Cache Replacement Policy Based on Expected Hit Count

Armin Vakil-Ghahani[§] Sara Mahdizadeh-Shahri[§] Mohammad-Reza Lotfi-Namin[§]
Mohammad Bakhshalipour[§] Pejman Lotfi-Kamran[‡] Hamid Sarbazi-Azad[§‡]
§ Department of Computer Engineering, Sharif University of Technology
‡ School of Computer Science, Institute for Research in Fundamental Sciences (IPM)
{vakil,smahdizadeh,mrlotfi,bakhshalipour}@ce.sharif.edu,plotfi@ipm.ir,azad@sharif.edu

ABSTRACT

Memory-intensive workloads operate on massive amounts of data that cannot be captured by last-level caches (LLCs) of modern processors. Consequently, processors encounter frequent off-chip misses, and hence, lose significant performance potential. One of the components of a modern processor that has a prominent influence on the off-chip miss traffic is LLC's replacement policy. Existing processors employ a variation of least recently used (LRU) policy to determine the victim for replacement. Unfortunately, there is a large gap between what LRU offers and that of Belady's MIN, which is the optimal replacement policy. Belady's MIN requires selecting a victim with the longest reuse distance, and hence, is unfeasible due to the need for knowing the future. In this work, we observe that there exists a strong correlation between the expected number of hits of a cache block and the reciprocal of its reuse distance. Taking advantage of this observation, we improve the efficiency of last-level caches through a low-cost-yet-effective replacement policy. We suggest a hit-count based victim-selection procedure on top of existing low-cost replacement policies to significantly improve the quality of victim selection in last-level caches with low area overhead. Our proposal offers 12.9% performance improvement over the baseline LRU in a multi-core processor without data prefetchers and outperforms state-of-the-art replacement policies.

KEYWORDS

CRC-2, memory system, memory-intensive workload, last-level cache, replacement policy, Belady's MIN, expected hit count.

1 INTRODUCTION

Multi-gigabyte datasets of memory-intensive workloads dwarf on-chip caches and reside in memory. Consequently, existing processors encounter many off-chip misses, and hence, lose significant performance potential when they execute these workloads.

One of the components in modern processors that has an eminent impact on the number of off-chip misses is the *replacement policy* of the last-level cache (LLC). As the size of the dataset in memory-intensive workloads is much larger than the on-chip capacity, modern processors frequently require evicting a piece of data from the last-level cache to open room for a new piece of data. A replacement policy determines, out of all possible candidates, which one should be evicted from the cache upon arrival of a new piece of data.

Existing processors use a variation of *least recently used (LRU)* policy to determine the victim. Many studies pointed out the deficiencies of LRU for common access patterns [1, 6, 8, 15]. The deficiencies lead to a large gap between what LRU offers and the opportunity.

The optimal replacement policy (i.e., Belady's MIN) evicts a piece of data that is going to be referenced further in the future. As the optimal replacement policy requires knowledge of the future references, and hence, is not feasible, all practical replacement policies use some indicators to guess which piece of data will be referenced further in the future [6, 8, 15].

In this work, we make the observation that there exists a strong correlation between the expected hit count of a cache block and the reciprocal of its reuse distance. This means that out of all potential replacement candidates, the one with the lowest expected hit count is likely to be referenced further into the future. Based on this observation, we propose *Expected Hit Count (EHC)* policy for replacement of cache blocks in last-level caches. EHC is an effective, low-cost replacement policy that associates an expected hit count to a block and seeks to evict the block that is expected to have *fewer* hits in the future.

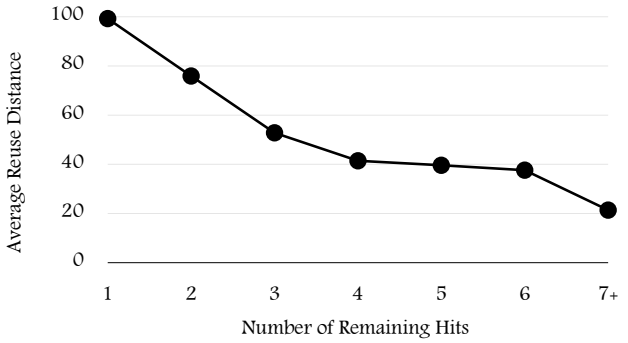

Figure 1: The average reuse distance as a function of the number of remaining hits of cache blocks during their lifetime in a cache with Belady's MIN replacement averaged across all workloads. Bucket 7+ represents all cache blocks with at least 7 remaining hits.

Using cycle-accurate full-system simulation infrastructure, we evaluate our proposal for both single-core and multi-core processors on a diverse set of workloads. Our results show that EHC offers, on average, 3.5% (12.9%) speedup on a single-core (multi-core) processor, and 11% MPKI reduction over the baseline LRU and outperforms state-of-the-art replacement policies.

The rest of this paper is organized as follows. Section 2 motivates the intuition behind the EHC replacement policy. Section 3 outlines the details of our proposal. Section 4 explains our methodology for evaluating EHC. Section 5 presents the evaluation results and Section 6 concludes the paper.

2 MOTIVATION

In this work, we make the observation that reuse distance of a cache block has a strong correlation with the reciprocal of its remaining number of hits in the current lifetime in the cache: the larger the number of remaining hits, the sooner the cache block is re-referenced. Figure 1 shows the average re-reference distance of cache blocks as a function of the number of remaining hits, when LLC benefits from the Belady's MIN replacement algorithm. The reported numbers are averaged over all benchmarks. In this experiment, we associate the number of accesses between two re-references of a cache block in a set (i.e., reuse distance) to the number of remaining hits of the cache block with Belady's MIN. We report the average reuse distance of all cache blocks for hit counts of 0 to 7 in Figure 1. The bucket named 7+ includes all cache blocks with at least 7 hits.

As Figure 1 clearly shows, there exists a strong correlation between the reuse distance and the reciprocal of hit counts. On average, as the number of remaining hits of cache blocks decreases, their reuse

distance increases. We take advantage of this phenomenon to propose an effective and low-cost replacement policy for last-level caches.

Many pieces of prior work [7–13, 15] enhanced replacement decisions by identifying and evicting/bypassing dead (on arrival) blocks. A dead block is a cache block with no further reference during its current lifetime in the cache (i.e., blocks with zero remaining hits). Our proposal is a generalization of prior work: instead of just relying on dead blocks (i.e., blocks with zero remaining hits), our proposal estimates the expected number of hits of a cache block (zero or larger) and benefits from that information in the decision making process.

We found that prior proposals that employ a dead-block predictor for making replacement decisions fundamentally suffer from one or two of the following problems: (1) whenever a live block mistakenly identifies as dead, a cache miss is inevitable, and (2) whenever all blocks in a set are predicted as live blocks, the replacement policy is unable to effectively choose the best victim for replacement.

3 THE PROPOSAL

Taking advantage of the correlation between the reuse distance and the reciprocal of hit counts, we suggest a replacement policy that benefits from the remaining number of hits of cache blocks to make good replacement decisions. As we need to measure the number of hits that a cache block experiences, we need a baseline replacement policy. The measured hit count will be used to improve the decision making process of the baseline policy. Although, we can use any low-cost replacement algorithm as our baseline replacement policy, in this paper, we use Dynamic Re-Reference Interval Prediction (DRRIP) [8], as it has low storage overhead and offers good performance.

The number of hits of a cache block is not directly available, so we suggest a simple hit-count predictor. The predictor relies on the repetitiveness of control flow and data accesses [3] to determine the *Expected Hit Count (EHC)* of a cache block based on the number of hits of the block in its prior residencies in the cache. For each cache block, we store the number of hits that it has experienced in their past two residencies in the cache. We use the average number of hits in the past two residencies as the expected hit count.

Unfortunately, storing the hit history of all cache blocks imposes significant storage overhead. Therefore, instead of storing the hit history per each cache block, we collect the hit records per each region, which is represented by a *tag* in the cache. As the number of unique tags in the LLC is much smaller than the number of unique blocks, the area overhead of the predictor reduces (at least by one order of magnitude). We found that storing the history of hits per each tag does not considerably reduce the accuracy of predictions. Our finding corroborates prior work that showed blocks with the same tag share an identical spatial region and exhibit similar behavior [15].

3.1 Design Overview

Our approach, named Expected Hit Count (EHC), is built on top of the baseline replacement policy and employs a metadata table, named *Hit History Table (HHT)*, for storing the hit counts of two prior residencies of each tag. Figure 2 shows an overview of HHT. The table is a 16-way associative structure to offer low conflict rates and contains the following entries:

Valid. A single bit indicating whether the entry is valid or not.

LRU Recency. The HHT is managed with LRU replacement policy. We also evaluated sophisticated replacement policies for HHT but did not observe a notable performance improvement.

Tag. Stores part of the tag bits of a cache block in the LLC that is not used as an index to HHT.

Hit Count Array. Each entry in the HHT is equipped with two 3-bit saturating counters each remembers the number of hits of the corresponding cache block in a specific residency of the block in the LLC. These counters constitute a 'Hit Count Array' and are managed as a FIFO queue.

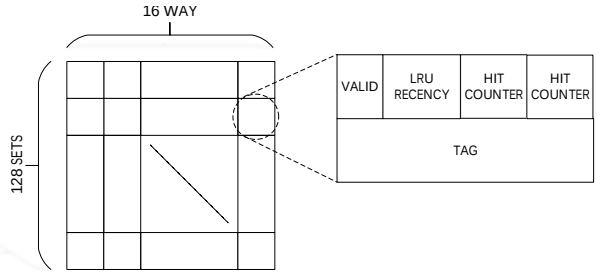

Figure 2: The overview of the hit history table.

EHC also requires extending the tag storage of each block in the LLC by a 3-bit storage unit. The storage unit, named *Current Hit Counter*, is a saturating counter that counts the number of hits experienced by the block in its current residency in the LLC.

3.2 Updating Metadata

The HHT gets updated either when *Current Hit Counter* saturates or when a block gets evicted from the cache and its *Current Hit Counter* is not saturated. Whenever one of these events occurs, we look up the HHT with the tag of the cache block. If the lookup results in a hit, we make the corresponding entry in the HHT as MRU, and push the *Current Hit Counter* to the front of *Hit Count Array*. Otherwise, we allocate a new entry in the HHT (LRU entry of the corresponding set), set its *Valid* bit, make it MRU, update its *Tag*, and finally clear the *Hit Count Array* and push the *Current Hit Counter* to the front of *Hit Count Array*.

3.3 Selecting Victim

RRIP, which is the baseline replacement in our proposal, associates a number to each cache block named Re-Reference Prediction Value (RRPV), and replaces a block with the highest RRPV. EHC updates the victim selection mechanism based on "how many further hits do we except to receive from each cache block?". For each candidate, we take an average of its corresponding counters in *Hit Count Array* of HHT and name it 'E' (If an entry not found in HHT, 'E' is two to indicate two re-references for the block). This parameter shows the hit count we expect for that block in its current residency. Each block also has its own *Current Hit Counter* in the cache, which indicates the number of hits that the block already experienced since the entrance to the cache. '$E - CurrentHitCounter$' indicates how many further hits we expect to have for the block. We examine the value of '$E - CurrentHitCounter - RRPV$' for all candidates (including the incoming block for enabling bypassing) and evict (or bypass) the block with the lowest value.

4 METHODOLOGY

We evaluate our proposal using the simulation framework released by the second cache replacement championship (CRC-2) [5]. Table 1 summarizes the key elements of our methodology. We target both single-core and four-core processors with a 2 MB per-core shared LLC. The processors benefit from non-inclusive cache hierarchies that employ LRU as the baseline replacement policy. We include a variety of workloads listed in Table 2 from SPEC CPU2006 [4]. For each benchmark, we execute 4-billion instructions per core and use half of the instructions for warm up and the rest for performance measurement.

We compare our proposal against the following state-of-the-art replacement policies.

Dynamic RRIP (DRRIP) [8]. Each block stores 3 bits indicating RRPV. Upon each hit, RRPV of the block is set to zero and upon each miss, the block with the maximum RRPV is evicted. If none of the blocks have the maximum RRPV, the RRPV of all blocks is

Table 1: Evaluation Parameters.

Parameter	Value
Processing Nodes	6-stage pipeline, 256-entry ROB
L1-D/I Caches	32 KB, 8-way, 4-cycle load-to-use
Private L2 Cache	256 KB, 8-way, 8-cycle access latency
Shared LLC	2 MB per core, 16-way, 20-cycle hit latency
Config-1	Single-core, no data prefetcher
Config-2	Single-core, with data prefetcher
Config-3	Four-core, no data prefetcher
Config-4	Four-core, with data prefetcher
Data Prefetcher	L1 next-line prefetcher L2 PC-based stride prefetcher

Table 2: Simulated Workloads.

Name	Benchmarks
Single-Core	bwaves, bzip, cactusADM lbm, libquantum, mcf, xalan
Mix1	bwaves, libquantum, mcf, xalan
Mix2	bzip, lbm, libquantum, mcf
Mix3	bzip, lbm, mcf, xalan

incremented. This procedure is repeated until at least one block gets the maximum RRPV. It uses set-dueling for choosing an insertion policy between SRRIP and BRRIP. In SRRIP, all blocks are inserted with RRPV of maximum minus one. In BRRIP, the RRPV of inserted block gets the value of maximum minus one with the probability of $\frac{1}{32}$ and gets the value of maximum with the probability of $\frac{31}{32}$. Thirty two random sets emulate SRRIP and another 32 random sets emulate BRRIP. Remaining sets follow the winner of the duel. The total area overhead of DRRIP is 12 KB/48 KB in a single-core/four-core substrate.

SHiP [15]. A predictor is employed on top of RRIP [8] to identify whether the incoming block is dead-on-arrival or not. Blocks that are predicted to be dead-on-arrival are inserted with maximum RRPV, and the other blocks get the value of maximum minus one. To identify dead blocks, signatures are created based on the PC of the demanding instructions. The total area overhead is 17.75 KB/53 KB in a single-core/four-core substrate.

EVA [2]. Each block stores 7 bits indicating the age of the block and 1 bit symbolizing whether the block is reused or not. Each set has a 4-bit counter for per-set aging. Upon each hit or eviction, the corresponding counter of the block increases. Every 256 K accesses, EVA trains the metadata through a software update process, which stores the gathered information in a priority array. The priority array is used for calculating the EVA of each block, which is used in replacement decisions. The total area overhead is 34.5 KB/133.25 KB in a single-core/four-core substrate.

EHC. Implemented on top of DRRIP. HHT has 2 K entries per core, where each entry has one *Valid* bit, 20 bits for the *tag*, two 3-bit hit counters in *Hit Count Array*, and 4 bits for *LRU Recency*. The total area of HHT is 7.75 KB. As the tag of each block extended by 6 bits (current hit count and RRPV), EHC imposes extra 24 KB storage overhead. The total area is 31.75 KB/127 KB in a single-core/four-core substrate.

5 EVALUATION RESULTS

We run trace-based simulations for Miss-Per-Kilo-Instruction (MPKI) studies and detailed cycle-accurate full-system timing simulations for performance experiments.

We do not account for the overhead of the software update process (some tens of kilo cycles [2]). Therefore, the results of EVA are strictly better than the practical design.

5.1 Trace-Based Evaluation

Figure 3 presents the MPKI reduction of various policies over the baseline LRU. As shown clearly, our proposal outperforms other policies. On average, our proposal reduces the MPKI by 11%. The second best policy is EVA, which offers 9.2% MPKI reduction.

Although SHiP is efficient in many workloads, its effectiveness is significantly less than that of EHC. SHiP speculatively classifies the incoming blocks into two categories (binary classification), which makes the method more sensitive to the accuracy of the predictor. Whenever the predictor wrongly foretells the outcome, a large overhead is highly probable due to an extra off-chip miss. EHC, on the other hand, exhibits less sensitivity to the accuracy of the predictor, as it does not employ a binary classification. Moreover, in cases where RRPVs of multiple blocks saturate, SHiP randomly selects a victim, which may not be a good one. EHC, on the other hand, always seeks to evict a block with the lowest likelihood of liveliness.

EVA outperforms SHiP in five of seven workloads, but stands behind EHC. Moreover, as hardware-only implementation of EVA requires tremendous area overhead, EVA performs most of its operations by an OS runtime. Unfortunately, rapid changes in the datasets of workloads [14] make such software-based approaches inefficient.

5.2 Cycle-Accurate Evaluation

Figure 4 compares the performance improvement of the evaluated policies over the baseline LRU when data prefetchers are turned off. We use the Instruction per Clock (IPC) as the metric for performance. As shown, EHC offers the highest performance improvement among the evaluated policies for both single-core and multi-core processors. The performance improvement of our proposal ranges from -1.1% in *bzip* to 10.7% in *mcf*. The average performance improvement is 3.5% (12.9%) across all workloads on a single-core (multi-core) processor. The second best policy for a single-core/multi-core processor is EVA with an average performance improvement of 1.3%/8.6%.

Figure 5 compares the performance improvement of the evaluated policies over the baseline LRU when data prefetchers are turned on. The results are similar to those with no data prefetchers in many cases. EHC offers the highest performance improvement for both single-core and multi-core processors. The average performance improvement is 2.7% (11.9%) across all workloads on a single-core (multi-core) processor. The second best policy for a single-core/multi-core processor is SHiP/EVA with an average performance improvement of 0.6%/8.3%. Unlike other polices, EHC offers good performance on both single-core and multi-core processors.

6 CONCLUSION

In this paper, we proposed a replacement policy for last-level caches to improve their effectiveness. We identified a strong correlation between the number of remaining hits of a cache block and the reciprocal of its reuse distance. Taking advantage of the strong correlation, the proposed policy takes into account the expected hit count of a block to make a mature replacement decision. We evaluated our proposal with 15.75 KB storage overhead over the baseline LRU in the CRC-2 framework. The evaluation showed 2.7% (3.5%) performance improvement over LRU with (without) data prefetchers.

REFERENCES

[1] Nathan Beckmann and Daniel Sanchez. 2016. Modeling Cache Performance Beyond LRU. In *Proceedings of the 22nd IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 225–236. https://doi.org/10.1109/HPCA.2016.7446067

[2] Nathan Beckmann and Daniel Sanchez. 2017. Maximizing Cache Performance Under Uncertainty. In *Proceedings of the 23rd IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 109–120. https://doi.org/10.1109/HPCA.2017.43

[3] Trishul M. Chilimbi. 2001. On the Stability of Temporal Data Reference Profiles. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 151–160. https://doi.org/10.1109/PACT.2001.953296

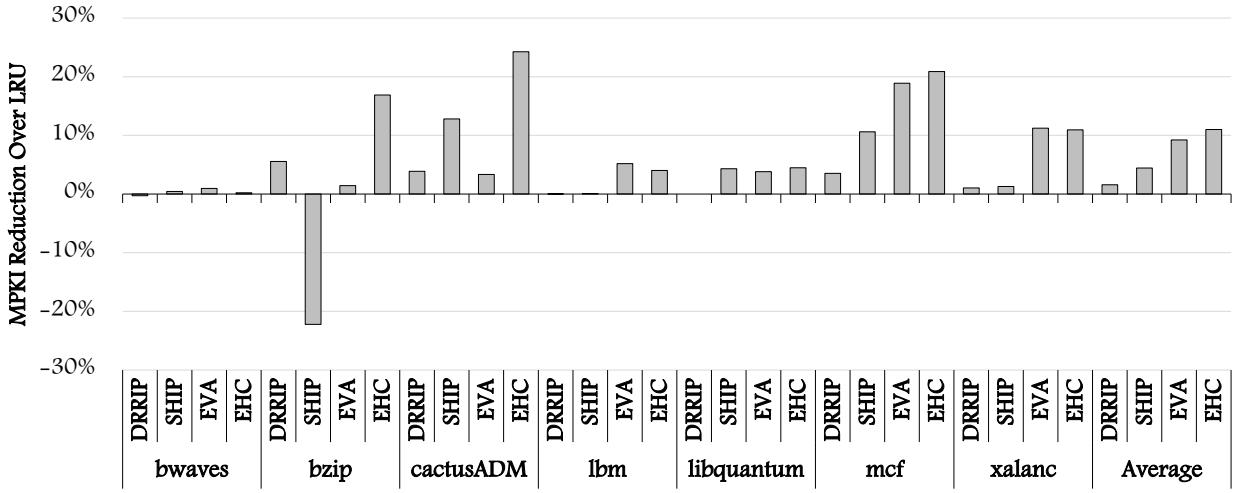

Figure 3: MPKI reduction of competing replacement policies over the baseline LRU.

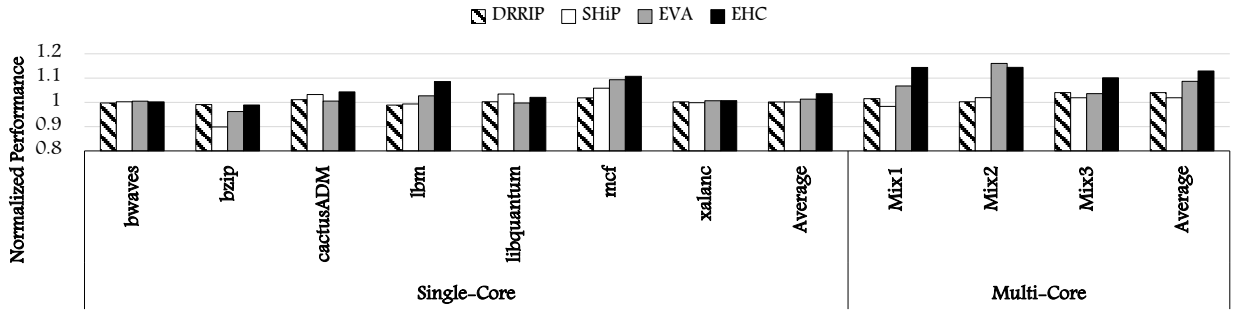

Figure 4: Performance improvement of competing replacement policies over baseline LRU. Data prefetchers are turned off.

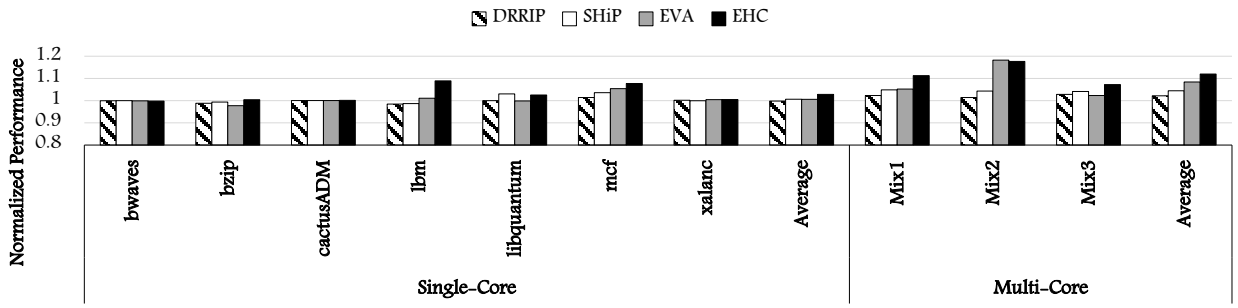

Figure 5: Performance improvement of competing replacement policies over baseline LRU. Data prefetchers are turned on.

[4] SPEC Standard Performance Evaluation Corporation. 2006. https://www.spec.org/cpu2006. (Aug. 2006).

[5] CRC-2. 2017. The 2nd Cache Replacement Championship. http://crc2.ece.tamu.edu/. (June 2017).

[6] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 389–400. https://doi.org/10.1109/MICRO.2012.43

[7] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. 2008. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 208–219. https://doi.org/10.1145/1454115.1454145

[8] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. 60–71. https://doi.org/10.1145/1815961.1815971

[9] Samira M. Khan, Daniel A. Jiménez, Doug Burger, and Babak Falsafi. 2010. Using Dead Blocks as a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 489–500. https://doi.org/10.1145/1854273.1854333

[10] An-Chow Lai and Babak Falsafi. 2000. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. 139–148. https://doi.org/10.1145/339647.339669

[11] An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-Block Prediction & Dead-Block Correlating Prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*. 144–154. https://doi.org/10.1145/379240.379259

[12] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 222–233. https://doi.org/10.1109/MICRO.2008.4771793

[13] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. 381–391. https://doi.org/10.1145/1250662.1250709

[14] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*. 252–263. https://doi.org/10.1109/ISCA.2006.38

[15] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. 2011. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 430–441. https://doi.org/10.1145/2155620.2155671