Prefetching in Memento

15-618 Project Proposal

Pei Li
peili@andrew.cmu.edu

1 Introduction

1.1 Ephemeral Memory Management Overhead and Memento

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 Methodology

3.1 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.

3.2 Evaluation

3.2.1 Simulation Infrustructure

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

3.3 Metrics

3.3.1 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.

3.3.2 Quantifying prefetchers

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

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

2) 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.

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

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

3.4 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.

4 Schedule

4.1 Goals

Our 75% goal is to measure the theoretical benefits, as well as challenges of prefetching, and start building a simple prefetcher that is able to prefetch every memory allocations. We will measure current Memento performance in benchmarks, as well as microbenchmarks, as baseline, and observe the prefetching opportunities.

Our 100% goal is an implementation of the 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.

Our 125% goal is observing a meaningful reduction of memory bandwidth usage and latency, by continue to optimize our prefetching approach. By meaningful, we attempt to achieve performance improvement by at least 10%.

4.2 Week-by-week Schedule

Time	**Objective**
Week 1	Understand prefetch opportunities
Week 2	Implement a fetch-everything prefetcher
Week 3	Implement a selective prefetcher
Week 4	Optimization on previous prefetcher
Week 5	Gather statistics and write 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.