE, 99–104.

[10] containerd. 2023. containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io.

[11] 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.

[12] crun. 2023. crun container official github repo. https://github.com/containers/crun/.

[13] Docker. 2023. Docker: Accelerated Container Application Development. https://www.docker.com.

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

[15] Jason Evans. 2006. A scalable concurrent malloc implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[16] Tais B. Ferreira, Rivalino Matias, Autran Macedo, and Lucio B. Araujo. 2011. An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*. 92–98. https://doi.org/10.1109/PDCAT.2011.18

[17] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[18] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 386–400.

[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[20] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. 2023. Profiling Hyperscale Big Data Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 47, 16 pages. https://doi.org/10.1145/3579371.3589082

[21] jemalloc. 2023. jemalloc official github repo. https://github.com/jemalloc/jemalloc/.

[22] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/3477132.3483541

[23] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3445814.3446701

[24] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. https://doi.org/10.1145/3448016.3459240

[25] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[26] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating memory allocation. *ACM SIGPLAN Notices* 52, 4 (2017), 33–45.

[27] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 462–478. https://doi.org/10.1145/3466752.3480051

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

[29] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[30] Knative. 2023. Knative. https://knative.dev/docs/.

[31] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: Towards Efficient RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Virtual).

[32] Sangho Lee, Teresa Johnson, and Easwaran Raman. 2014. Feedback Directed Optimization of TCMalloc. In *Proceedings of the Workshop on Memory Systems Performance and Correctness* (Edinburgh, United Kingdom) *(MSPC '14)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2618128.2618131

[33] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[34] Wentong Li, Saraju Mohanty, and Krishna Kavi. 2006. A page-based hybrid (software-hardware) dynamic memory allocator. *IEEE Computer Architecture Letters* 5, 2 (2006), 13–13.

[35] Wentong Li, Mehran Rezaei, Krishna Kavi, Afrin Naz, and Philip Sweany. 2007. Feasibility of decoupling memory management from the execution pipeline. *Journal of Systems Architecture* 53, 12 (2007), 927–936.

[36] Linux manual. 2023. mmap(2) - Linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html.

[37] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. 2021. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 243–262.

[38] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009), 28.

[39] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.

[40] CPython official repo. 2023. The Python Benchmark Suite. https://github.com/python/pyperformance/.

[41] OpenFaaS. 2023. OpenFaaS. https://www.openfaas.com.

[42] Python. 2023. CPython official github repo. https://github.com/python/cpython/.

[43] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2022. Reinforcement Learning for Resource Management in Multi-Tenant Serverless Platforms. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems* (Rennes, France) *(EuroMLSys '22)*. Association for Computing Machinery, New York, NY, USA, 20–28. https://doi.org/10.1145/3517207.3526971

[44] Arun F Rodrigues, Gwendolyn Renae Voskuilen, Simon David Hammond, and Karl Scott Hemmert. 2016. *Structural Simulation Toolkit (SST).* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[45] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[46] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[47] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3472883.3486972

[48] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 753–767. https://doi.org/10.1145/3503222.3507750

[49] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. 2022. Lukewarm serverless functions: characterization and optimization.. In *ISCA*. 757–770.

[50] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (apr 2021), 76–84. https://doi.org/10.1145/3406011

[51] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296

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

[53] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.

[54] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 138–152. https://doi.org/10.1145/3472883.3486981

[55] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 34, 15 pages. https://doi.org/10.1145/3579371.3589069

[56] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[57] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3302424.3303978

[58] 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 768–781. https://doi.org/10.1145/3503222.3507709

[59] 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* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739. https://doi.org/10.1145/3477132.3483580

[60] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3567955.3567960