Last-Level Cache Insertion and Promotion Policy in the Presence of Aggressive Prefetching

Daniel A. Jiménez, *Fellow, IEEE*, Elvira Teran, *Member, IEEE*, and Paul V. Gratz, *Member, IEEE*

***Abstract*—The last-level cache (LLC) is the last chance for memory accesses from the processor to avoid the costly latency of going to main memory. LLC management has been the topic of intense research focusing on two main techniques: replacement and prefetching. However, these two ideas are often evaluated separately, with one being studied outside the context of the state-of-the-art in the other. We find that high-performance replacement and highly accurate pattern-based prefetching do not result in synergistic improvements in performance. The overhead of complex replacement policies is wasted in the presence of aggressive prefetchers. We find that a simple replacement policy with minimal overhead provides at least the same benefit as a state-of-the-art replacement policy in the presence of aggressive pattern-based prefetching. Our proposal is based on the idea of using a genetic algorithm to search the space of insertion and promotion policies that generalize transitions in the recency stack for the least-recently-used policy.**

***Index Terms*—Cache replacement, cache prefetching, last-level cache, genetic algorithms, machine learning for systems.**

I. Introduction

THE last-level cache (LLC) is the largest on-chip component of the memory hierarchy, keeping as many accesses as possible from going off-chip. Research in LLC management has focused on replacement and prefetching, but the interaction of these two techniques have not been studied adequately. Work on replacement tends not to use a state-of-the art prefetcher in its baseline, and work on prefetching is often done with a simple replacement policy. However, the choice of prefetcher can have a significant impact on the effectiveness of the replacement policy.

Fig. 1 shows the performance of various combinations of prefetchers and replacement policies on a simulated cache hierarchy over a wide range of benchmarks described in Section IV. The graph shows geometric mean speedup over a baseline least-recently-used (LRU) policy managing the LLC. Replacement policies modeled are multiperspective reuse prediction [1], signature-based hit prediction [2], and Glider [3].

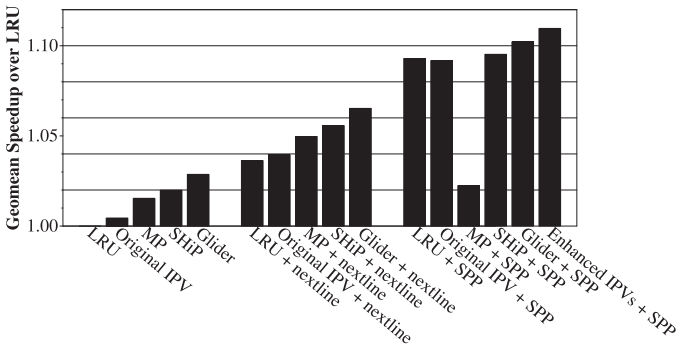

Fig. 1. Performance of combinations of replacement and prefetchers.

Prefetchers modeled are none, next-line, and signature-path-prefetcher (SPP) [4]. Without prefetching, modern replacement policies improve performance. For example, Glider achieves 2.9% improvement. With a next-line prefetcher, Glider achieves 2.8% improvement. However, with the complex pattern-based prefetcher SPP, the performance improvement of all cache replacement policies is negligible at best, and drops significantly for multiperspective. Glider+SPP yields 10.2% speedup over LRU alone, but only 0.8% speedup over LRU+SPP, quite a disappointment compared to the potential 2.9% improvement without prefetching. The technique we introduce in this paper using insertion and promotion vectors (IPVs) yields a 11.0% speedup over LRU alone, or a 1.5% improvement over LRU+SPP, outperforming previous policies with significantly lower hardware complexity.

Prefetchers and replacement policies should be developed together. We propose a novel and low-cost replacement policy based on *insertion and promotion vectors*(IPVs) [5], [6]. IPVs are developed in the presence of a high-performance prefetcher. The policies can adapt to the prefetcher's behavior as well as that of the current workload. The idea is that insertion and promotion is governed by two IPVs: one for demand accesses and one for prefetch accesses. The IPVs used for the current workload will change based on set-dueling [7] a small number of pre-developed IPVs to find the best one at the moment. The set of IPVs themselves are developed by a genetic algorithm running over hundreds of training workloads.

II. Background and Related Work

A great deal of work has focused on cache replacement and prefetching. Cache replacement work has focused on reuse prediction [2], [8], [9]. More recent work has used complex

Manuscript received 20 December 2022; revised 31 January 2023; accepted 1 February 2023. Date of publication 3 February 2023; date of current version 17 February 2023. This work was supported by the National Science Foundation under Grants CNS-1938064 and CCF-1912617, and Semiconductor Research Corporation project under Grant GRC 2936.001. *(Corresponding author: Daniel A. Jiménez.)*

Daniel A. Jiménez is with the Department of Computer Science & Engineering, Texas A&M University, College Station, TX 77843 USA (e-mail: djimenez@acm.org).

Elvira Teran is with the School of Engineering, Texas A&M International University, Laredo, TX 78041 USA (e-mail: elvira.teran@tamiu.edu).

Paul V. Gratz is with the Department of Electrical & Computer Engineering, Texas A&M University, College Station, TX 77843 USA (e-mail: pgratz@gratz1.com).

Digital Object Identifier 10.1109/LCA.2023.3242178

algorithms based on machine learning [1], [3], [10], or emulating Bélády's MIN algorithm [11], [12]. These approaches have grown increasingly complex and rely on front-end program features such as the program counter to make predictions. Work on prefetching has been active, with many recently proposed complex designs [4], [13], [14], [15]. At least one of these designs, SPP, has been adopted in hardware [16] so we focus on it as an example of a feasible complex pattern-based prefetcher. More recent work has taken into account the interaction of the replacement policy and the prefetcher [12], [17] but co-designing the replacement policy and prefetcher while using program features typically available to microarchitectural designers remains a poorly explored area; we know of only one such proposal [18].

III. Demand and Prefetch Insertion and Promotion Vectors

A. Insertion and Promotion Vectors

Replacement policies such as LRU and RRIP associate a *position* with each block in a set. For an n-way set-associative cache, the LRU policy assigns a position from $0..n-1$ to each block, ordering the blocks from most recently used in position 0 to least recently used in position $n-1$.

When a block is first inserted into a set, it is placed in an initial position. An incoming block is placed into position 0, the most-recently-used (MRU) position. A subsequent hit to a block will promote it to position 0.

Previous work has noted that these choices for insertion and promotion positions can be improved. *Dynamic Insertion Policy* (DIP) [7] adjusts the insertion position for LRU to 0 or $n-1$ based on which of these policies is working best for a small group of sampled sets, an adaptive technique called *set dueling* [7]. *Genetic Insertion and Promotion for PseudoLRU* (GIPPR) [19] and follow-up work [6] use a genetic algorithm to explore the space of possible insertion and promotion positions for a PseudoLRU replacement policy. These policies use an *insertion and promotion vector* (IPV) of $n+1$ elements describing the insertion and promotion policy. An IPV is a vector $V[n+1] = [p_0, p_1, ..., p_{n-1}, p_n]$. On a miss, the incoming block is placed in position p_n. On a hit, if the matching block is in position i, it is promoted to position p_i and blocks previously at position i or higher shift toward the LRU position to accommodate the promoted block. This idea can be applied to LRU- or RRIP-based replacement policies. In this paper we explore LRU-based policies.

B. Extending IPVs

We extend the idea of IPVs in two ways:

1) Demand and Prefetch Accesses: First, instead of one IPV, we develop two: one for demand accesses, and another for prefetch accesses. We develop a pair of IPVs $V[2][n+1] = [d_0, d_1, ..., d_{n-1}, d_n][p_0, p_1, ..., p_{n-1}, p_n]$ such that on a demand hit to recency position i, the accessed block is placed in position $V[0][i]$, while on a prefetch hit to recency position i, the touched block is promoted to position $V[1][i]$. On a demand miss, the new block is placed in recency position $V[0][n]$, while a missed prefetch would be placed in position $V[1][n]$. There are $(n \times n!)^2$ such possible pairs of IPVs making exhaustive search of the space of IPVs impossible for reasonable associativities, so we find good candidates using the genetic algorithm on a training set of traces.

TABLE I
ChampSim Simulation Parameters. The L1 dCache and L2 Cache Implement Next-Line Prefetching

L1 icache	32KB, 8-way, 4-cycle, 8-entry MSHR
L1 dcache	32KB, 8-way, 4-cycle, 8-entry MSHR
L2 cache	512KB, 8-way, 10-cycle, 16-entry MSHR
LLC	2MB per core, 16-way, 20-cycle, 64-entry MSHR
DRAM	4GB, DDR4, 4GHz, 3200 MT/s
Branch Pred	Hashed Perceptron

2) Set-Dueling IPVs: Different workloads exhibit different behaviors. There is no best policy, so in order to adapt to the running workload, we develop a handful of IPVs and use set-dueling to choose the best one [7]. Set-dueling chooses between two cache management policies using two groups of *leader sets*, , *i.e.,* small subsets of the cache dedicated to each policy. Set-dueling has been extended to multiple policies [20]. In this work, we extend set-dueling to m IPVs by keeping m saturating counters initialized to zero, one for each group of leader sets tied to a particular IPV. When the i^{th} group experiences a miss, the i^{th} counter is incremented. When a counter saturates, all counters are halved. The rest of the cache sets follow the IPV with the minimum counter.

C. Contrast With Previous Work

In contrast to previous work, our proposed technique uses minimal extra hardware over the baseline replacement policy For m IPVs, we require only m 12 bit counters to implement multi-way set-dueling and a tiny amount of SRAM, 136 bytes in our implementation, to encode the learned IPVs. Thus, the total storage required for our idea is 148 bytes. The additional logic is straightforward and confined to the LLC. Previous proposals require complex predictors with many kilobytes of tables and other bookkeeping structures as well as channels to communicate program features not typically available to LLC replacement policy logic, such as the address of the instruction causing a hit or miss.

IV. Methodology

This section describes our methodology for evaluating the idea. We describe the simulator, the workloads, and the training methodology for the genetic algorithm.

A. Microarchitectural Simulator

We devised an infrastructure built on ChampSim, a commonly used microarchitectural simulator with a robust modeling of a three-level cache hierarchy and out-of-order execution. Table I shows the microarchitectural parameters chosen for ChampSim.

ChampSim comes with code for simulating the LRU and SHiP replacement policies as well as the next line and SPP prefetchers. We obtained the source code for multiperspective reuse prediction and Glider from the respective authors. We implemented insertion and promotion vectors for demand and prefetch accesses as well as set-dueling between IPVs.

B. Benchmarks

For this work, we required workloads for training the IPVs and separate workloads for reporting results. For the training workloads, we chose the traces made available by Qualcomm for the First Championship Value Prediction contest [21]. The workloads are divided into broad categories such as *compute_fp*, *crypto*, *compute_int*, and *srv*. We chose a subset that has a least 1 miss per thousand instructions (MPKI) in the LLC. The workloads were converted to ChampSim format. The Qualcomm traces are generated from Arm64 compiled binaries.

For the testing workloads, we chose a subset of single-threaded SPEC CPU 2006 and SPEC CPU 2017 simpoints [22] with at least 1 MPKI in the LLC, as well as benchmarks from the GAP benchmark suite [23] and XSBench benchmarks [24] with at least 1 MPKI in the LLC. Each simpoint contains 1.5 billion instructions allowing ample time for warming cache structures. These traces are generated from x86_64 compiled binaries.

C. Genetic Algorithm

To find good sets of IPVs, we use a genetic algorithm. The algorithm starts with a population of random vectors, evaluating each one using a fitness function. In subsequent generations, members of the population are replaced with new vectors obtained by combining vectors that yielded the best fitness from the previous generation as well as occasional mutations numerically perturbing the vectors. After many generations, the population converges. At this point, we use the best vector to measure the optimization.

In the original IPV work, authors used a genetic algorithm to find the best IPV. The fitness function was an estimate of the speedup over LRU given a simple cache model based on demand misses [5], [6] allowing the fitness function to be evaluated quickly compared to a full cycle-accurate simulator. This approach fails in our context because the impact of prefetching is not easily modeled with a simple simulator. Thus, we use the full cycle-accurate simulator, increasing the runtime by two orders of magnitude. We use an alternate method: implementing the genetic algorithm directly into the simulator, allowing vectors to evolve over the run-time of the training benchmark. Each training workload runs one billion instructions evolving a population of 64 IPVs, each sampling $\frac{1}{64}$th of the cache. At the end of each run, the best IPV is collected into a database of IPVs for the different training workloads. A single generation monitors 10 million memory accesses and the typical workload goes through about 100 generations.

D. Finding a Good Set of IPVs

We evaluate each training workload on each IPV to get the speedup obtained by using only that IPV. Then we try many combinations of IPVs to find a good set of IPVs to duel. To find the best combination of n IPVs, we choose all combinations of n IPVs and compute the geometric mean of the minimum speedup delivered for each benchmark by one of those n IPVs, thus estimating the speedup that would be obtained if set-dueling always chose the best of the n vectors for each workload. This selection process is computationally intensive, so rather than exhaustive search, we use a large number of randomly sampled subsets of IPVs and improve the best subset through greedy search through the other IPVs. We developed 8 sets of demand and prefetch IPVs for single-core workloads.

TABLE II
GOOD IPVS FOR SINGLE-CORE WORKLOADS

Demand Vector	Prefetch Vector
[0 0 0 0 1 3 4 3 0 2 3 0 7 3 1 15 0]	[0 0 1 1 1 4 3 0 4 0 3 5 4 10 12 15 15]
[0 0 0 0 1 2 3 2 4 0 2 4 8 0 1 15 14]	[0 0 0 0 1 4 2 4 5 5 10 3 12 13 7 15 15]
[0 0 1 3 0 0 5 0 6 3 0 6 6 12 8 7 10]	[0 0 1 2 1 0 2 0 2 0 1 10 11 13 10 10 13]
[0 0 0 0 1 0 0 6 0 5 0 9 6 5 1 14 15]	[0 0 0 0 1 5 3 5 7 2 0 9 0 4 1 15 15]
[0 0 0 0 2 2 3 0 1 0 7 5 6 4 2 11 12]	[0 0 0 0 0 2 2 0 1 3 5 2 5 3 2 15 15]
[0 0 2 0 3 5 3 0 6 9 7 3 6 4 4 10 13]	[0 0 0 1 1 2 2 1 2 0 0 9 10 12 1 13 15]
[0 0 0 1 1 4 4 5 0 2 2 5 7 5 1 15 15]	[0 0 0 0 4 0 1 6 6 7 4 11 10 10 2 5 11]
[0 0 2 3 1 0 5 6 3 7 2 5 0 9 4 7 2]	[0 0 0 1 4 2 1 5 3 4 1 7 1 5 9 15 0]

E. Multi-Programmed Workloads

We also simulate multi-programmed workloads. We simulate 4 cores, each running a different workload. We choose 50 mixes of 4 randomly-chosen workloads from the single-core training workloads for training, and 50 mixes of 4 randomly-chosen workloads from the testing workloads for testing, developing another 8 sets of demand and prefetch IPVs. For both training and testing we allowed the simulations to run for one billion instructions.

V. RESULTS

In this section we document the simulated performance and miss rates from our technique, and illustrate a good set of IPVs found.

A. Good IPVs

We write IPVs as pairs of vectors giving the insertion and promotion policies for demand and prefetch accesses. Elements $i \in \{0..n-1\}$ of a vector give the position that a block in recency position i should be promoted, while element n gives the insertion position of an incoming block. For single-core workloads, the best set of IPVs is given in Table II. Most vectors tend to place demand and prefetch insertions close to the LRU position. However, for example the first vector places incoming demand misses into the MRU position while placing incoming prefetch misses into the LRU position. The last vector places demand misses close to MRU and prefetch misses into MRU. Most of the vectors keep blocks in the LRU position near LRU, but once they reach the middle of the recency stack they are quickly promoted close to MRU. Space does not allow a full analysis of the nature of these vectors. They could either be deployed in a replacement policy as is, or further studied to find new insights into insertion and promotion policies distinguishing between demand and prefetch accesses.

B. Performance

Fig. 1 in the Introduction shows the geometric mean speedup of combinations of replacement policies and prefetchers, including the speedup for our technique of using demand and prefetch IPVs together with SPP. Fig. 2 shows S-curves of the performance of the various replacement policies with SPP

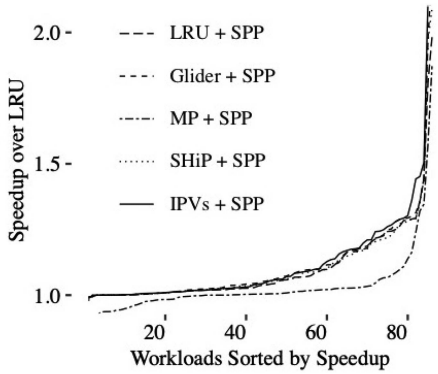

Fig. 2. Speedups of various replacement policies with SPP.

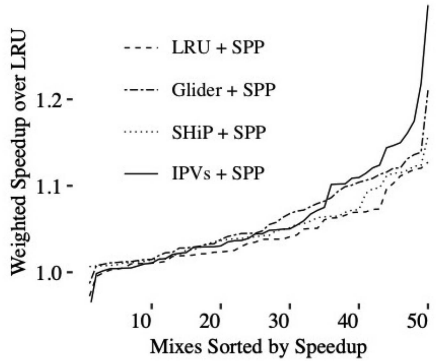

Fig. 3. Speedups of various replacement policies with SPP and multi-core workloads.

over the 86 workloads tested. IPVs+SPP outperforms the other techniques for the majority of workloads. We also evaluated the original IPV used in previous work [19]. It showed negligible performance improvement over LRU.

Fig. 3 shows weighted speedups for multi-programmed workloads. Again, IPVs+SPP outperforms the other techniques for most workloads, achieving a weighted speedup of 6.2% over LRU, compared with 6.0% for Glider+SPP and 5.1% for SHiP+SPP.

VI. Conclusion

We have shown that dueling genetically evolved demand and prefetch insertion and promotion vectors yields a speedup exceeding that of complex state-of-the-art replacement policies. Our proposal requires minimal extra storage over the LRU policy to represent set-dueling counters and IPVs, compared to many kilobytes and new communications channels from the front-end to the LLC required by previous work. We believe this practical proposal could quickly be adopted by industry.

Acknowledgments

Generous gifts from Intel. Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

References

[1] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proc. 50th Int. Symp. Microarchit.*, 2017, pp. 436–448.
[2] C.-J. Wu et al., "SHiP: Signature-based hit predictor for high performance caching," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 430–441.
[3] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, pp. 413–425. [Online]. Available: https://doi.org/10.1145/3352460.3358319
[4] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence-based lookahead prefetching," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–12.
[5] D. A. Jiménez, "Insertion and promotion for tree-based PseudoLRU last-level caches," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, ser. MICRO-46, 2013, pp. 284–296.
[6] E. Teran, Y. Tian, Z. Wang, and D. A. Jiménez, "Minimal disturbance placement and promotion," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 201–211.
[7] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 381–391. [Online]. Available: https://doi.org/10.1145/1250662.1250709
[8] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," *SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 144–154, 2001.
[9] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 175–186.
[10] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, IEEE, 2016, pp. 1–12.
[11] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in *Proc. 43rd ACM/IEEE Int. Symp. Comput. Archit.*, ser. ISCA '16, 2016, pp. 78–89.
[12] I. Shah, A. Jain, and C. Lin, "Effective mimicry of Belady's MIN policy," in *Proc. IEEE Int. Symp. High-Perform. Comput. Architect.*, 2022, pp. 558–572.
[13] P. Michaud, "Best-offset hardware prefetching," in *Proc. 22nd IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 469–480.
[14] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 118–131.
[15] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 83–95.
[16] B. Grayson et al., "Evolution of the Samsung Exynos CPU microarchitecture," in *Proc. 47th ACM/IEEE Annu. Int. Symp. Comput. Archit.*, Valencia, Spain, 2020, pp. 40–51.
[17] A. Jain and C. Lin, "Rethinking Belady's algorithm to accommodate prefetching," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 110–123.
[18] 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 *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 737–749.
[19] D. A. Jiménez, "Insertion and promotion for tree-based PseudoLRU last-level caches," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, ser. MICRO-46, New York, NY, USA, 2013, pp. 284–296.
[20] S. M. Khan and D. A. Jiménez, "Insertion policy selection using decision tree analysis," in *Proc. IEEE 28th Int. Conf. Comput. Des.*, 2010, pp. 106–111.
[21] Championship Value Prediction (CVP). [Online]. Available: https://www.microarch.org/cvp1/
[22] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 318–319, 2003.
[23] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," 2015, *arXiv:1508.03619*.
[24] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *Proc. Conf. Role React. Phys. Toward Sustain. Future*, Kyoto, 2014, pp. 1–12. [Online]. Available: https://www.mcs.anl.gov/papers/P5064--0114.pdf