Cache Replacement Policy Based on Expected Hit Count

Armin Vakil-Ghahani, Sara Mahdizadeh-Shahri, Mohammad-Reza Lotfi-Namin, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, *Member, IEEE*, and Hamid Sarbazi-Azad

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 without commensurate area overhead. Our proposal offers 12.2 percent performance improvement over the baseline LRU in a multi-core processor and outperforms EVA, which is the state-of-the-art replacement policy.

Index Terms—Memory system, memory-intensive workload, last-level cache, replacement policy, Belady's MIN

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. Numerous studies pointed out the deficiencies of LRU for common access patterns [1], [2], [3], [4], [5].

- *A. Vakil-Ghahani, S. Mahdizadeh-Shahri, M.R. Lotfi-Namin, and M. Bakhshalipour are with the Department of Computer Engineering, Sharif University of Technology, Tehran 11155-11365, Iran.*
  E-mail: {vakil, smahdizadeh, mrlotfi, bakhshalipour}@ce.sharif.edu.
- *P. Lotfi-Kamran is with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran 19538-33511, Iran. E-mail: plotfi@ipm.ir.*
- *H. Sarbazi-Azad is with the Department of Computer Engineering, Sharif University of Technology, Tehran 11155-11365, Iran and also with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran 19538-33511, Iran. E-mail: azad@sharif.edu.*

Manuscript received 9 June 2017; revised 18 Aug. 2017; accepted 21 Sept. 2017. Date of publication 16 Oct. 2017; date of current version 19 Mar. 2018.
(Corresponding author: Mohammad Bakhshalipour.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2017.2762660

Such 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, practical replacement policies use some indicators to guess which piece of data will be referenced further in the future [2], [3], [4], [5], [6].

In this work, we make the observation that there is a strong correlation between the expected hit count of a cache block and the reciprocal of its reuse distance. 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.

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 (12.2) percent speedup on a single-core (multi-core) processor, and outperforms the state-of-the-art replacement policy.

2 Motivation

In this work, we make the observation that reuse distance of a cache block has a strong correlation to the reciprocal of its remaining number of hits in its current lifetime in the cache. With Belady's MIN replacement policy, it is intuitive that a block is evicted whenever its remaining hit count becomes zero (as the remaining hit count is determined by Belady's MIN). We show that the hit count of a cache block can be estimated using its past history. Fig. 1 shows the difference of the actual hit count of a block and its prediction when the replacement policy is Belady's MIN. For prediction, we use the average number of references that a block is experienced in its past four residencies in the cache. On average, the hit count of 69 percent of cache blocks can be estimated exactly. The prediction for most of the remaining blocks is also close to the actual hit count.

Not only the future hit count of a cache block can be estimated using the past history of the block, but also using past history of blocks that share most-significant bits of the block address (Corroborating prior work [2]). Fig. 2 shows the difference of the actual hit count of a block and its prediction when the replacement policy is Belady's MIN. For prediction, we use the average number of references that four most recent blocks with the same tag (i.e., most-significant bits of the address) experienced. On average, the hit count of 65 percent of cache blocks can be estimated exactly. The prediction accuracy is close to that of Fig. 1. We use this phenomenon to reduce the area overhead of the proposed replacement policy.

In this section, we showed that the expected hit count of a cache block can be estimated with high accuracy. In the evaluation section, we show that even a hit count that is estimated with a non-ideal replacement policy (e.g., DRRIP [4]) has correlation with the reciprocal of the reuse distance.

Many pieces of prior work [2], [4], [7], [8], [9], [10], [11], [12] enhanced replacement decisions by identifying and evicting/bypassing dead 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

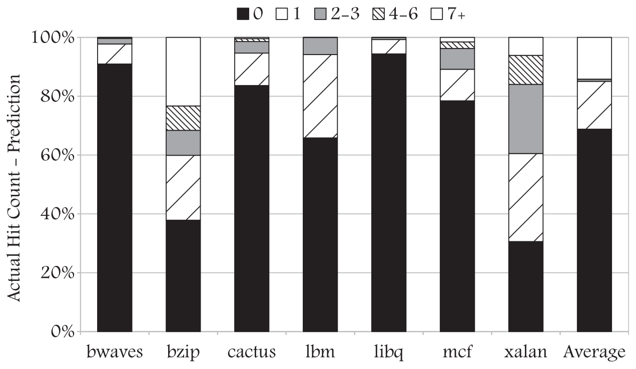

Fig. 1. The difference of the average of hit counts in the last four residencies of a block in the cache (i.e., prediction) and the actual hit count with Belady's MIN replacement policy.

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 replacement policy. Although, we can use any low-cost replacement algorithm as our baseline, in this paper, we use Re-reference Interval Prediction (RRIP) [4], as it has low storage overhead and offers good performance.

As motivated by Fig. 2, we use a simple hit-count predictor. The predictor relies on the repetitiveness of control flow and data accesses [13] to determine the *Expected Hit Count* of a cache block. Our predictor uses the average hit count of four most recent blocks that share 20 most-significant bits of the block address.

3.1 Design Overview

Our approach, named Expected Hit Count, is built on top of the baseline replacement policy and employs a metadata table, name Hit History Table (HHT), for storing the hit counts of four prior residencies of each tag (20 most-significant bits of the block address). Fig. 3 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 if the entry is valid.

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.

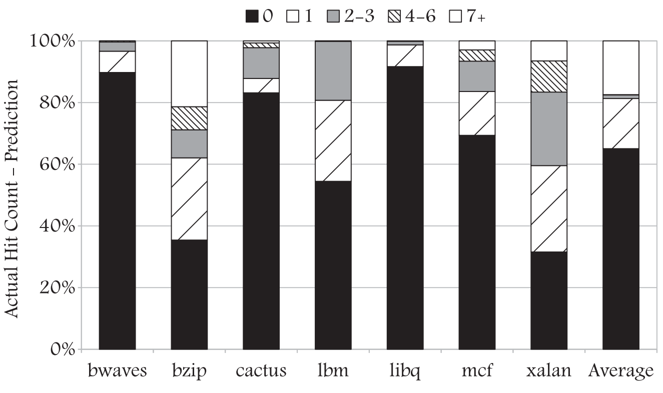

Fig. 2. The difference of the average hit count of last four blocks that share the same tag (i.e., most-significant bits of the address) and the actual hit count with Belady's MIN replacement policy.

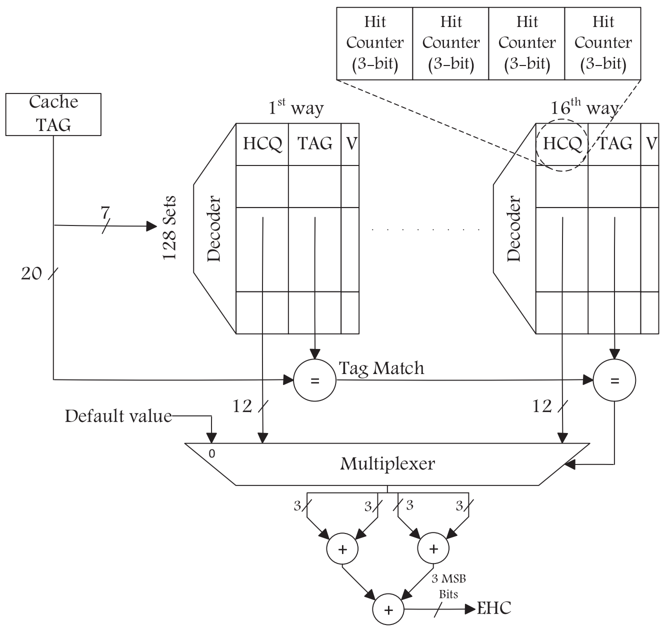

Fig. 3. The overview of the hit history table.

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 Queue. Each entry in the HHT is equipped with four 3-bit storage units each remembers the number of hits of a cache block in a specific residence of the block in the LLC. These storage units constitute a 'Hit Count Queue' and are managed as a FIFO queue.

EHC also requires extending the tag storage of each block in the LLC by two 3-bit storage units. One 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. The other storage unit, named *Expected Further Hits*, is a saturating count-down counter that stores the number of further hits that the block is expected to have in its current residency in the cache.

3.2 Updating Metadata

The HHT is updated either when *Current Hit Counter* saturates or when a block gets evicted from the cache. 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 Queue*. 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 Queue* and push the *Current Hit Counter* to the front of *Hit Count Queue*.

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?". Every time that a new block is brought into a set, its expected hit count is placed into *Expected Further Hits* counter. The expected hit count of a block is the average of the four storage units in the *Hit Count Queue* in case the tag is found in HHT or the default value, which is experimentally chosen to be one. For every access to this block, the content of *Expected Further Hits* is decremented if it is greater than zero. When a victim needs to be selected, we combine *Expected Further Hits*, which determines how many further hits we expect to receive for this block, and RRPV to choose a victim. We examine the value of '$ExpectedFurtherHits - RRPV$' for all candidates (including the

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
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	bzip, lbm, libquantum, mcf
Mix2	bwaves, libquantum, mcf, xalan
Mix3	bzip, cactusADM, libquantum, xalan
Mix4	bzip, lbm, libquantum, xalan
Mix5	bwaves, cactusADM, mcf, xalan
Mix6	bzip, lbm, mcf, xalan

incoming block for enabling bypassing) and evict (or bypass) the block with the lowest value (if more than one block has the lowest value, we pick the first in the set).

4 Methodology

We evaluate our proposal using the simulation framework released by the second cache replacement championship (CRC-2) [14]. 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 [15]. 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 evaluate the following replacement policies.

Dynamic RRIP (DRRIP) [4]. 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 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 single-core/four-core substrate.

EVA [16]. 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,[1] 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 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*, four 3-bit hit counters in *Hit Count Queue*, and 4 bits for *LRU Recency*. The total area of HHT is 9.25 KB. As the tag of each block is extended by 9 bits (current hit count, expected further hits, and RRPV), EHC imposes extra 36 KB storage overhead. The total area is 45.25 KB/181 KB in single-core/four-core substrate.

5 Evaluation Results

We run trace-based simulations for the correlation of reuse distance and hit count and Miss Per Kilo-Instruction (MPKI) studies and detailed cycle-accurate full-system timing simulations for performance experiments.

5.1 Trace-Based Evaluation

To show that the expected hit count with a non-ideal replacement policy has correlation with the reciprocal of the reuse distance, Fig. 4 compares the quality of the chosen replacement victims in DRRIP and EHC. Every time a victim needs to be selected, we sort the blocks in the set and the incoming block based on their reuse distance. *Block 0* has the longest and *Block 16* has the shortest reuse distance. Ideally, a replacement policy always picks *Block 0*.

Comparing EHC and DRRIP reveals that the victims chosen by EHC have higher quality as compared to those chosen by DRRIP. While on average, both EHC and DRRIP choose equal number of *Block 0*, when it comes to *Block 1* and *Block 2-8*, EHC is clearly the winner. As a result of having more *Block 1* and *Block 2-8*, the fraction of *Block 8-16*, which is undesirable, is significantly reduced in EHC. As EHC is DRRIP with the expected hit count, the difference between the two replacement policies shows the correlation of reuse distance and the reciprocal of the expected hit count.

Fig. 5 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 11.2 percent. The second best policy is EVA, which offers 6.5 percent MPKI reduction. 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 [17] make such software-based approaches inefficient.

5.2 Cycle-Accurate Evaluation

Fig. 6 compares the performance improvement of the evaluated policies over the baseline LRU. We use the Instruction per Cycle (IPC) as the metric for performance. EHC offers the highest performance improvement on both single- and multi-core processors.

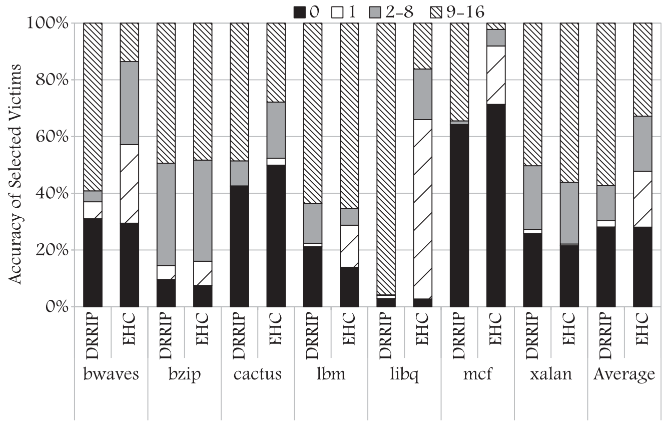

Fig. 4. The quality of chosen victims in DRRIP and EHC. Blocks in a set are sorted with the reuse distance: *Block 0* has the longest and *Block 16* has the shortest reuse distance. Selecting a victim with lower index is better.

1. We do not account for the overhead of this process (some tens of kilo-cycles [16]). Therefore, the results of EVA is strictly better than the practical design.

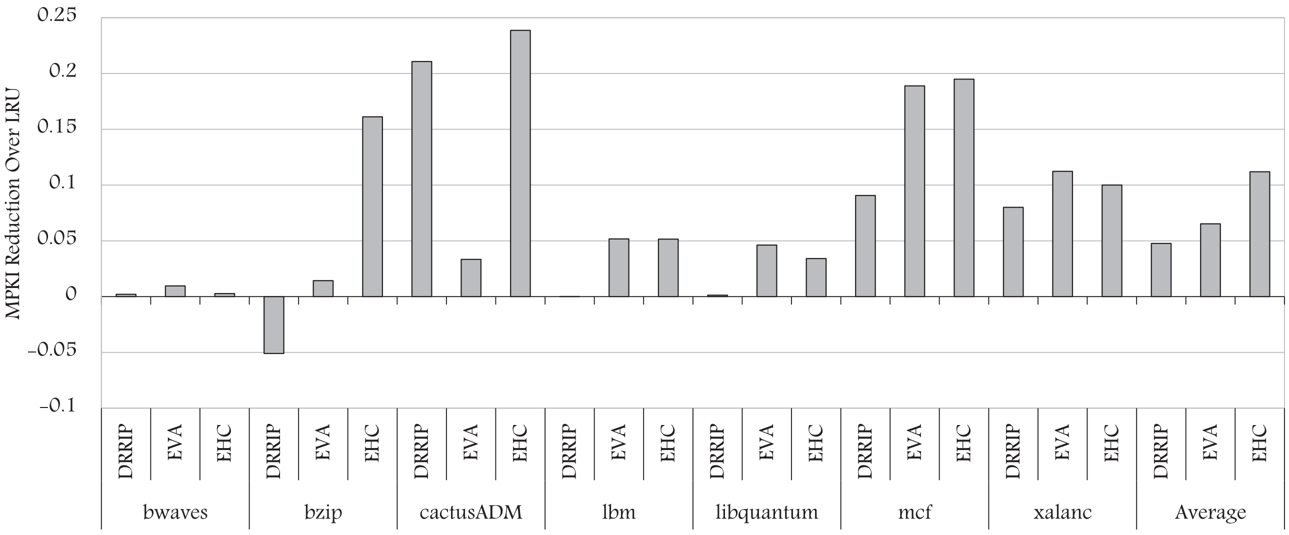

Fig. 5. MPKI reduction of competing replacement policies over the baseline LRU.

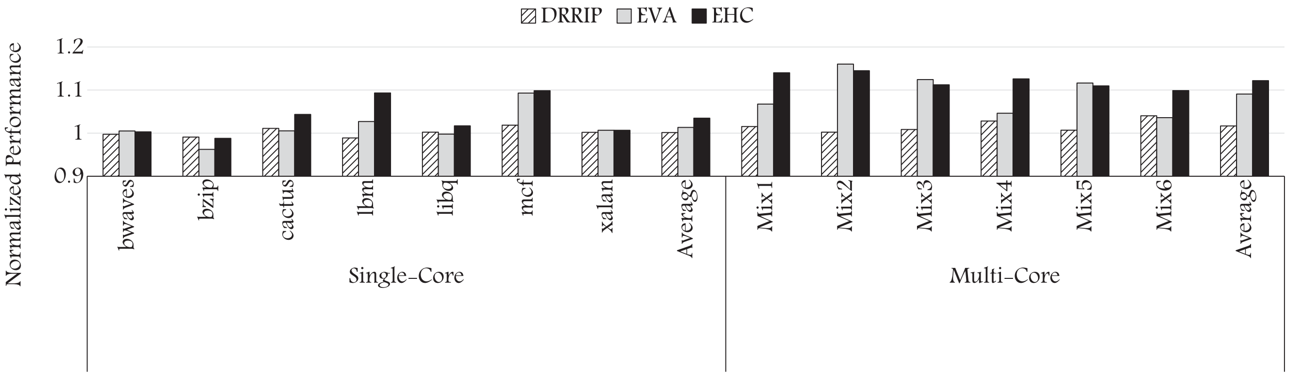

Fig. 6. Performance improvement of competing replacement policies over the baseline LRU.

The average performance improvement is 3.5 (12.2) percent across all workloads on a single-core (multi-core) processor. The second best policy is EVA with an average performance improvement of 1.3/9.1 percent on single-/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 correlation, the proposed policy takes into account the expected hit count of a block to make a mature replacement decision. The evaluation showed 12.2 percent performance improvement over LRU.

References

[1] N. Beckmann and D. Sanchez, "Modeling cache performance beyond LRU," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 225–236.

[2] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr., and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 430–441.

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

[4] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, 2010.

[5] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2012, pp. 389–400.

[6] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *Proc. 25th Int. Conf. Comput. Des.*, Oct. 2007, pp. 245–250.

[7] A.-C. Lai and B. Falsafi, *Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction*. New York, NY, USA: ACM, 2000, vol. 28, no. 2.

[8] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," *ACM SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 144–154, 2001.

[9] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi, "Using dead blocks as a virtual victim cache," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 489–500.

[10] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proc. 41st IEEE/ACM Int. Symp. Microarchitecture*, 2008, pp. 222–233.

[11] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 208–219.

[12] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 381–391, 2007.

[13] T. M. Chilimbi, "On the stability of temporal data reference profiles," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2001, pp. 151–160.

[14] The 2nd Cache Replacement Championship. (2017). [Online]. Available: http://crc2.ece.tamu.edu/

[15] S. S. P. E. Corporation. (2006). [Online]. Available: http://www.spec.org/

[16] N. Beckmann and D. Sanchez, "Maximizing cache performance under uncertainty," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 109–120.

[17] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proc. 33rd Annu. Int. Symp. Comput. Archit.*, Jun. 2006, pp. 252–263.