

Cache Block Watermarking: Over-engineered and Underwhelming

Cesar Gomes*, Steven Battle[†], Siddarth Nilakantan[‡], and Mark Hempstead*

Affiliation(s): *Tufts University, [†]IBM, [‡]Nvidia

Abstract—Increased last level cache capacity, poor scaling of the least recently used cache management policy, and proliferating approximations of reuse distance sparked a lot of research in replacement policies. Many modern policies exploit data patterns and properties of the cache such that the capacity can be used for effectively. We propose Cache Watermarking to manage last level cache in light of *ping-pong* behavior exhibited by program access patterns. *Ping-pong* behavior sees blocks being evicted shortly before reuse, incurring an off-chip access. Our technique evicts, promotes, and inserts cache blocks based on observed reuse in context of an LRU-based reuse stack with the intent of prioritizing removal of short-lived or *dead* blocks. We detail efforts to apply watermarking to each cache block, as well as adding sampling techniques like set dueling and sampling. We further evaluate how the technique performs in context of a prefetcher and different cache inclusion principles. In each scenario, we discuss how watermarking contributes or conflicts with the current context of analysis.

I. INTRODUCTION

The Least Recently Used (LRU) policy demonstrates diminishing performance and intractability of overhead state as caches grow larger. Qureshi et. al. [1] suggests that streaming and thrashing access patterns negatively impact cache efficacy. They propose Dynamic Insertion Policy (DIP) which uses set-dueling as a method of implementing two insertion policies within a level of cache. A global counter is updated pending access behavior of certain *leader* sets. The global counter is consulted by non-leader sets in order to decide which policy is the best choice. DIP is built on top of LRU. Khan et. al. [2] suggests cache efficacy is compromised by blocks that never hit in cache, or are *dead*. They propose correlating sampled cache miss behavior to program counter values as a means of predicting blocks that do not hit, or *die*, in the cache. This is also built on top of LRU and its related approximations to varying effect. Jaleel et. al. [3] suggest intraset cache reuse can be generalized along a proposed re-reference chain rather than a reuse stack (as in LRU). They use a proxy for this re-reference chain to implement Re-Reference Interval Policy (RRIP) and build set-dueling-based [3] and sampling-based [4] techniques on top of this base policy.

We approach improving cache efficacy by minimizing dead lines at a finer granularity. We propose cache block watermarking as a method of minimizing the *dead* time between when a block is hit and when it is evicted. In so doing, the technique minimizes *ping-pong* behavior which causes cache blocks to repeat a pattern of eviction and insertion without any reuse. In this paper, we discuss ping-ponging behavior in brief detail,

how a watermarking works, how the design of watermarking evolved, and the shortcomings built into such a technique in context of modern cache design.

II. EXPLOITING CACHE ACCESS BEHAVIOR

Cache performance depends on the access patterns exhibited by the workload. A cache which prioritizes near re-use will be sub-optimal for workloads with little locality. We define a specific type of behavior caused by sub-optimal cache eviction policies: *cache ping-pong*, where cache blocks are evicted and re-inserted within a defined window of insertions. This behavior is exacerbated by *dead-line insertion*, where space in the cache is occupied by lines that will not be re-referenced prior to eviction. This is a problem in LRU-like schemes which favour frequently re-used lines as the dead-line must traverse the entire LRU stack before being an eviction candidate. A replacement policy which prioritizes eviction of *dead-on-insertion* blocks will make room for *ping-ponging* cache blocks. In last-level caches, non-LRU-like policies can take advantage of these dead-lines and prioritize them for eviction. We propose a novel watermarking scheme which adapts the LRU stack depth to the access patterns experienced by the cache line and a light-weight prediction mechanism to identify dead-lines at block insertion.

A. Ping Pong Access Patterns

In the context of this work, we define *re-use distance* of a line in a set as the number of *unique* accesses to that set in between two consecutive accesses to the same line [5]. This is in contrast to other definitions of re-use distance that count all intervening accesses to that set, including multiple accesses to the same lines [6], [7]. The advantage of our definition of re-use distance is that it can be easily measured at run-time by examining the “LRU stack”. Every set in an LRU cache can be logically represented as a stack of ‘ways’, ordering each ‘way’ in the set from most-recently-used (MRU) line to LRU. The MRU position is on the “top” of the stack. This can be extended to pseudo-LRU, which approximates the LRU-stack with a tree. However additional state may be needed to encode the depth.

Figure 1 illustrates the lifetime of an example cache line with LRU replacement. Line ‘A’ is inserted at time $t(0)$ at the MRU position of the set. Every following hit to ‘A’ is marked in a gray bubble. The dashed arrows connecting the gray bubbles represent the re-use distance between those accesses.

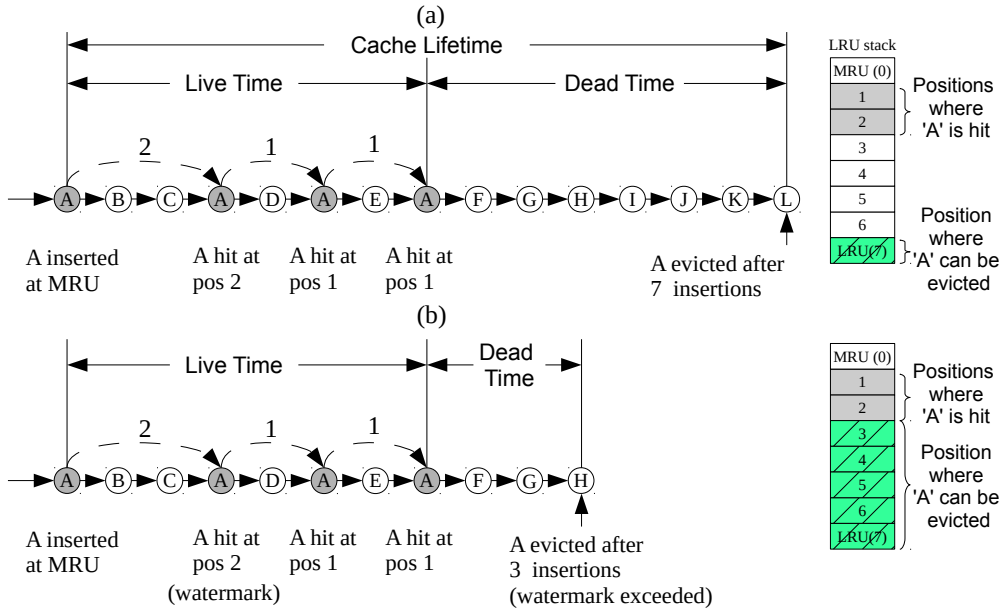


Fig. 1. (a) Typical lifetime of a cache-line with LRU replacement showing depth in LRU stack for 8-way set. Line is evicted after reaching LRU. (b) Lifetime of cache-line with watermark history. Line is evictable after the watermark is exceeded, reducing dead-time.

The text below each gray bubble also indicates the position in the LRU stack where the line is hit. Every time A is hit, it moves back to the MRU position. Line A is hit 3 times with a maximum re-use distance of 2. It is finally evicted when it reaches LRU, after the last access. This results in a *deadtime* of 7. For LRU, when a line is last-accessed, it must wait for accesses to N unique lines, where $N = \text{associativity} - 1$, before falling to the LRU position in the stack and being replaced by the next insertion to the cache set. During this *dead-time* other *live* lines would make better use of the cache.

Figure 1 (b) shows how a cache block can be evicted earlier, if knowledge of its prior access behavior is known. A replacement policy can exploit the regularity of caches accesses (e.g. from loops) to make more informed decisions when evicting a line. If the lowest position in the stack is recorded after a hit, a block can be expected to not exceed that distance in its lifetime.

B. Cache-Hit Behavior

Figure 2 show how LRU replacement can be inefficient in the context of last-level caches where temporal locality has been filtered. This figure shows the distribution of ‘hit’-positions in the last-level cache’s LRU stack for several workloads from the SPEC2006 [8] suite. These workloads were simulated for 1 Billion instructions with a 2-MB, 16-way L3 cache, recording the position in the LRU stack when a cache-block was re-used (hit) after insertion.

The top row shows the position in the LRU stack when a line is ‘hit’. The bottom row shows the position of the line for the ‘hit’ immediately preceding the eviction of the block. The ‘last hit’ position mirrors the overall ‘hit’ pattern observed by the workload. This indicates that the cache access

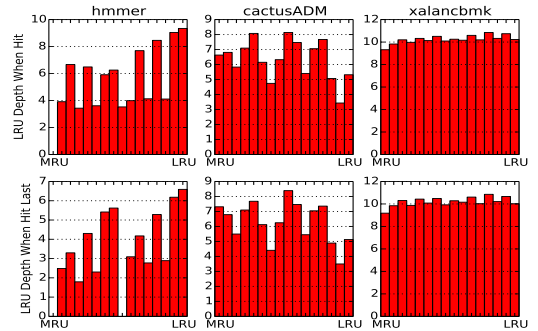


Fig. 2. Hit analysis for hmmer, cactusADM, and xalancbmk. Top: Positions in pseudoLRU stack where cache-block is hit. Bottom: Position of last hit prior to eviction

pattern is regular on aggregate. Tracking the previous hit history of a cache-block will encode this hit-behavior into the cache eviction decision. We exploit this predictability and repeatability in order to make better eviction decisions and discover (and predict) which instructions insert dead-lines into the cache.

III. METHODOLOGY

A. Simulator

We built all versions of cache block watermarking into ChampSim [9], which was the simulator used in the recent Cache Replacement Competition [10]. The version we use is a full version which was recently released. The simulator was modeled after the Intel Skylake [11] (see Table I). In addition to the watermarking policies, we have DBP, SHiP, and Perceptron to compare against (see Table II). For the

TABLE I
ARCHITECTURAL PARAMETERS

	Reorder Buffer Entries	Width	Branch Prediction	Core Model
ChampSim	256	6	Bimodal	Intel Skylake
	Capacity/core	Associativity	Block Size	
L1(D/I)	32KB	8	64B	
L2	256KB	8	64B	
L3	2048KB	16	64B	

TABLE II
REPLACEMENT AND INSERTION POLICY EVALUATION PARAMETERS

Predictor	Policy	Sampler	Predictor Table Count	Bypass
DBP	LRU	Sampler Cache	3	Yes
SHiP	Static RRIP	Random Sets In Cache	1	No
Perceptron	pLRU	Sampler Cache	6	Yes

prefetching analysis, please note that a stride prefetcher is implemented in ChampSim, in line with Intel L2 Streamer Prefetchers

B. Benchmarks

The benchmarks used to evaluate the replacement policies are from the SPEC 2006 benchmark suite [8]. Utilizing the simpoints methodology [12] on 29 SPEC 2006 benchmarks, we chose a 1 billion instruction slice from each run with the highest weight as a representative instruction trace.

IV. EVOLVING CACHE WATERMARK

In designing cache watermark, an obvious issue lies in the overhead state, erroneous watermarking, and unknown, general behavior. We propose a compressed method of extracting reuse from an approximation of LRU to address the overhead issue. We also apply methods like set-dueling and sampling to regulate erroneous watermarking and predict watermarks. Lastly, in response to unknown behavior that can occur when sampling, we depend on general hit behavior.

A. Approximate Watermark: pLRU Tree Distance

Remembering a watermark would double the replacement state overhead of an already expensive base policy. We address this in two ways: using pseudo LRU (pLRU) as the base policy; and approximating reuse distance in context of this pLRU. We use tree-based pseudo LRU, which stores <1 bit per cache line in a given set. Further, we borrow methods to extract pseudo LRU stack positions that are detailed in Jiménez [13]. Considering the cost to keep a full stack position, we need sought an alternative to reuse that could be extracted from the pLRU tree itself. To this end, we propose the idea of *tree distance* as a proxy for reuse. The tree distance in this context refers to the number tree node toggles required to move a line from the current pseudo stack position to the most recently used (MRU) position on a hit. For example, assume there is a hit to a block in a 16-way set-associative cache. If a block is hit when the pseudo stack position method calculates a position of 7, this requires 3 tree nodes be toggled in order to move the hit block to MRU. Therefore, we store a tree distance of 3. In total, for a 16-way set-associative cache, we reduce overhead to be comparable to RRIP (<3 bits per block vs. 3 bits per block).

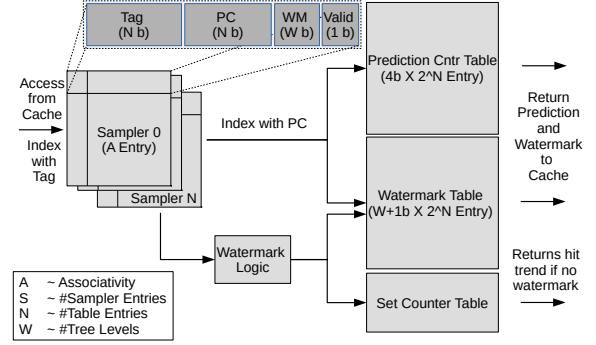


Fig. 3. Sampler-based Cache Watermarking Hardware Diagram

B. Predicting Watermark: Set-dueling and Set-sampling

Using watermarking alone as a replacement policy does not help performance as well as expected. In fact, it can increase misses because there are access patterns which work best with LRU and the related approximations of LRU. We address this in two ways: set-dueling watermarking with pLRU to address phases where watermarking fails; and set-sampling with algorithms to manipulate pLRU trees according to predicted watermarks. Set-dueling is implemented in the same fashion as DIP [1] and DRRIP [3], so details of this implementation are not explained. In context of set-sampling, this version of watermark can be perceived as an augmented hit predictor. By using a sampler, we are watermarking sampled behavior, thus increasing the overhead of that sampler. Further, we have two options: either store this predicted watermark with a block, or manipulate a block on insertion and allow an update until a block is predicted to not be hit. The first option evicts in the manner described previously. The second option translates watermarks to the number of tree nodes to toggle from root to leaf (way). This value is used to move a newly inserted block near MRU with respect to the associated watermark prediction as opposed to MRU. Like DBP [2], SHiP [4], and Perceptron [14], we modify watermark for intervention at insertion and promotion, rather than tracking additional information per block.

C. Generalize Watermark: Set-wise Hit Behavior

While addressing some overhead and prediction concerns, we've introduced an issue of unknown behavior reuse behavior. Set-dueling addresses this by only operating in the two states dictated by the set dueling counter and set dueling monitors. Sampler based predictors also assume two states, with hit predictors determining if a given block is hit or not after the current access. By adding reuse distance, we've introduced unknown reuse on first access in the scenario where a block transitions from *dead* to *live* in the cache. In these situations, we added set-wise reuse counters for generalized hit behavior. These counters are consulted when there is not watermark.

After addressing the above, we show the final hardware diagram in Figure 3. It is worth noting that this sampler

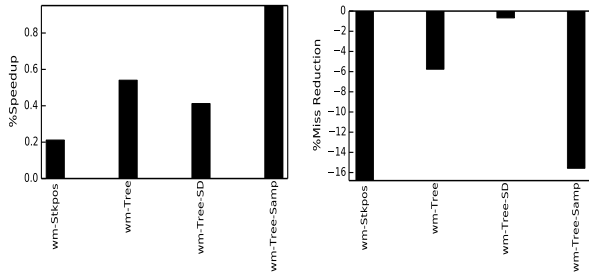


Fig. 4. Geometric Mean Speedup of All Watermark Iterations, from left to right: pseudoLRU Stack Position based WM, Tree-distance based WM, Tree-distance based WM set duelled with pLRU, and sampler TreeWM and set-wise behavior counters

uses pLRU in order to better In the figure, we see constants representing number of sampler entries, table entries, and watermark width. For brevity, we do not discuss tuning so we present a good configuration where S, N, and W are 64, 12, and 2, respectively. Figure 4 shows the geometric mean speedup demonstrated for each version of watermark for the SPEC 2006 benchmark suite [8].

V. RESULTS AND ANALYSIS

Figure 5 shows results for all benchmarks in SPEC 2006. The policies that we are evaluating are shown in this order: DBP, SHiP, and the sampler-based version of Watermarking we discussed in section IV, hereby known as WM-Tree. These results are in comparison to a pLRU back, 2MB, inclusive LLC, and higher is better. The from the speedup figure (left), we see that there are certain workloads which benefit greatly from all of these prediction based policies (gcc, mcf, sphinx3). Ther are even a few that prefer one predictor over another (e.g. lbm, leslie, libquantum, soplex, and wrf favoring DBP). However, the remaining workloads show little to no benefit from these predictors. In fact, the figure on the right which shows % of MPKI reduced highlights that these techniques increase misses, but the workloads are either tolerant to this loss or slow down. Figure 6 shows the average Speedup and MPKI reduction values for each predictor. DBP, SHiP, and WM-Tree have 2.2%, 1%, %, and 0.9% speedup, respectively. Further, the four predictors demonstrate MPKI increases of 28%, 7%, %,and 19%, respectively.

DBP has the greatest speedup, but also the greatest MPKI increase. Predictor accuracy plays a part here. The three skewed table structure implemented in DBP allows for block "revival," or for some correlating information to not get stuck in either a *dead* or not state. The indexing scheme prevents this, but that does not make DBP immune to mistakes, as is evidenced by the MPKI increase. SHiP does not have the performance impact of DBP, but it also does not miss as often. While only using one predictor table, being built on top of a better performing base policy (SRRIP) and not bypassing allows SHiP to minimize incorrect evictions. Perceptron outperforms both SHiP and WM-Tree, but does not match up to DBP in this context. The reasoning can be attributed to not

tuning the policy parameters to a 2MB cache, and reliance on bypassing.

However, this cannot be said for WM-Tree. WM-Tree depends on two tables, one with a watermark prediction and one with prediction counters (similar to SHiP). In applying watermarking to a predictor, we did not address the fact that once a WM is set, dead or not, it would not be unset. Further, pLRU can only guarantee the MRU position, and therefore can only guarantee a tree distance of 0 assuming the MRU block is hit in consecutive accesses. Another issue with tree distance as a reuse proxy is aliasing of reuse distance. Given the design of samplers and the overhead restrictions, assuming local data can contribute to generalized behavior in this way is naive. Having said that, we were curious to see if removing the overhead state restrictions would allow WM-Tree to show some improvement. The results showed that we could match DBP in terms of performance given and *unlimited* watermark table, which is not practical. The implication of this analysis is that *ping-pong* behavior is not prevalent enough within workload patterns to warrant recording local reuse information in a naive fashion given these resource constraints placed on modern caches. Lastly, modern versions of predictors [14], [15] that out perform DBP have been published since 2010.

A. Prefetching

Prefetchers are common components of a given cache hierarchy, so evaluating these techniques in that context modernizes our configuration in one aspect. Figure 7 shows the average Speedup and MPKI reduction values for each predictor in context of a stride streaming prefetcher. All performance metrics are in comparison to a pLRU backed, 2MB inclusive LLC with the mentioned prefetcher, and higher is better. Here, performance trends remain the same as in the non-prefetching case discussed in the previous subsection. However, WM-Tree now does worse than the base configuration. We attribute this to the prefetches being assigned watermarks before they have a chance to be inserted into the cache. While we attempt to mitigate that with general hit information provided by the set-wise counters, this does not prevent the decrease in performance.

B. Inclusiveness

Cache inclusiveness is also different in modern caches which is made evident by non-inclusive caches in the the Skylake [11] architecture that ChampSim [9] is based on. Figure 8 shows the average Speedup and MPKI reduction values for each predictor in context of a non-inclusive LLC. All performance metrics are in comparison to a pLRU backed, 2MB non-inclusive LLC, and higher is better. In this context, DBP still outperforms the other 2 techniques, but by a slimmer margin. The implication here is that non-inclusive caches are more forgiving of incorrect insertion and eviction predictions, which can simultaneously hide naive technique errors

In attempting to add local characterization to reuse prediction with watermarks, we've inadvertently lessened the

pp. 244–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=942806.943854>

- [13] D. A. Jiménez, “Insertion and promotion for tree-based pseudolru last-level caches,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 284–296. [Online]. Available: <http://doi.acm.org/10.1145/2540708.2540733>
- [14] E. Teran, Z. Wang, and D. A. Jiménez, “Perceptron learning for reuse prediction,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.
- [15] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, “Kill the program counter: Reconstructing program behavior in the processor cache hierarchy,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 737–749. [Online]. Available: <http://doi.acm.org/10.1145/3037697.3037701>