Allocation-driven Cache Prefetching

15-618 Project Milestone

Pei Li
peili@andrew.cmu.edu

1 Introduction

1.1 Memento: Architectural Support for Ephemeral Memory Management

Serverless computing is an increasingly attractive cloud paradigm, which enables the decomposition of applications to smaller functions. As a result, it allows a ease of use and fine-grained pay-for-what-you-use billing for customers, and gives cloud providers the flexibility to provision resources. However, serverless computing poses new challenges to system design due to its short-lived execution model.

The ephemeral nature of serverless computing is ignored by current operating system and hardware layers. Specifically, users have to pay a high cost of memory management of the full critical path including userspace and OS compare the actual time that this memory chunk is utilized.

Memento is an ongoing project aiming at providing architectural support for alleviate such 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.

1.2 Prefetching Diminishes Memory Allocation Latency

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].

While hardware-based prefetching schemes have several important disadvantages in detecting the memory access patterns[6], we consider explicit memory allocation, such as 'malloc()' as such a strong prefetch predicate. The key observation of this project is that write requests can be avoided in the critical path, and when accessing a DRAM that is newly allocated, we will only write to that physical address before reading it. As a result, we do not have any expectation on the actual contents inside that address, but only care that the size of the buffer returned are large enough to hold the incoming data written. We can easily issue prefetch for a given size into cache if it is malloc'ed without actually touching DRAM to save memory bandwidth.

Current design of Memento is able to utilize preallocated DRAM in arena list to manage memory allocation efficiently, however, it will still hit memory wall doing an unnecessary DRAM access when 'malloc()'. By enabling prefetching, we plan to observe significant improvement of memory bandwidth.

2 Problem

While this project is illuminated based on a simple observation, the design of prefetcher varies based on these following problems that we need to take into consideration.

2.1 What to Prefetch

The first problem is deciding what is needed to be prefetched. In order to answer this question, we need to understand the tradeoffs between fetching every memory allocation. If everything is prefetched, we will have a high possibility of cache hit. However, this may not be as beneficial considering the diminishing returns on the following scenarios: 1) Cache pollution which causes cache eviction of potential cache hits due to conflict of prefetching. 2) The cache line that is prefetched is not accessed. To address this problem, we need to design a more selective policy.

2.2 When to Prefetch

The second problem is when to issue the prefetch request. The time to prefetch is extremely important as it will have direct effect on prefetcher performance considering the following scenarios: The first access is created long after allocation, and the prefetched data, arrives either too early so that is evicted, or too late that the access still need to wait.

2.3 How to Prefetch

The last problem is how to achieve prefetching. There are 2 different solutions: 1) Prefetch propagation from L1 cache to LLC. This solution will fast as each write request only goes to L1 cache on the critical path, but at the risk of pollute L1 cache and violate cache inclusion policy. 2) Prefetch promotion from LLC to L1 cache. This solution can be slower for going through all these different levels of cache at a latency that cannot be avoided. Thus the gap between the first access requests of this allocation may be smaller than cycles take for this promotion, and we still have to pay for cache miss due to this lateness.

3 Work Completed So far

3.1 Case study: Prefetching Opportunities

3.1.1 Metrics

In order to identify the opportunities on prefetching, we need to observe the behavior of different workloads when allocating memory and accessing memory allocations. We therefore look into 2 different metrics, which will evaluate the potential possible cycle to issue prefetching requests and potential benefits that can be gained by prefetching respectively.

The first metric, called **allocate-access distance**, will measure 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. 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.

The second metric, called **first-access latency**, will measure 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. We expect this request to be a write request by the semantics of the user space memory allocation. This metric describes potential benefits that we possibly achieve if enabling prefetching.

3.1.2 Results

We conducted experiments on a typical serverless benchmark of Web applications called dynamic-html in SeBS to study their memory allocation and access behavior. Our detailed analysis demonstrates that most allocate-access distance are short, and most of these access goes into DRAM.

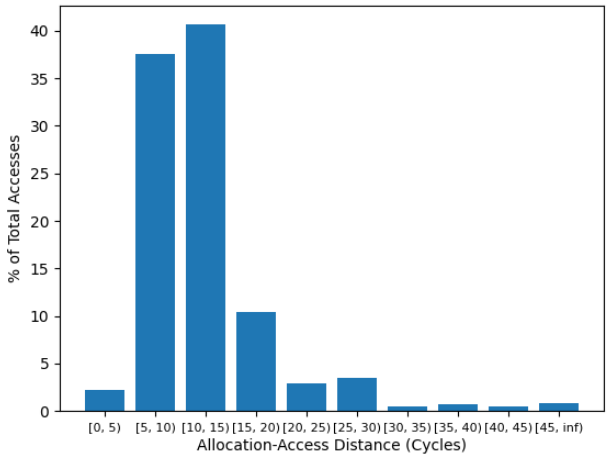

Figure 1: (a) Access gap measured as allocate-access distance

Fig. 1 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, which is too short for a prefetching request to be created, issued and handled. A quicker response

path of accesses must be raised.

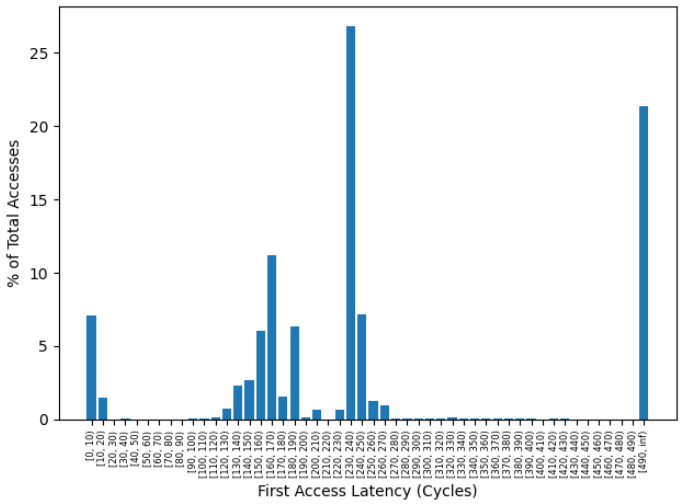

Figure 2: (b) Access latency measured as first access request to memory allocations

Fig. 2 shows the first-access latency. The result shows that the latency can be divided into 4 clusters. In the first small cluster whose latency is between 0 to 20 cycles, most of the accesses hit on L1 cache. 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. 2 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.

3.2 Design

3.2.1 Bypassing DRAM

To leverage the two observations in Section 3.1.2, we sketch the design of a hardware object prefetcher that collaborates with Memento as follows. 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. Therefore, in order to fully record each cache line for every memory allocation, 1 byte is needed. This can be further optimized as most memory allocations are smaller than 512 bytes, but for now we will just follows the simplest design.

For each request arrives at L3 cache, L3 will check whether this request is tagged as first access of memory allocations. If so, L3 cache 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).

Again, the correctness of this creation can be maintained because users do not care about the contents inside the buffer of memory allocation. We are only bypassing DRAM in L3 cache to maintain the cache inclusion policy. Future optimizations can be made to directly create a line in L1 cache, but it requires adjustment on complicated cache coherence protocols and thus not an ideal place to start.

We do not expect this bypass in L3 cache 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, L3 cache still need to decide which lines are to be evicted, 2) L3 cache is very large compare to 64 bytes, and all L3 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.

3.2.2 Implementation

This section starts with the code path of memory access in SST and then describes 2 different ways that can be used to hide the latency of DRAM. It is simply about detailed code implementation and the idea behind these two methods are identical. The first one is directly create cache lines in the hierarchy. The second method is adding a fake DRAM backend component with 0 latency.

1) CPU (ArielCore) sends memory requests by committing Read events and Write events. It will communicate with memory hierarchy (class MemHierarchyInterface) through function calls, and these memory events will be transformed to requests, then send

to cache thourgh links.

2) Cache coherence manager (CoherentMemController) handles memory requests between CPUs and maintain status and coherence of cache arrays. When receiving a memory request, and a cache miss occurs, the request will be inserted into the MSHR, and forwarded to remote cache. All processing on the address is blocked (excluding invalidations) until the response event from the lower level cache arrives. Different versions of coherence protocols are supported in SST, and we are using MESI inclusive protocol.

3) DRAM backend (MemBackend) is an abstract class of different memory backend such as DRAMSimMemory that we used. It will handle requests and send response to cache.

The first method requires us to i) Allocate a line in LLC; ii) Allocate and insert into MSHR when eviction required; iii) Send response to upper level of cache. This method will be easier to generalize for different levels of cache, but requires great understanding of existing MSHR logical and carefully handle of stalls and failures.

The second method requires us to add another DRAM component that response immediately when receiving tagged requests. The correctness of this method will be easier to maintain but we will have to pay for the latency on link.

Currently, we have implemented the first method but got performance that does not fits our design. We suspect that there are some bugs in our implementation and plan to spend the next couple of days building into observability and debugging it. If still we are not able to produce a reasonable result, we will try to implement the second method instead.

The next step after bypassing DRAM access would be bypassing inside the cache hierarchy including L2 cache and L3 cache. If the coding process of bypassing DRAM goes on well, we will continue working on this part.

4 Methodology

4.1 Resources Needed

- Bare-metal servers to run experiments and simulation on.
- Full-system simulator for evaluation, which will be free, open source software.
- Code repository of the initial design of Memento.

4.2 Evaluation

4.2.1 Simulation Infrustructure

We evaluate Memento and hardware object prefetcher with full-system simulation using QEMU [1] integrated with the SST framework [7] and DRAMSim3[5]. We use Ubuntu 20.04 for the OS with the 5.18 kernel.

4.2.2 Metrics

1) **End-to-end metrics**

The first set of metrics that we use is to evaluate end-to-end system performance. Thus, we plan to adapt a set of common metrics including latency, speedup, and memory bandwidth, etc.

2) **Quantifying prefetchers**

The second set of metric that we use is specially to evaluate the performance of prefetchers.

i) Accuracy. This is used to measure the percentage of hits of prefetch over total amount of prefetch issued.

ii) Lateness. This is used to measure the percentage of prefetched block that are still on its way over total amount of prefetch request that hit.

iii) Pollution. This is used to measure the cache miss caused by prefetched block eviction, which select demand block that will be used as victim.

iv) Coverage. This is used to measure the fraction of misses avoided by prefetch.

4.3 Benchmarks

We plan to run several serverless cloud function benchmark suite in the simulation environment. We will assemble our own workloads from popular memory intensive services such as FunctionBench [4], SeBS[3], and benchmarks for high-level programming language runtime such as Python's official benchmarking suite pyperformance. We plan to show plots of their results at the poster session.

5 Schedule

5.1 Goals

PLAN TO ACHIEVE: We plan to measure the theoretical benefits, as well as challenges of prefetching, and building a solution of bypassing DRAM in LLC for first memory allocation accesses. We will measure current Memento performance in benchmarks, as well as microbenchmarks, as baseline, and observe the prefetching opportunities.

HOPE TO ACHIEVE: We hope to look into cache inclusion policies to find solution for bypassing L3 and L2 cache, and implement a more selective prefetching policy. We will then use our observability tools to compare this new scheme to baseline under certain workloads, and hopefully report some positive results.

5.2 Week-by-week Schedule

Time	Objective
Week 1	Debugging and evaluating bypass DRAM solution, and gather statistics for profiling cache and DRAM behaviors
Week 2	Implement bypass L2 and L3 solution
Week 3	Experiments and work on final report

Table 1: The week-by-week schedule.

Our week-by-week schedule is shown in 1. This is a reference of time about what work needs to be done and will be dynamically adjusted according to current progress.

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] Jeongchul Kim and Kyungyong Lee. FunctionBench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. ISSN: 2159-6190.

[5] 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.

[6] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. 27(9):62–73.

[7] 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.