Hardware-Software Co-Designed Cache Bypass Mechanism on X86 Machine

EECS 583 Fall 2020 Final Project Report
Group 11: Yichen Yang, Wentao Zhang, Wenqi Zhu, Zhiyi Pan
University of Michigan, Ann Arbor, MI, USA
{yangych, zwtao, zwq, zhiyipan}@umich.edu

***Abstract*—With the growing size of real-world datasets running on modern x86 machines, real-world application are suffering a lot from the memory bottleneck. Multi level cache hierarchy is designed to store the most recent used data near the core for future accesses with lower latency comparing to the DRAM. Thus, to acquire a piece of data from the memory hierarchy, normal CPU usually follows the path of L1 cache, L2 cache and DRAM. However, the least recent use (LRU) eviction policy may not work well for some instructions. Some data are not reused again after being brought into the cache. In such scenario, the cache is polluted with useless data and may evict data that are needed in the near future.**

Previous works have proposed doing ucache bypass on a EPIC (Explicitly parallel instruction computing) machine. They use compiler to analyze the application characteristic and schedule the instructions with the awareness of the ucache bypass mechanism. The compiler selects some load instructions that will pollute the ucache to do the bypass, thus getting better utilization of the ucache and better overall performance.

Inspired by ucache bypass on EPIC machine, we explore the opportunity of doing general cache bypass on modern x86 out-of-order machine. We developed multiple compiler pass based on different bypass instruction selection algorithms to analyze the cache profiling stats and mark instructions to bypass the cache. We also evaluated and compared different selection algorithms on some graph workloads. Our proposed cache bypass mechanism on X86 machine can reduce the L1D cache miss rate under some scenarios.

I. Introduction

The development of semiconductor technology makes it possible to build high performance computer systems with multiple processors. The execution speed of operations grows faster, but fetching data from the main memory is slow. Use of cache can ameliorate this problem since cache reference time is much less than main memory reference time. However, fetching infrequently used data into the cache on a cache-miss sometimes reduces system performance eve if the cache size is infinitely large [4]. Figure 1 shows percentage of total execution time cost on different parts of the workload. It indicates some graph algorithms are bottle-necked by the memory. Fortunately, cache bypassing, where the memory requests can selectively bypass the cache, is one of effective solutions.

Wu *et al.* [10] have explored selective micro-cache(*ucache*) bypassing for high performance EPIC processors. Their work focuses on a small and fast data cache at the beginning of the memory hierarchy and EPIC microprocessors.

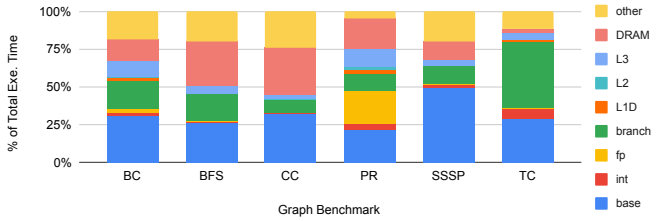

Fig. 1: Percentage of total execution time spent on different part of the workload. Dataset is Slashdot0922 [8]. Most graph workloads are bottle-necked by the memory.

EPIC microprocessors execute instructions in an order governed by the availability of input data and execution units [5]. However, many modern processors like Intel x86 use out-of-order execution.

In this paper, we develop a L1/L2 cache bypassing system on X86 machines. We first introduce background in Section II, then illustrate different parts of our design in Section III and the integrated methodology in Section IV. Evaluation and results are discussed in Section V and VI.

II. Background

Cache profiling technology has been explored by many researchers in past years [1, 3, 6, 9]. Our approach utilizes cache profiling data to determine which part need to be bypassed as shown in section III.

The profiling time is large for complex applications with a large input data. The running time and compilation time will be significantly decreased if we feed those applications with small input data. Furthermore, there are many overlap bypassing PCs (program counters) between running small and large input data on the same application. Hence, we can do cache profiling on a application with the small input data to generate a bypass list, then use this bypass list on the application with larger input data to achieve better performance as shown in section IV. This will greatly reduce the compilation overhead while maintaining a decent performance.

III. Design

The hardware-software co-designed cache bypass mechanism consists of three parts: 1) cache profiling, 2) compiler pass and 3) hardware modification. This section will introduce these three parts in detail. Note that we only bypass the load instructions, and all the store instructions perform as normal.

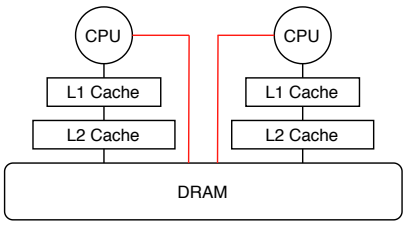

Fig. 2: Memory hierarchy after modification. Modification is marked in red.

A. Cache Profiling

Firstly, we need to run an non-optimized program on the machine to collect the cache performance stats. To support a variety of compiler algorithm, the profiling stats includes hit/miss for each instruction, hit/miss rate for each PC and hit/miss rate for each target address, and pass these stats to the compiler.

B. Compiler Pass

Secondly, we use the compiler to analyze the cache profiling stats and mark these selected load instructions as cache bypass load instructions. Specifically, we proposed several algorithms.

- **Load Instruction Miss Rate (Naive):**
  The compiler will select the PCs that has a overall miss rate higher than a certain threshold (swept in later experiment) to bypass the cache.
- **Top % Miss Rate:**
  The compiler will select the top certain percent (swept in later experiment) of PCs to bypass the cache.
- **Reuse Probability:**
  Following [10], the load miss no reuse probability ($mnrp$) is calculated as Equation 1. The compiler will select the PCs that has a $mnrp$ higher than a certain threshold (swept in later experiment) to bypass the cache.

$$mnrp = \frac{number\_misses\_no\_reuse}{number\_loads\_executed} \quad (1)$$

C. Hardware Modification

After the compiler marked load instructions that should bypass the cache, the hardware should accommodate this ISA level modification. In addition to the normal memory hierarchy, another link from CPU cores to the DRAM should be added (marked in red in Figure 2). The core will then determine which path to load the data based on the compiler marked instructions. If the instruction is marked as bypass, it will follow the red line and directly access the memory. Otherwise, it will follow the normal path and access L1 cache first.

D. Integration

Put all these together. Our design first collects cache profiling stats and adopts a compiler pass to mark the instructions to bypass the cache in ISA level. The modified hardware will based on the mark passed by the compiler to directly access the DRAM or access the L1 cache as normal.

IV. Methodology

We use Dynamorio [1] with its built-in cache simulator as our main tool. For the cache profiling, we hacked Dynamorio to dump a offline memory accessing trace including load&store instruction PC, hit/miss for this instruction and target data address. For the compiler pass, we developed a compiler pass in Python to analyze the offline trace and generate a **Bypass-List** storing all the PCs that should bypass the cache. In addition to generating only one **Bypass-List**, we also take the intersection of several datasets' **Bypass-List** to form a common **Bypass-List** in some experiments below. For the cache simulation, we modified Dynamorio to read the **Bypass-List** created by our compiler pass and perform cache bypass. Note that all the experiments are conducted after turning off Address Space Layout Randomization (ASLR) on a Linux machine to keep the instruction address consistent in several runs.

A. Benchmarks

We used graph workloads from gapbs [2], including Breadth-First Search (BFS), Single-Source Shortest Paths (SSSP), PageRank (PR), Connected Components (CC), Betweenness Centrality (BC) and Triangle Counting (TC) because graph workloads is mostly bottle-necked by the memory (Figure 1). We mainly used BFS in the following evaluation part.

B. Dataset

We used SNAP dataset [7] in the graph algorithms. Table I shows a list of them.

Dataset	#Node	#Edges	Category
email-Eu-core (email)	1K	25.5K	Train/Test
p2p-Gnutella08 (p2p)	6.3K	20.7K	Train
wiki-Vote (vote)	7.1K	104K	Train
ca-GR-QC (qc)	5.2K	12.5K	Test
Slashdot-09 (slash)	82K	948K	Test
Pokec (pk)	1.6M	30M	Test

TABLE I: Graph datasets used in experiment. *Train* means it is used to generate **Bypass-List**, and *Test* means it is used to test the bypass performance.

C. Simulated Cache Hierarchy

We used default cache setting from Dynamorio cache simulator. Specifications are listed in Table II.

L1 Size	32KB
L1 Assoc	8
L2 Size	8MB
L2 Assoc	16
Cache Line Size	64

TABLE II: Cache simulator specification.

V. Evaluation

In this section, we will use BFS to evaluate our cache bypassing algorithm on the benchmarks mentioned in Section IV. We evaluate our algorithms based on the L1D cache hit and miss number. Then the best algorithm is used to test its performance on different data size, cache size and cache line size. Finally, we will estimate its performance gain.

A. Bypassing List Similarity

Figure 3 shows the similarity between the bypass lists generated by different data sets, `email`, `p2p` and `vote` in our case. The similarity is defined as the intersection of the PCs in bypass lists divided by the union of them. The average common rate across different bypass threshold value can reach 73%. This shows that for a certain program, the majority of the bypass list will be the same even if the data set it uses is different. This also validates the assumption in Section II that using small data set generated bypass list to optimize the program with larger data sets.

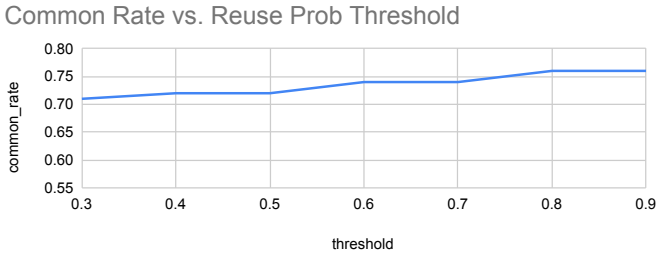

Fig. 3: Bypass lists similarity of BFS running `email`, `p2p` and `vote` dataset with different reuse-prob. threshold.

B. Bypassing Algorithm Evaluation

1) Naive and Top Miss Rate: Figure 4 and Figure 5 show the performance of the Naive algorithm and the Top Miss Rate algorithm. Both of them didn't achieve the goal to reduce the number of L1D cache miss while keep the L1D hit opportunity. The Naive algorithm reduces the L1D cache miss, but it also cuts too many L1D cache hits. The Top Miss Rate Algorithm doesn't filter out the miss effectively. The reason behind these two algorithm's bad performance can be that they make the bypassing decision only based on one PC's miss probability. In reality, the cache line loaded by this specific instruction can be reused by other instructions, which forms a producer and consumer pattern. So simply bypassing the "producer" instruction may incur "consumers's" miss thus increasing the overall miss rate.

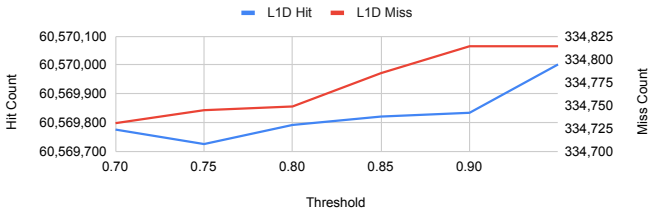

Fig. 4: Number of L1D Cache hits and misses (Naive Algorithm)

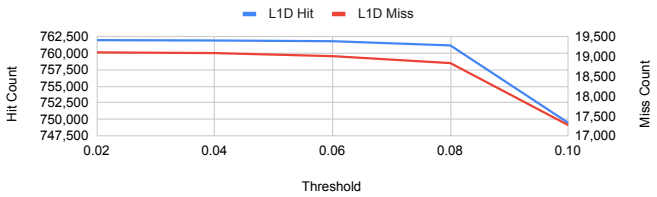

Fig. 5: Number of L1D Cache hits and misses (Top Miss Rate Algorithm)

2) Reuse Probability: Figure 6 shows the L1D cache hit and miss number with different bypassing thresholds. The result is collected from the `email` data set. When the bypassing threshold is 0.5, the hit number almost reaches the maximum value while the miss number remains low. Further increasing the threshold value will not gives us higher hit counts, but will solely increase the miss numbers. The threshold of 0.5 seems to be a sweet point for the reuse probability algorithm. Thus, in the following discussion, we will use 0.5 as our threshold value for the Reuse Probability Algorithm.

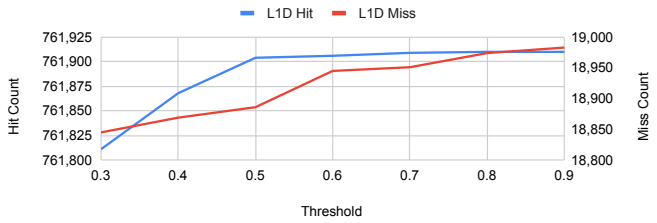

Fig. 6: Number of L1D Cache hits and misses (Reuse Probability Algorithm)

C. Effects of Bypassing on Cache Performance

Section V-B shows that the Reuse Probability algorithm with threshold value 0.5 achieves the best performance on the email data set. So in the following discussion, we will use this setting and evaluate its performance with regard to the data set size, cache size and the cache line size.

1) Data Size: The Reuse Probability algorithm will have better performance on the data sets whose edges and nodes number is similar to the training data set size. For the large data set, this algorithm won't achieve satisfying result. The benefit provides by bypassing the loads will be diminished by the later store operation. Since the number of stores increases with the data size, the impact of store will become dominant in cache hits and misses.

2) Cache Size: Figure 7 shows the L1D cache hit and miss counts with and without bypassing with regard to the different cache size. The reduction in number of hits does not scale with the cache size. However, from Figure 8, we can observe the the bypassing algorithm is more effective on small cache size. This is because the smaller cache will have more conflict misses, thus providing the bypassing algorithm more opportunities.

3) Cache Line Size: Figure 9 shows the relationship between the reduction of L1D cache miss and the cache line size. With smaller cache line size, the miss reduction counts

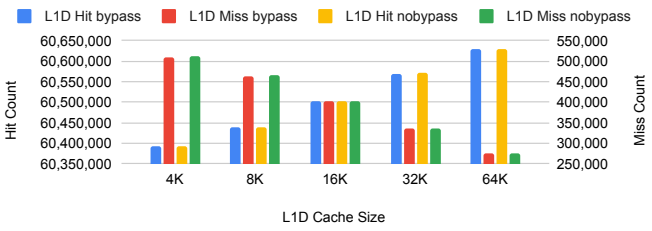

Fig. 7: Number of L1D Cache hits and misses

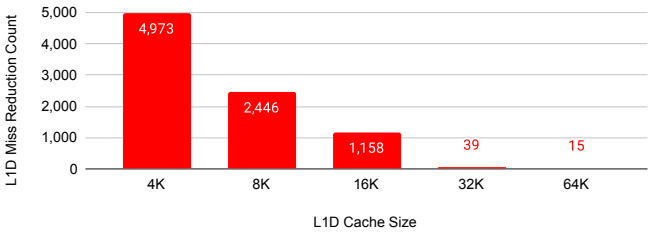

Fig. 8: L1D cache miss reduction for different L1D cache size.

significantly increases. This is because that the smaller cache line size provides smaller spatial locality. The reuse probability of a specific cache line will decrease. Thus, bypassing that cache line will potentially cause less new cache miss. This leads to the greater reduction in L1D cache misses.

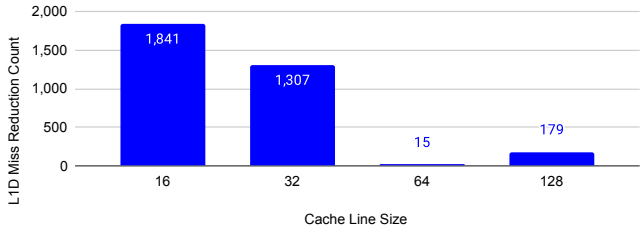

Fig. 9: L1D cache miss reduction for different cache line size.

D. Performance Gain

Finally, we will estimate the total cache access time based on the number of cache hits and misses to show the performance gain provided by the Reuse Probability bypassing algorithm. We choose Intel i7-4770 as our estimation CPU, since it has the same size L1 cache as our simulator. The specification used in calculation is listed in Table III

L1 Size	32KB
L1 Assoc	8
L2 Size	8MB
L2 Assoc	16
Cache Line Size	64
L1 Hit Latency (cycles)	5
L2 Hit Latency (cycles)	66
L2 Miss Latency (cycles)	128
Bypass Latency (cycles)	62

TABLE III: Cache specification and access latency for overall performance estimation.

The total cache access time without bypassing will be calculated as:

$$access\_time\_nobypass = hit\_time \times hit\_counts + miss\_time \times miss\_counts \quad (2)$$

The total cache access time with bypassing will be calculated as:

$$access\_time\_bypass = hit\_time \times hit\_counts + miss\_time \times miss\_counts + bypass\_time \times bypass\_counts \quad (3)$$

The L1 miss time will be calculated as:

$$L1\_miss\_time = L1\_hit\_time + LL\_hit\_time + LL\_miss\_time \times LL\_miss\_rate \quad (4)$$

Bypass count will be calculated as:

$$bypass\_count = L1\_hit\_nobypass + L1\_miss\_nobypass - (L1\_hit\_bypass + L1\_miss\_bypass) \quad (5)$$

Figure 10 shows the estimated overall performance gain on different test graph data set in terms of of execution cycles improved. We see that the bypass algorithm works better on some smaller data sets (`email` and `qc`), while not achieving good performance gain on some large data sets (`slash` and `pokec`). One reason is that the bypass opportunity will decrease as the data set size increases. In the large data set, a cache line will be less likely to be reused by the other PCs before it is evicted and this leads to the lower bypass opportunity and less performance gain. Also, bypass the L1 cache will lead to higher LL cache miss in the large data set. This causes the performance lost.

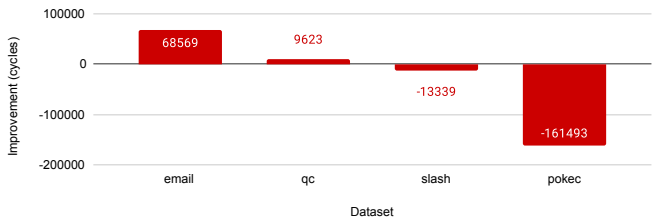

Fig. 10: Estimated overall performance gain on different test graph data set.

VI. Discussion and Future Works

In this section we discuss the limitations in our current method, along with possible workarounds and future works.

Not perform well on large data set. As mentioned in Section V, our method doesn't perform well on large data set, and it even causes performance downgrade on pokec data set.The possible reasons is that data-access pattern of applications on large dataset is different from that on small dataset. We can investigate data-access pattern in larger dataset and adjust our algorithms.

A single iteration might not suffice for reaching global optimal. All the experiments we conduct are single-iteration,

namely we only run our tool chain once to get the results. It might not be enough for an optimized result: single-iteration bypass generates new miss. For future works, it's worth trying running the tool chain for multiple iterations. We expect multiple iterations will finally converge to a global optimal, where no more new miss will be generated after bypassing.

Current method bypass the whole cache. In the current version of our bypassing tool chain, once the load is annotated as a bypassed load, it will bypass the whole cache. We think it might cause performance downgrade under some circumstances. If a load instruction is never reused in L1D but is frequently reused in LLC, it's not smart to bypass the whole cache including LLC. To further improve the bypassing tool chain, our future work includes bypassing selected layer of cache based on both L1D and LLC profiling statistics.

VII. Conclusion

In this paper we discussed if x86 machine has cache bypass opportunity like Epic machine. We build an end-to-end HW-SW co-designed tool chain to show that cache bypass brings performance gains to X86 machine under some circumstances. More specifically, our tool chain with reuse probability algorithm works best among all three proposed algorithms and it performs better on small cache size and small cache line size. We demonstrate that X86 do have cache bypass opportunities under some circumstances.

References

[1] "Dynamic instrumentation tool platform," https://dynamorio.org/.

[2] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[3] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 169–180. [Online]. Available: https://doi.org/10.1145/1064212.1064232

[4] C. Chi and H. Dietz, "Improving cache performance by selective cache bypass," in *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume 1: Architecture Track*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, jan 1989, pp. 277,278,279,280,281,282,283,284,285. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HICSS.1989.47168

[5] W. W. S. Chu, Dimond RG, S. Perrott, S. P. Seng, and W. Luk, "Customisable epic processor: architecture and tools," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 3, 2004, pp. 236–241 Vol.3.

[6] A. R. Lebeck and D. A. Wood, "Cache profiling and the spec benchmarks: a case study," *Computer*, vol. 27, no. 10, pp. 15–26, 1994.

[7] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[8] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[9] E. van der Deijl, G. Kanbier, O. Temam, and E. D. Granston, "A cache visualization tool," *Computer*, vol. 30, no. 7, pp. 71–78, 1997.

[10] Youfeng Wu, R. Rakvic, Li-Ling Chen, Chyi-Chang Miao, G. Chrysos, and J. Fang, "Compiler managed micro-cache bypassing for high performance epic processors," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, 2002, pp. 134–145.