Allocation-aware Cache Management Policy In Memento

Pei Li
peili@andrew.cmu.edu

15-618 Project, Fall 2022

1 Introduction

The design of the cache hierarchy remains relatively static while the capacity keeps scaling. The traditional load-store interface of the memory hierarchy is not sufficient to react to runtime dynamic software behavior and misses opportunities for optimization based on such information.

One reasonable design option is to let allocation requests drive prefetching decisions, which would work well for workloads in which an object will be initialized shortly after allocation returns the object's address. In addition, we observe that the first memory operation to a newly allocated object is almost always a write because, according to the semantics of the user space memory allocator (i.e., `malloc`), newly allocated objects contain undefined data.

Memory bandwidth is a long standing performance issue. However, such kind of bottleneck can be hidden with adequate hints from software and enabling prefetching from DRAM to different levels of cache[2].

Memento is an ongoing project aiming at providing architectural support for alleviate ephemeral memory management overheads in serverless computing environment. Thus it offers the opportunity to amortize the cost of memory allocation and deallocation. Memento explores memory management by a hardware-centric solution, and achieves a significant speed-up of function execution time. Memento handles memory allocation whose size is below 512 bytes and keeps track of each allocation as its metadata.

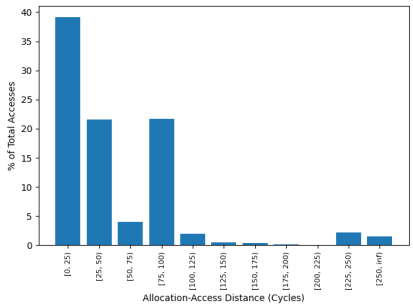
2 Summary

We modified SST(Simulation Structural Toolkit) to implement our allocation-aware cache management policy including a DRAM bypass policy and a hardware object prefetcher that work in tandem, and evaluate its effectiveness on representative serverless workloads. In summary, we make the following contributions:

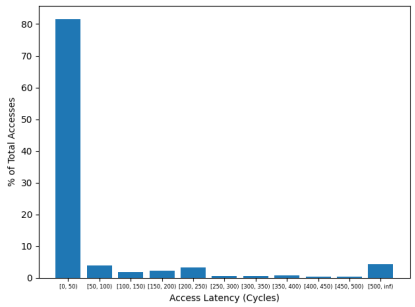
- We profiled the allocation-access work pattern for serverless workloads of Memento. We identified prefetch opportunities by studying these work patterns.
- We proposed a DRAM bypass policy at LLC (Last Level Cache) that creates cache lines directly in LLC without fetching from DRAM for each memory allocation from Memento.
- A new type of `PREFETCH` request that is triggered by memory allocation and an allocation-aware hardware object prefetcher (HOP) works inside L2 cache that handles `PREFETCH` request and prefetches every object allocation from Memento.

3 Background and Motivation

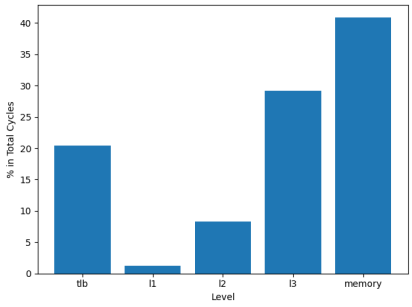
To study the memory allocation and access behavior of serverless benchmarks, we need to observe the behavior of different workloads when allocating memory and accessing memory allocations. Therefore we conducted



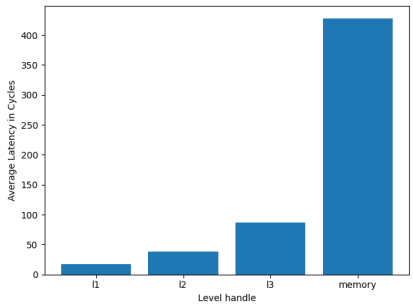
(a) Allocate-access distance



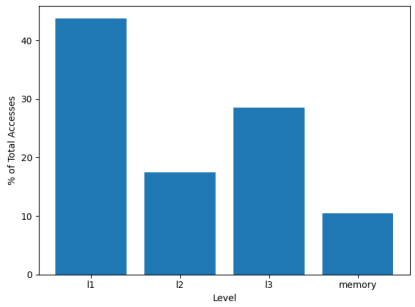
(b) Access latency for allocations



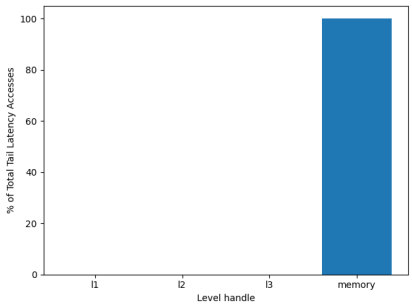
(c) First memory allocation access cycle breakdown



(d) Average single access latency inside memory hierarchy



(e) Level of first access requests handled in memory hierarhcy



(f) Components of first allocate-access tail latency

Figure 1: Identify Prefetch Opportunities

preliminary experiences on the dynamic-html benchmark in SeBS[3]. Our detailed analysis demonstrates that DRAM access is the hindrance to most latency.

3.1 Result 1: How long does it take for allocation to be accessed for the first time?

First, we measure **allocate-access distance**, which is the distance (in cycles) between the time of an allocate request created and sent to cache, and the first time accessing cache lines inside the corresponding memory allocation, which is usually for initialization. To identify which cache lines belong to which allocation, we keep track of a list of virtual addresses memory allocation that is handled by Memento, and the physical addresses they mapped to. Assuming that the size of each cache line is 64 bytes, we can calculate the base address of this allocation inside cache line. This metric is important because we expect prefetching requests to be finished in this interval. Otherwise before the prefetching request finishes, the access request will go to MSHR and stall.

Fig. 1a shows the distribution of the allocate-access distance. The result shows that after data was allocated, it will be accessed soon after a very short amount of time. Specifically, 78.25% of these allocations will be accessed within the next 5 cycles to 15 cycles.

3.2 Result 2: How long does it take to handle those access?

Second, we measure **access latency**, which is the time interval (in cycles) it takes from the first memory access request sent to cache to this request was finished and received by CPU. Fig. 1b shows the access latency. The results show that the majority of first access requests are handled within the first few hundred cycles. In particular, 83.57% of these requests are handled within the first 250 cycles.

Fig. 1b shows the first-access latency. The results show that the majority of first access requests are handled within the first few hundred cycles. In particular, 83.57% of these requests are handled within the first 250

cycles.

3.3 Result 3: Where is time spent throughout the memory hierarchy?

Third, we measure **cycle breakdown**, which is the percentage of cycles spent on accesssing each component of the memory hierarhcy. 1c shows that the majority of the latency comes from DRAM access and TLB misses, which eventually perform page walk to access DRAM.

The majority of the latency comes from DRAM access and some of them lies inside the cache hierarchy itself. These are the latency that we want to avoid. Notably, the last bar in Fig. 1b indicates the accesses that needs to perform a page walk. The need to go to DRAM to fetch the necessary structures inside the Radix page table hierarchy (PGD, PUD, PMD, PTE) introduces significant latency, ranges from 1000 to 2000 cycles. Third, we measure the execution time for each first allocation-access request and breakdown results to show where time are spent throughout the memory hierarchy.

3.4 Result 4: Where and how long in the memory hierarchy are first time access requests to allocations handled?

Forth, we dive into each access requests to understand where inside the memory hierarchy are they handled. Fig. 1d shows that comparing to DRAM access, latency most access requests in cache hierarchy are not magnificent.

Fig. 1e shows that the majority of access requests are handled inside cache hierarchy. However, still around 10% of requests require access to memory, which are very expensive.

Furthermore, we studied requests with extremely high latency (over 500 cycles), which we refer to as "tail requests". As expected in Fig. 1f, the majority of tail requests are caused by direct access to memory. A small number of tail requests are caused by page walks, which are slow but relatively rare. These results highlight the importance of optimizing access to memory in order to improve performance and reduce the occurrence of tail requests.

3.5 Result Summary

Overall, these results highlight the importance of optimizing DRAM access in order to improve the performance of serverless computing environments. By implementing our allocation-aware cache management policy and prefetching strategy, Memento is able to significantly reduce the time spent in DRAM access and improve the overall performance of serverless functions.

4 Design and Implementation

To leverage the observations in Section 2, we sketch the design of an allocation-aware cache management policy that consists of DRAM bypass policy and a hardware object prefetcher that collaborates with Memento as follows.

4.1 Eliminate DRAM access latency

We start with directly create cache blocks in the hierarchy, in a way that resembles "Overlay-on-Write" without actually fetching data from the main memory. We will add an extra byte (8 bit) metadata for each memory allocation in Memento to record whether this is the first time accessing a memory allocation. A whole byte is necessary because each cache line can only accommodate 64 bytes, while the maximum possible size of allocation in Memento is 256 bytes.

For each request arrives at LLC, LLC will check whether this request is tagged as first access of memory allocations. If so, it will magically "create" a cache line for this request and return it to upper level cache by choosing a victim line based on its cache replacement policy (LRU).

The correctness of this creation can be maintained due to user-space memory allocation semantics, that the content inside a buffer returned by `malloc` is undefined. We do not expect this bypass in LLC to pollute cache for the following reasons: 1) This memory access will be referenced immediately after allocation, so we expect a very high hit rate. Also, this bypass is necessary as in the normal path where DRAM is accessed, LLC still need to decide which lines are to be evicted, 2) LLC is very large compare to 64 bytes, and all LLC misses are cold miss.

For security reason, zeroing of the allocated line (which many OS kernels will do) can also be piggybacked on block insertion by simply zeroing the content of the blocks after they are inserted.

4.2 Hide LLC latency

Illuminated by [2], we inserted a hardware object prefetcher (HOP) at L2 cache. When memory allocation was handled by Memento, the CPU will also invoke a custom `PREFETCH` request to the cache hierarchy. The hardware object prefetcher is responsible for detecting and dealing with such request.

Further optimizations can be made to let HOP decides whether or not this request should be forwarded to the next level of cache hierarchy, and if so, it decides when this request should be sent. Also, we can handle and optimize `free` requests to make use of free space in cache line and avoid useless write-back [6].

4.3 Put it together

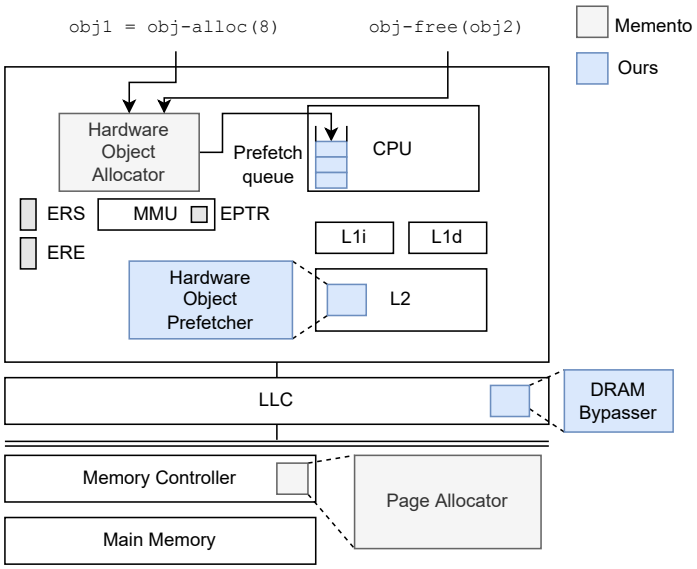


Figure 2: Design Overview

Our detailed design is shown in Fig. 2. When the hardware object allocator in Memento receives a memory allocation request, it will insert a `PREFETCH` request inside the CPU issue queue. When this `PREFETCH` request is issued, it will be passed down inside the cache hierarchy to L2 cache, unless it is hit in L1 cache.

Upon the hardware object prefetcher in L2 cache receiving `PREFETCH` request, it will immediately check its own tag array for a hit and issue a `PREFETCH` request to LLC if miss.

Each `PREFETCH` request that arrives at LLC will not be forwarded further to main memory. Instead, it will create cache lines for this request in its cache array directly when miss. Moreover, if the `PREFETCH` request arrives slower than the actually access request, the DRAM bypass policy will also instruct LLC to bypass DRAM access.

4.4 Future work

Further optimizations can be made to enhance the HOP to find a better timing and accuracy for each prefetching request. This can be done with the following thoughts:

- **A more selective HOP on each level of cache hierarchy.** Instead of blindly fetching every memory allocations, HOP decides whether or not this request should be forwarded to the next level of cache hierarchy, and if so, it decides when this request should be sent.
- **Using history to predicate prefetch decisions.** A new component called the hardware history profiller that profiles history of allocation accesses and fine-tuning the HOP based on those results.
- **Page-level prefetching in page allocator.** Perform a page-level prefetching to LLC when the initial tag walk request on a newly allocated page is received. The hardware page allocator can directly manipulate cache lines inside LLC rather than main memory.
- **Cooperate with software garbage collecotor.** The `free` operation is typically called explicitely inside the garbage collector. We can handle and optimize `free` requests to make use of free space in cache line and avoid useless write-back [6]

5 Evaluation

Simulation framework: We evaluate Memento and our cache management policy with full-system simulation using QEMU [1] integrated with the SST framework [5] and DRAMSim3[4]. We use Ubuntu 20.04 for the OS with the 5.18 kernel.

Table 1: Simulation Configuration.

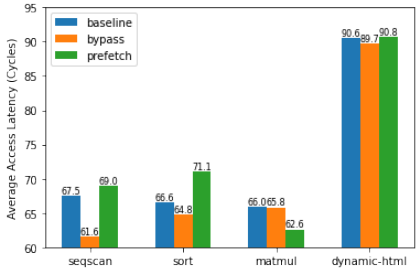
CPU	4-issue OOO, 3 GHz, 256-Entry ROB, 64-Entry LSQ
TLB	L1 64-Entry, 4-Way; L2 2048-Entry, 12-Way
L1d	32KB, 8-Way, 2 Cycle, LRU Replacement
L1i	32KB, 8-Way, 2 Cycle, LRU Replacement
HOT	3.4KB, Direct-Mapped, 2 Cycle
L2	256KB, 8-Way, 14 Cycle, LRU Replacement
LLC	2MB Slice, 16-Way, 40 Cycle, LRU Replacement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

System parameters: Our system parameters are given in Table 1. The two main hardware structures of Memento, the hardware object table (HOT) and the arena allocation cache (AAC) of the hardware page allocator are modeled as a 3.4KB direct-mapped cache and a 32-entry direct-mapped cache respectively.

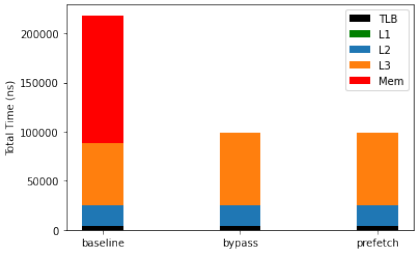
Benchmarks: To evaluate our cache management policy, we constructed our own benchmark suite consisted of 3 different microbenchmarks, which will respectively perform sequential scan on a array (Seqscan), sort an

array with randomly generated numbers (sort), and a full-scale serverless benchmark of Web applications called dynamic-html in SeBS.

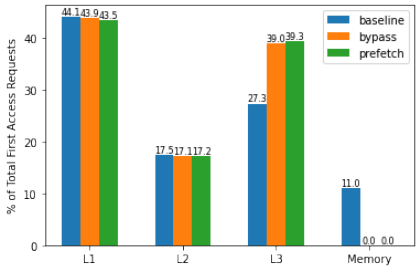
6 Results



(a) Memory allocation access average latency



(b) First allocation access cycle breakdown for `dynamic-html`



(c) First allocation access level hit breakdown for `dynamic-html`

Figure 3: Benchmark results

6.1 Memory allocation access latency

Fig. 3a shows the latency for every access to memory allocation for 4 benchmarks achieved by our design as the speedup over the baseline system (Memento). The speedup of DRAM bypass policy over the baseline is $0.3 - 9.6\%$.

The performance of hardware object prefetcher is almost always slightly worse than baseline system. We think the reason are mainly twofolds. First, CPU has to issue `PREFETCH` requests, which occupies cycle for normal events and overwhelming the cache hierarchy. Second, due to short allocation-access distance, the prefetch requests are issued too late and we have to pay for the latency of accessing MSHR.

Table 2: Avg. Access latency (Cycles)

Benchmark	Baseline	Bypass	Prefetch
Seqscan	67.53356282271945	61.631469979296064	69.03518454639531
Sort	66.59137931034483	64.75862068965517	71.10489993098689
Matmul	65.9701965757768	65.83195592286502	62.60138539042821
Dynamic-html	90.56061467965006	89.74099213648451	90.7662624343933

6.2 Memory allocation access Cycle breakdown

We further study the source of speedups, we present in Fig. 3b, a breakdown of cycles spent for first allocation accesses, and Fig. 3c, a breakdown of hits inside different levels of memory hierarchy. These results show that main memory access, though only take up a small fraction of total accesses, is the main root of latency.

The figure shows two main sources of gains that are be achieved: i) Bypassing DRAM access. 2) Higher L2 and LLC hit rate.

7 Discussion

7.1 Surprises

- Prefetch requests put pressure on CPU instructions as the number of instructions increases. We need to further fine-tune the performance of HOP to find the sweet point between hide LLC latency and avoid redundant prefetches.
- TLB misses are very expensive. Page table walks in TLB for first allocation accesses usually take a very long time.
- L1 cache latency are high even when hits. This is because some requests are still pending for response in the lower level of memory hierarchy, and the accesses that shares the same cache line with those requests will be considered hit in MSHR through stall for results.

7.2 Insights

- Memory allocations in serverless benchmarks will be accessed soon after their allocation.
- DRAM accesses for memory allocations at the **first** time are useless which by semantics of `malloc` will fetch undefined data and can be avoided.

7.3 Were proposal goals met?

We meet our 100% goal by implementing the allocation-aware cache management policy that combines a DRAM bypass policy and a hardware object prefetcher that will prefetch every memory allocation. We do not make claims for the performance of workloads and leave them to the 125% goal after we have a more optimized implementation of the whole design.

References

[1] Fabrice Bellard. {QEMU}, a fast and portable dynamic translator.

[2] Abhishek Bhattacharjee. Translation-triggered prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 63–76. Association for Computing Machinery.

[3] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: a serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, pages 64–78. Association for Computing Machinery.

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

[5] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The structural simulation toolkit. 38(4):37–42.

[6] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. Cooperative cache scrubbing. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, PACT '14, pages 15–26. Association for Computing Machinery.