Exploring the Latency Sensitivity of Cache Replacement Policies

Ahmed Nematallah, Chang Hyun Park, *Member, IEEE*, and David Black-Schaffer, *Member, IEEE*

***Abstract*—With DRAM latencies increasing relative to CPU speeds, the performance of caches has become more important. This has led to increasingly sophisticated replacement policies that require complex calculations to update their replacement metadata, which often require multiple cycles. To minimize the negative impact of these metadata updates, architects have focused on policies that incur as little update latency as possible through a combination of reducing the policies' precision and using parallel hardware. In this work we investigate whether these tradeoffs to reduce cache metadata update latency are needed. Specifically, we look at the performance and energy impact of increasing the latency of cache replacement policy updates. We find that even dramatic increases in replacement policy update latency have very limited effect. This indicates that designers have far more freedom to increase policy complexity and latency than previously assumed.**

***Index Terms*—Cache replacement policies, computer architecture, high-performance computing.**

I. Introduction

CACHE replacement policies play an essential role in the memory hierarchy as they significantly affect cache hit rates, and, thereby, performance. While many advanced replacement policies have been proposed, there are a range of factors that make them challenging to implement, including metadata update latency and storage, performance predictability, verification, security, etc. In this work we evaluate the impact of metadata update latency, as delayed updates affect the accuracy of replacement decisions which can hurt performance.

Existing work generally assumes low metadata update latency is important, and therefore proposes a range of optimizations to achieve it: parallelism [1], reductions in accuracy (sampling, reduced precision/history) [1], [2], [3], and/or simply leaving latency optimizations for future work [3]. Low metadata update latency also appears important for keeping up with the incoming stream of cache accesses. Indeed, back-to-back accesses make up approximately 14% of the total last-level cache accesses across SPEC2006, as shown in Fig. 1, and more than 50% occur within 23 cycles. In addition, metadata update latency is coupled to the metadata storage capacity: if a low update latency is required, then only small memories close to the update logic can be accessed in time.

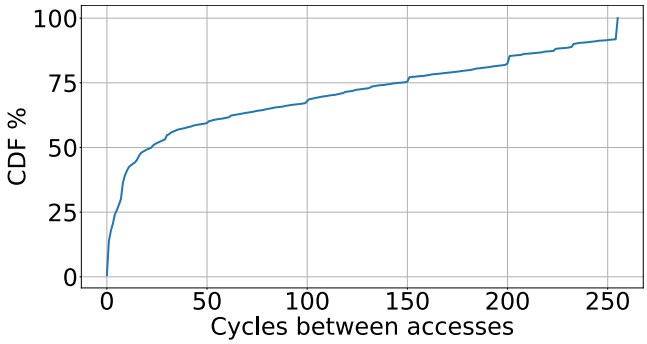

Fig. 1. CDF of cycles between LLC accesses of 28 benchmarks from SPEC2006. Accesses over 254 cycles are grouped in one bin.

In this letter we investigate whether these concerns are valid by exploring the impact on performance (IPC) and memory system pressure (MPKI, which is a proxy for the dynamic DRAM energy) of explicitly increasing the cache replacement policy latency by delaying cache replacement metadata updates for the LLC. We explore the latency sensitivity of multiple simple and complex cache replacement policies (LRU, DRRIP [4], Hawkeye [1], Mockingjay [2]) with varying amounts of delay (from thousands to hundreds of thousands of cycles). We also examine how the location of the delay affects the performance, as complex policies have multiple update steps of varying complexity. We show that low metadata update latency is not critical, which opens up for designs with simpler hardware (decoupled or reduced pipelining/parallelism update logic) and more complex updates (higher precision, more history, neural networks, etc.). In particular, by showing that metadata updates can tolerate thousands of cycles of delay, we demonstrate that it could be practical to use much larger metadata structures [3] stored in slower DRAM.

II. Methodology

We modeled increased metadata update latency by introducing a delay in the update computation for the replacement metadata. This is achieved by computing the metadata update from the input access data (e.g., which line to evict in LRU) periodically, as opposed to immediately upon access. Specifically, we process the metadata updates in a round-robin fashion. (e.g., for a replacement policy that collects per set status, we process each set using round-robin, rather than on each access.)

Manuscript received 21 June 2023; accepted 11 July 2023. Date of publication 19 July 2023; date of current version 14 August 2023. This work was supported in part by Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program under Grant 2015.0153, in part by European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Program under Grant 715283, and in part by Swedish Research Council under Grant 2019-02429. The computation was enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala University through the aforementioned grants. *(Corresponding author: Ahmed Nematallah.)*

The authors are with the Department of Information Technology, Uppsala University, SE-751 05 Uppsala, Sweden (e-mail: ahmed.nematallah@it.uu.se; chang.hyun.park@it.uu.se; david.black-schaffer@it.uu.se).

Digital Object Identifier 10.1109/LCA.2023.3296251

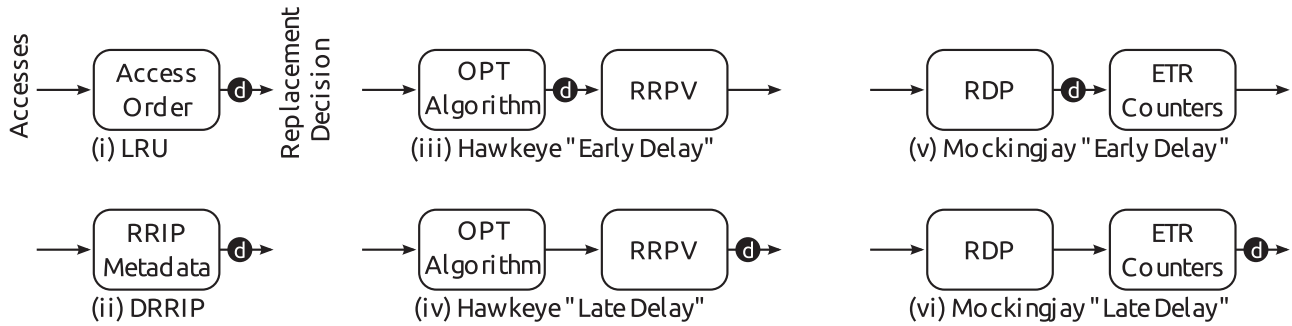

Fig. 2. Modeling metadata delay. The delay location is shown with a black circle d. LRU (i): we compute the update to access order from accesses but delay applying it to replacement decisions. DRRIP (ii): We update the RRIP metadata from accesses but delay applying it for replacement decisions. Hawkeye and mockingjay: both provide two options for delay: early in the update computation when intermediate metadata is updated (iii), (iv) or late, when the final replacement metadata is updated (v), (vi).

This means the average metadata age is determined by how rapidly the round-robin processing goes through the previous stage metadata.

This round-robin approach has two implications. First, since our updates are applied round-robin to all metadata entries, we compute updates for both modified and unmodified metadata and they are not applied in the cache accesses order. This means that we are at a performance disadvantage compared to an implementation that only processes modified metadata.[1] We chose this approach as it provides a simple model for decoupling the update processing latency from the rate at which updates arrive. Second, we make no assumptions about the implementation or pipelining of the update computation itself.

We modeled delays from 1024 to 512k cycles for 4 policies: LRU, DRRIP, Hawkeye, and Mockingjay. Fig. 2 shows where we insert delays for each policy. As Hawkeye and Mockingjay involve complex updates, we modeled two different locations for inserting the delay for each policy.

LRU and DRRIP are delayed by computing the updates to the main metadata arrays (Access Order for LRU and RRPV for DRRIP) and storing which line should be evicted next per set in another array and updating it in a round-robin fashion. We use the delayed data in the second array for the actual replacement decisions.

Hawkeye [3] works by running Belady's OPT algorithm [5] (the optimal replacement policy) on past accesses to predict which PCs produce cache-friendly or cache-averse accesses. It then uses these predictions as a parameter for the insertion policy of a slightly modified version of RRIP (where on a hit, we set the RRPV to either zero or the maximim RRPV value), which performs the actual replacement decisions. Hawkeye's approximation of Belady's algorithm relies on complex operations. In particular, the occupancy vector, which determines which lines are stored in a cache set, requires a scan of the history on every update to a sampled set to determine which cache ways would have been full between two accesses to a line. Performing this update in a single cycle is a challenge, as it requires 32 iterations. There are 32 sets designated as sampled sets, and this means that multiple parallel calculation units may be needed to handle different accesses to sampled sets, especially if the hardware needs multiple cycles for computations.

As a result of this two-part design (first the OPT Algorithm processing and then the use of its output for the modified RRIP policy), there are two logical places where one could insert a delay. Early: after running Belady's OPT algorithm, thereby delaying the update of the cache friendliness metadata used by RRIP, but keeping the RRIP metadata used for replacement up-to-date, or, Late: delaying the RRIP metadata update itself, as we did in DRRIP, making replacement based on stale metadata. These two delays are shown in Fig. 2(iii) and (iv) respectively. The Early delay is plausible from an implementation point of view because the update of RRIP metadata takes very little time to compute and has already been implemented in hardware [6] while the computation for the OPT algorithm is much more complex, and therefore of more interest in terms of trading off latency for implementation cost.

Mockingjay [2] also tries to mimic Belady's OPT policy. However, instead of running Belady's OPT algorithm on the history to figure out which instructions are cache-friendly or cache-averse, it tries to directly predict future reuse distances from past ones and uses those predicted reuse distances to make decisions on which cache line to evict. This is very similar to Belady's OPT policy except that it relies on predicted reuse distances as opposed to actual knowledge of future accesses.

Mockingjay predicts reuse distances by storing timestamps of past accesses for sampled sets along with the PCs that invoked them and using re-accesses to compute an expected future reuse distance for each PC. These reuse distances are then stored in the Reuse Distance Predictor (RDP). Whenever a PC accesses a line, the predicted reuse distance is then used to update a per-line Estimated Time of Reuse (ETR) counters to indicate when the line is expected to be reused. The counters are decremented on accesses to the cache, and the replacement policy selects the line with the largest absolute counter value (time of reuse farthest from current time) to evict.

As with Hawkeye, there are two plausible locations for delay in Mockingjay: Early (Fig. 2(v)), which provides delayed predicted reuse distance updates, or Late (Fig. 2(vi)), which delays

[1] We confirmed this by testing in-order updates, which behaved as expected: a slight overall improvement with the most pronounced effects on the applications with the largest performance drops.

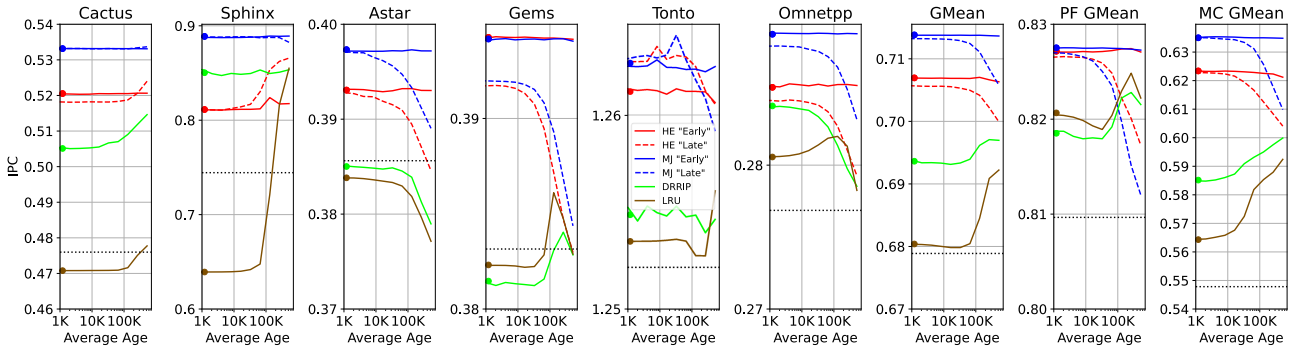

Fig. 3. Performance (IPC) as a function of average metadata age (delay). Rightmost: geometric means of prefetching (PF) and multicore (MC). Dotted lines represent random replacement IPC. Dots on the y-axis show the IPC of each policy without any delay. sphinx's surprising behavior is discussed in the results section.

TABLE I
SIMULATOR CONFIGURATION

CPU	OoO core, 352 entry ROB, LQ/SQ size = 128/72, Fetch/Decode/Execute/Retire width = 6/6/6/4
L1I	32KiB, 8-way, 4-cycle latency, 8 MSHRs
L1D	48KiB, 12-way, 5-cycle latency, 16 MSHRs
L2	512KiB, 8-way, 10+4/5 (I/D) cycle latency, 32 MSHRs
LLC	2MiB, 16 ways, 64B line, 2048 sets, 20+14/15 (I/D) cycle latency, 64 MSHRs

updates to the ETR counters, thereby using out-of-date estimates of predicted reuses when evicting. As with Hawkeye, updating the ETR counters (Late) is much simpler than the resuse distance update (Early).

III. RESULTS

We evaluated the impact of delaying LLC metadata updates using ChampSim [7] with the configurations in Table I, without prefetching. We evaluated the four replacement policies and two delay locations for two of them across 28 SPEC2006 benchmarks using traces from the second cache replacement championship (CRC2). Due to space constraints, we show six benchmarks and the mean for the whole suite. We varied the average metadata update delay from 1024 to 512k cycles to show the point at which delay matters.

We also enabled next-line prefetching on all cache levels and randomly picked 16 sets of 4-core multi-programmed workloads for a multicore evaluation. We present the means of the prefetching and multicore configurations. Additionally, we tested halving and doubling the cache size (1MB and 4MB) to confirm our results held across configurations. The results were largely similar to the default 2MB configuration.

Performance (IPC) is shown in Fig. 3 for each of the replacement policies as a function of the average metadata update delay in cycles (logarithmic x-axis). Even with average delays of up to 100k cycles, we see less than 0.5% IPC change across nearly all benchmarks. This demonstrates that even at quite substantial delays the performance decrease is insignificant.

One of the main reasons for the lack of impact of delayed metadata updates is that the rate of change in each set's metadata is already low. For example, we see an average of 775k cycles between accesses to the same set (ranging from 65k for lbm to 146M for povray). This suggests that updating per-set metadata more frequently is unnecessary. This is confirmed by more investigation into Hawkeye (Late) where we store the line to evict per set in a vector that gets updated periodically. This means that a second miss on the same set (before a metadata update) will evict the last (most recent) item installed, since the eviction metadata has not been updated. We changed this policy to immediately update the metadata of a set after an insertion. We found that even though this recovered 2/3 of the performance loss in astar, overall, the geometric mean only saw a 0.1 percentage point improvement.

Performance increase: Several benchamarks showed a performance increase with increasing delay for LRU and Hawkeye (Late). For example, sphinx sees about a 33% and 6% IPC increase for LRU and Hawkeye (Late), respectively. We believe this is because sphinx's working set is large enough to thrash the 2MB LLC. As Mockingjay and DRRIP are more thrash-resistant, they do not exhibit this behavior. To investigate this, we tried random replacement, which yielded better results than LRU for sphinx. Furthermore, as sphinx has a working set size between 2MB and 4MB [8], using a 4MB cache significantly reduced the performance gain.

We believe that Hawkeye performs poorly with cache thrashing in this case since it only classifies PCs as cache friendly or cache averse and then uses RRIP to perform the actual cache replacement. If there is just one PC, or a few PCs that are bringing in cache lines, and they all have the same classification (e.g., cache averse), then Hawkeye would perform similarly to SRRIP, which lacks the dynamic thrash-resistant attributes of DRRIP.

On the other hand, this changes for the advanced policies which use more global metadata, such as histories, which are updated on accesses to multiple different sets. In our experiments, the "Early" Hawkeye and Mockingjay versions delay the global metadata while the "Late" ones delay the per-set metadata. While the global metadata is changed far more frequently (approaching the rates of Fig. 1), it is also more stable (e.g.,

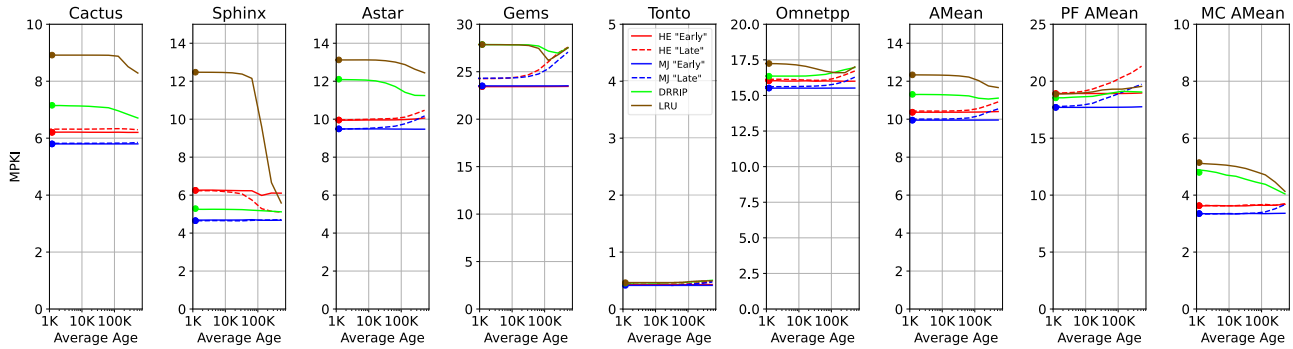

Fig. 4. MPKI as a function of the average metadata age (delay). Rightmost: arithmetic means of prefetching (PF) and multicore (MC). Dots on the y-axis show the MPKI of each policy without any delay.

the behavior of a particular PC does not change frequently) and therefore also highly insensitive to update latency.

DRAM accesses (LLC MPKI) are shown in Fig. 4. We see that MPKI correlates with performance. The magnitude of change is not always the same, but the trends are. For example, if we look at sphinx, specifically for LRU, there is a large drop in MPKI that corresponds to the IPC increase, and in GEMS we notice that the IPC increases until the average age reaches a few hundred thousand cycles, then drops again, which correspond to a dip then recovery of the MPKI. We also see that at very large delays, the MPKI actually decreases for LRU and DRRIP, which shows how far from optimal they can be.

As with other work, for estimating dynamic energy, we measure MPKI excluding misses occurring from writebacks, as these result in data moving from a lower-level cache to the LLC, and thereby do not significantly affect DRAM energy. We used the arithmetic mean for aggregating the MPKI results as it is more suitable for showing absolute differences as opposed to the geometric mean, which shows relative differences more clearly.

Prefetching shows a similar performance and MPKI trend, but with a smaller magnitude. Multiprogrammed runs also shows a similar trend, but with a larger magnitude, especially at larger delays. (Figs. 3 and 4).

IV. Implications

We have shown that the efforts to keep the latency of cache replacement updates low for the LLC are largely unwarranted from a performance point of view, even for extremely large delays of tens-to-hundreds of thousands of cycles. This hints at the potential to implement far more complex replacement updates, including programmable replacements and/or policies that access much larger memory structures, with little performance impact. Based on our findings, efforts to tradeoff policy accuracy to reduce update complexity and memory requirements (e.g., [1], [2], [3]) should be reconsidered. Indeed, the extreme insensitivity to delay we have observed indicates that it is likely better to have the increased precision and delay, rather than reduced precision and lower delay. The impact on MPKI is also minimal with increasing delay which means DRAM energy should not change much.

However, for very large delays, the location of the delay can impact performance. We found that delays to updating the final replacement data in complex policies (Late delays in Hawkeye and Mockingjay) have a larger performance and MPKI impacts than the Early delays, which keep those simpler parts of the metadata update process unmodified. This insight suggests that complex policies should separate their slow, complex updates from faster, simple decisions that can be kept up-to-date with low overhead. Our results show that such a partitioning allows designs to tolerate very significant delays with nearly no impact on performance. Interestingly, both the Hawkeye and Mockingjay designs are partitioned this way, although neither did so for this purpose.

References

[1] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *Proc. IEEE/ACM 43rd Annu. Int. Symp. Comput. Architecture*, 2016, pp. 78–89.

[2] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's MIN policy," in *Proc. IEEE Int. Symp. High-Perform. Comput. Architecture*, 2022, pp. 558–572.

[3] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *Proc. 37th Int. Conf. Mach. Learn.*, 2020, pp. 6237–6247.

[4] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. 37th Annu. Int. Symp. Comput. Architecture*, 2010, pp. 60–71.

[5] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[6] Henry, "Intel ivy bridge cache replacement policy," Jan. 25, 2013. [Online]. Available: https://blog.stuffedcow.net/2013/01/ivb-cache-replacement/

[7] N. Gober et al., "The championship simulator: Architectural simulation for education and competition," 2022, *arXiv:2210.14324*.

[8] M. Hassan, C. H. Park, and D. Black-Schaffer, "Architecturally-independent and time-based characterization of SPEC CPU 2017," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2020, pp. 107–109.