Victim cache implementation and performance analysis using gem5 - progress report

Madhav Rathi
mrathi3@wisc.edu

Jason Zhou
zzhou292@wisc.edu

I. Topic Description

Memory accesses is the key issue in increasing ILP and performance of processors. We can routinely spend 10s to 100s of cycles in fetching data from memory when there is a miss in the L1Cache. Jouppi in [1] showed that conventionally implementing caches will not help with increasing demand for performance, so Jouppi proposed victim cache as an intermediate cache between L1D and L2 cache, which stores evicted lines from L1D cache and thereby improves performance. In this project, we focus on victim cache as an optimization to reduce memory access latency. We propose to implement victim cache at multiple levels of cache hierarchy to analyze it's performance impact. Victim cache will be implemented for both simple in-order TimingSimpleCPU and MinorCPU, before moving on to complex out-of-order O3CPU. Validation will be done on in-order CPUs through simple microbenchmarks. We compare the performance impact of adding victim cache at different levels of cache hierarchy, and analyze it's performance compared to CPUs without victim cache. Victim cache inclusion in simulation will be configurable, similar to cache inclusion in gem5.

II. Topic Relevance

Jouppi in [1] showed that victim cache can eliminate conflict misses in L1 D-cache as much as 40-100%. Victim cache has minimal access latency due to being a small cache, thereby improving performance compared to lower level cache accesses. Gem5 being the state-of-art architecture simulation tool does not implement a victim cache, thereby missing key micro-architectural performance modeling information. Adding a victim cache to gem5 will help it mimic the micro-architecture further. We also identify that gem5 can be improved in terms of making it easier for the user to add components within a system without having to do large scale code changes. Even in recent years, victim caches (or similar concept) are actively being researched on and being used to improve performance and reduce power consumption. For example, victim caches are also being used in GPUs to reduce L1 cache miss penalty (which GPU has many misses of) [11], as well as used to increase register file space in the GPU (which GPU needs a lot of) [12]. We also see same structures being used for multiple purposes, such as a trace buffer also being used as a victim cache [9]. Victim caches are also begin used in mitigating side-channel attacks such as meltdown and spectre [10].

III. Evaluation Methods

We list the performance metrics (stats) compared and the benchmarks to be run for analysis.
Performance metrics compared:

1) Percentage of misses due to conflicts: Helps to identify the number of misses seen in the program without victim caches implemented.
2) Percentage of conflict misses removed by victim cache: When compared with the above stat gives the exact number of conflict misses removed by victim cache and its effectiveness.
3) Percentage of all misses removed due to victim cache: Capacity misses can also be reduced by using victim cache and this metric would allow us to identify the relative difference in the type of misses removed by victim cache.
4) Number of L2 accesses with and without victim cache at L1: Allows us to gauge the latency penalty we would have faced if the access was allowed to go to L2 cache.
5) Number of DRAM accesses with and without victim cache at L2.: Allows us to gauge the latency penalty we would have faced if the access was allowed to go to DRAM.

Benchmarks to be run for performance analysis:

1) MC and MCS microbenchmarks which exhibits many conflict misses [5]. Conflict misses arise when the program accesses many addresses which fall in the same cache set, thereby evicting useful lines from the cache set. Since victim cache effectively increases the associativity of a set by storing these evicted lines, the processor can benefit from victim cache lookup before accessing a lower level cache (which has high latency).
2) Spec2006 {464.h264ref-2, 464.h264ref-3, 456.hmmer-1} [6]. These benchmarks show high memory sensitivity, that is result in many cache misses due to large working set size accessed [8]. We want to profile the impact of victim caches in reducing lower level cache accesses.
3) CCa microbenchmark from [5]. Even though CCa is not memory access intensive, we would like to understand how much of diminishing return victim cache can have for these type of programs.

IV. Project Plan

Based on the current progress with the project, we believe that the original proposed deadlines are too aggressive. After adjustment considering the difficulties we are facing, the following tasks and target deadlines were updated:

1) Implementation of victim cache in TimingSimpleCPU and MinorCPU: The implementation will extend the BaseCache class and create a fully associative victim cache. First implemented in TimingSimpleCPU and MinorCPU. Victim cache implemented in data path, between L1 D-Cache & L2, L2 & L3.
November 22, 2022
2) Validation of victim cache in TimingSimpleCPU and MinorCPU: Validation will be performed on TimingSimpleCPU and MinorCPU with victim cache implemented between L1 D-Cache & L2. Once, the correctness of victim cache is established, we can move on to performance analysis. If any incorrectness has been discovered, the previous step will need to be modified and replayed.
November 25, 2022
3) Performance analysis on TimingSimpleCPU and MinorCPU: A benchmark comparison will be performed with victim cache implemented between (1) L1 D-Cache & L2; (2) L2 & DRAM; (3) Both L1 D-Cache & L2, L2 & DRAM. Performance analysis will be performed using several different benchmarks mimicking the performance analysis conducted in [1]
November 30, 2022
4) Implementation of victim cache in O3CPU: Once analysis completed for TimingSimpleCPU and MinorCPU, we implement victim cache in O3 CPU. This brings additional complexity since call to the cache is from LSQ, as well as victim cache must be non-blocking. Here, victim cache is implemented only between L1 D-Cache & L2.
December 5, 2022
5) Validation of victim cache in O3CPU: Validation will be performed on O3CPU with victim cache implemented between L1 D-Cache & L2. Once, the correctness of victim cache is established, we can move on to performance analysis. If any incorrectness has been discovered, the previous step will need to be modified and replayed.
December 10, 2022
6) Performance analysis on O3CPU: A benchmark comparison will be performed with victim cache implemented between L1 D-Cache & L2. Performance analysis will be performed using several different benchmarks mimicking the performance analysis conducted in [1].
December 17, 2022

V. Implementation details

Table 1 lists tentative microarchitecture details of victim cache architecture. Table 2 lists tentative microarchitecture details of base L1 D-Cache and L2 cache. We simulate our design only on single core architectures, which eliminates coherence checks of the victim cache.

TABLE I
VICTIM CACHE MICROARCHITECTURE[a]

Victim Cache	Microarchitecture Details			
	Size	*Associativity*	*Replacement*	*Line Size*
L1	4KB	Fully-associative	LRU	64B
L2	16KB	Fully-associative	LRU	64B

[a]Subject to change.

TABLE II
BASE CACHE MICROARCHITECTURE[b]

Base Cache	Microarchitecture Details			
	Size	*Associativity*	*Replacement*	*Line Size*
L1	64KB	2-way set associative	LRU	64B
L2	512KB	4-way set associative	LRU	64B

[b]Subject to change.

VI. Progress Report

As of November 18, 2022, we have implemented a victim cache between L1D cache and L2 cache in TimingSimpleCPU. The correctness and validation of the victim cache is in progress and scheduled to be completed by November 20, 2022. In summary, we attempted to use a hack to make the gem5 L2 cache cache behave as our victim cache, while also pushing the current L2 cache to the L3 cache level. By modifying access latency, size and associativity, theoretically this implementation can simulate a two level cache system with a victim cache servicing the 1st level cache. The block diagram of the cache hierarchy along with the buses is shown in 1. To emulate the behaviour of victim cache, we modify 'clusitivity' and 'writeback clean' properties of caches. First we explain 'clusitivity' and 'writeback clean' properties and their relation to the victim cache design.

a) Clusitivity: Clusitivity refers to the data exclusiveness between the caches in the cache hierarchy. Strongly inclusive property means that whenever data is loaded from the memory, this data is stored both in the lower level as well as upper level cache. Strongly exclusive property means that whenever

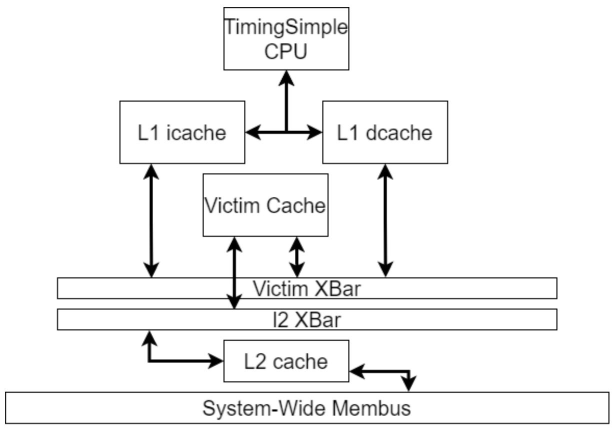

Fig. 1. The system configuration diagram of the currently implemented version of Victim Cache

data is loaded from the memory, only the highest level of cache caches the loaded line. In our implementation, a strongly inclusive property at L2 cache would mean that all data loaded from memory will be stored in L2, Victim and L1D cache. We do not want this behaviour since this contradicts victim cache architecture. So we choose strongly exclusive property at L2 cache. We chose the same exclusive property at victim cache. L1D cache remains strongly inclusive.

b) Writeback clean: Writeback clean refers to 'writing' back the lines in the lower level caches when a clean line is evicted from a higher level cache. In our implementation, if a writeback clean line is evicted from L1D cache, this needs to be written in to the victim cache (as it satisfies vicim cache definition). However, a decision needs to be made whether a writeback clean line from victim cache can be written back in to the L2 cache. We decide to not implement this behavior in victim cache, which means that lines evicted from the victim cache are not written back to L2 cache. L2 cache remains to have the writeback clean property.

By modifying the victim cache access latency, size and associativity, we make sure that implementation mimics victim cache behaviour. Victim cache has a 1 cycle load-to-use latency. The architecture of victim cache is given in Table 1. This methodology of implementing victim cache will scale to other CPU types as well since only the memory hierarchy is changing. Only changing the configuration script for different CPUs to run simulations should in theory suffice.

Validation of the victim cache is in progress, wherein we put a few debug statements to check that the lines evicted from L1 cache are filled in the victim cache. We are also working to validate whether the victim cache that we have implemented can actually reduce cache misses. A github repository has been created for this project and can be found at https://github.com/madrat01/ECE_752_Project.

In the current implementation, since we place the victim cache at L2 level, the victim cache will also be present for L1I cache. However, this is a rather unconventional configuration since usually victim cache is private for each cache. A solution here would be to make L1I cache as no writeback clean, which means that victim cache is not populated with L1I cache evicted lines. We are yet to identify the performance impact of this.

An option we considered was to embed victim cache object in the L1D cache. This required multiple changes to the L1 cache code in terms of victim cache call, moving data from L1 cache to victim cache and not doing a writeback clean evict from the victim cache. This route requires extra efforts since we would like to implement the victim cache at the l2 level as well. This would also require us to change code for multiple CPU types. For example, in O3CPU, L1D cache is called from LSQ and the way the cache responds is different than in TimingSimpleCPU.

Another approach we considered was to add an additional victim cache port to the CPU parallel to dcache and icache ports. This turned out to be complex to implement, since the information needs to be transferred between dcache and victim port, and the current cache return information doesn't return the full line evicted/accessed. The new port now also needs to deal with coherence snooping.

Regarding performance analysis, we have explored which programs can benefit from victim cache the most. Since victim cache stores evicted data blocks from upper level cache, it's reasonable to assume any program which shows recent reuse pattern can benefit from victim cache, while programs with sparse data access pattern might not. These reuse can result in conflict misses if the reuse trashes the same set. As identified in our proposal, we plan to go ahead with microbenchmarks and Spec2006 benchmarks mentioned in Section III.

VII. Related Work and Additional Work

The topic of the project is mainly inspired by the Jouppi paper [1] we have discussed in class. We started from the Jouppi paper to explore more research related to improving victim cache performance. We have found many victim-cache related work regarding improving victim cache hit rate by adopting more effective victim selection policies. However, there are two problems which we found in the current research - how victim cache scales with the size (most research only focus on small victim cache) and what type of data access pattern can benefit the most by using victim cache have not been addressed in detail. From there, we have diverted our research on related gem5 documents along with modern and comprehensive bench mark programs to provide guidance for our project.

Jouppi in [1] proposed the concept of victim cache and provided benchmark results showing that victim cache can significantly improve the cache performance. This paper tries to emulate this victim cache design in gem5. We present some basic conceptual implementation details which serve as a baseline for implementation and validation on TimingSimpleCPU and MinorCPU.

References [2], [3], and [3] optimize victim cache performance. Both [2] and [3] propose a modified selection and prediction policy identifying the content filled in the victim cache. In [2], Kim et al. propose an adaptive block management scheme for victim cache, wherein blocks filled in the victim cache also consider L1 cache history information. In [3], victim cache is also used as a miss cache, wherein an incoming block fill to L1 cache is selectively placed in L1 or victim cache based on the history of that block. We do not consider prediction or history information while filling the victim cache, as similar to [1], every eviction from the cache is filled in to the victim cache. [3] implements and benchmarks a combination of prefetching and victim caching, although this might not be directly related to our topic, it sheds lights on some benchmark results by implementing victim cache only. Our scope for this paper is only on victim caching, and we do not consider prefetch impact.

[5] provides many microbenchmark algorithms in five categories - Control, Data Parallel, Execution, Memory, and Store Intense - which we can take advantage of to understand how the implemented victim cache performs for various

applications. For example, MIM exhibits stream accesses, MC which generates a lot of conflict misses, while CCa is not memory heavy. [6] provides CPU benchmark standards released on August 24, 2006 by Standard Performance Evaluation Corporation (SPEC), which we can utilize as a reference during our performance analysis.

[7] will serve as a general guide as we are navigating through the gem5 simulator.

Although we do not plan to implement additional victim selection policies as some related work mentioned, we have found out that there is not a lot of data which evaluates the performance of victim cache on L2 level, or with a slightly larger capacity. Since increasing size of the victim cache provides a larger victim buffer window for the program, we should theoretically expect a higher hit rate and therefore a better overall performance. However, the problem lies on scaling, are we expecting a larger size victim cache to be more efficient compared to only a small 2-entry victim cache, or there is a diminishing gain. Also, benchmark algorithms used in some related work are either outdated or the details of the bench marking algorithms not fully unfolded. With our proposed gem5 implementation, we can flexibly adjust the capacity and the configuration of the simulated system and therefore gain more insights on how the size of the victim cache is affecting its refill capability to the L1 cache and whether L2 cache can benefit from victim cache by an increased size compared to L1 victim cache. All these insights can be obtained by flexibly adjusting our simulation values. Also, since we are using a comprehensive bench marking package as proposed in [6] and [5], we would stand out by evaluating what types of data pattern can benefit from victim cache more effectively, and whether for some data patterns, victim cache might even negatively impact the performance.

Besides, within the scope of the current discussion and our proposed work, we have been largely focused on single CPU system, a possible expansion on the topics can be how to adapt victim cache into a multi-core system. Abu Asaduzzaman, Mark P. Allen, and Tania Jareen in [13] proposed a locking-free l1 common smart victim cache system to enable effective victim selection based on history data. Such topics related to a common victim shared through all CPUs in a multi-core system can be an expansion to the current topic. However, the complex implementation of such a victim cache will not be feasible within the limited time for project. Therefore, our research project will still focused on the testing the victim cache implementation in in-order single core system.

Most of our references are gathered from google scholar, references from major papers on caches and victim caches for CPUs and GPUs, as well as our course knowledge of combining multiple schemes for better performance (e.g., prefetch algorithms which are aware of victim cache). We plan to follow similar approach to find additional research in this area.

References

[1] Norman P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” ACM SIGARCH Computer Architecture News Volume 18, Issue 2SI, June 1990.

[2] Cheol Hong Kim, Jong Wook Kwak, Seong Tae Jhang, Chu Shik Jhon, “Adaptive Block Management for Victim Cache by Exploiting L1 Cache History Information” International Conference on Embedded and Ubiquitous Computing, EUC 2004: Embedded and Ubiquitous Computing pp 1-11, 2004.

[3] Walter Schilling, Mansoor Alam , “The Impact of Prefetching and Victim Caching on Computer Systems Performance.” Conference: Proceedings of the ISCA 15th International Conference Computers and Their Applications, March 29-31, 2000, New Orleans, Louisiana, USA, January 2000.

[4] D. Stiliadis and A. Varma, ”Selective victim caching: a method to improve the performance of direct-mapped caches,” in IEEE Transactions on Computers, vol. 46, no. 5, pp. 603-610, May 1997.

[5] UC Davis Computer Architecture Research Group (July 12, 2019). Extremely Simple Microbenchmarks [Github].

[6] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. SIGARCH Comput. Archit. News 34, 4 (September 2006), 1–17.

[7] Lowe-Power, Jason & Ahmad, Abdul & Akram, Ayaz & Alian, Mohammad & Amslinger, Rico & Andreozzi, Matteo Maria & Armejach, Adrià & Asmussen, Nils & Bharadwaj, Srikant & Black, Gabe & Bloom, Gedare & Bruce, Bobby & Carvalho, Daniel & Castrillón, Jerónimo & Chen, Lizhong & Derumigny, Nicolas & Diestelhorst, Stephan & Elsasser, Wendy & Fariborz, Marjan & Zulian, Éder. (2020). The gem5 Simulator: Version 20.0+.

[8] Kathlene Hurt and Eugene John. 2015. Analysis of Memory Sensitive SPEC CPU2006 Integer Benchmarks for Big Data Benchmarking. In Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems (PABS ’15).

[9] N. Jindal, P. R. Panda and S. R. Sarangi, ”Reusing trace buffers to enhance cache performance,” Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, 2017, pp. 572-577.

[10] S. Kim et al., ”ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage,” 2020 IEEE Secure Development (SecDev), 2020, pp. 96-107.

[11] Wang, Jianfei et al. “Incorporating selective victim cache into GPGPU for high-performance computing.” Concurrency and Computation: Practice and Experience 29 (2017).

[12] Y. Oh, G. Koo, M. Annavaram and W. W. Ro, ”Linebacker: Preserving Victim Cache Lines in Idle Register Files of GPUs,” 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019, pp. 183-196.

[13] A. Asaduzzaman, M. P. Allen and T. Jareen, ”An effective locking-free caching technique for power-aware multicore computing systems,” 2014 International Conference on Informatics, Electronics & Vision (ICIEV), 2014, pp. 1-6.