Hawkeye: Leveraging Belady's Algorithm for Improved Cache Replacement

Akanksha Jain Calvin Lin
Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712, USA
{akanksha, lin}@cs.utexas.edu

ABSTRACT

This paper evaluates the Hawkeye cache replacement policy on the Cache Replacement Championship framework. The solution departs from that of the original paper by distinguishing prefetches from demand fetches, so that redundant prefetches can be identified and cached appropriately.

Evaluation on SPEC2006 shows that in the absence of prefetching, Hawkeye provides a speedup of 4.5% over LRU (vs. 3.4% for SHiP) on the single-core configuration and a speedup of 9.2% (vs. 7.1% for SHiP) on the four-core configuration. In the presence of prefetching, Hawkeye's performance is marginally better than SHiP's (2.25% speedup vs 2.09% speedup over LRU).

Keywords: Cache replacement, Belady's Algorithm

1. INTRODUCTION

The Cache Replacement Championship (CRC) evaluates replacement policies for configurations that include both single and multi-core settings, each with and without a prefetcher. Our broad strategy for all configurations is based on the Hawkeye cache replacement policy [2], which learns from Belady's optimal solution for past references to predict the caching behavior of future references.

In the absence of prefetching, Hawkeye performs well for both single-core (4.5% speedup over LRU) and multi-core configurations (9.2% speedup over LRU), outperforming the previous state-of-the-art replacement policy [8].

However, in the presence of prefetching, Hawkeye performs poorly on regular workloads. As a result, on the CRC framework, Hawkeye's performance is 1% worse than LRU's. The problem is that if Hawkeye were to treat demand and prefetch requests identically, it would cache redundant prefetches, even though they do not increase the number of cache hits. On the other hand, if Hawkeye were to ignore prefetch requests, then Hawkeye would be unable to distinguish useful prefetches from useless prefetches.

To accommodate prefetch requests, we present simple changes to Hawkeye. Our solution ignores redundant prefetches by only considering prefetch requests that are subsequently followed by a demand request. Furthermore, to distinguish the caching behavior of demands and prefetches by the same load instruction, we use separate predictors for demand and prefetch requests. Thus, our solution can learn, for example, that a given instruction tends to load cache-friendly demand accesses but inaccurate prefetches.

With these changes, Hawkeye performs marginally better than SHiP in the presence of prefetches, as both policies see roughly 2% speedup over LRU. We find that both policies benefit from identifying and evicting inaccurate prefetches. In the presence of prefetches, Hawkeye's more accurate prediction for demand accesses is diminished because poor replacement decisions are easily compensated by an aggressive prefetcher.

This paper focuses on (1) the additions that we have introduced to accommodate prefetching for this competition and (2) an explanation of the hardware requirements.

Section 2 discusses the main ideas behind the Hawkeye policy, and Section 3 describes key implementation details. Section 4 evaluates our solution. We conclude in Section 5.

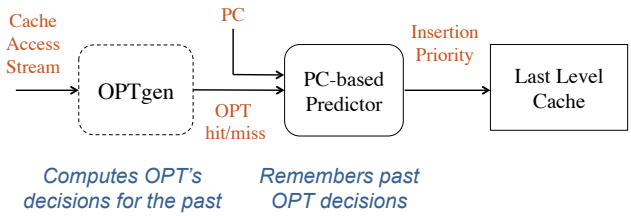

Figure 1: Block diagram of the Hawkeye replacement algorithm.

2. BACKGROUND

This section summarizes the original Hawkeye solution [2].

Hawkeye reconstructs Belady's optimal solution for past accesses and learns this optimal solution to predict the caching behavior of future accesses. To compute the optimal solution for past accesses, Hawkeye uses the OPTgen algorithm [2], and to learn OPTgen's solution, Hawkeye uses a PC-based predictor that learns whether load instructions tend to load *cache-friendly* or *cache-averse* lines. Lines that are

predicted to be cache-friendly are inserted with high priority in the cache, while lines that are predicted to be cache-averse are inserted with low priority.

An accurate reconstruction of Belady's optimal solution requires a long history of past references (8× the size of the cache). To reduce Hawkeye's area overhead, we apply OPTgen to 64 sampled sets only [6], which results in onlyl a 12KB overhead for a 2MB cache and is sufficient for distinguishing cache-friendly load instructions from cache-averse load instructions.

Figure 1 shows the overall structure of Hawkeye. Its main components are the Hawkeye Predictor, which makes eviction decisions, and OPTgen, which simulates OPT's behavior to produce inputs that train the Hawkeye Predictor. The system also includes a Sampler Cache, which stores a long history of past references.

2.1 OPTgen

OPTgen determines what would have been cached if the OPT policy had been used. One of the key insights behind OPTgen is that for any given cache access to X, this determination can be made accurately when X is next reused. This is because any later reference will be farther into the future, so Belady's algorithm will favor X over that other line. Thus, OPTgen computes the optimal solution by assigning cache capacity to lines in the order that they are reused.

To understand how OPTgen assigns cache capacity, we first define a *usage interval* for a reference to X to be the time period that starts with a reference to X and proceeds up to (but not including) its next reference X'. When a line is reused, OPTgen determines that it would be a cache hit with the OPT solution if there is enough space in the cache over the duration of the line's usage interval.

For example, consider the sequence of accesses in Figure 2, which includes X's usage interval. Here, the cache capacity is two. We assume that OPTgen has already determined the A, B, and C can be cached with OPT, and since these intervals never overlap, the maximum number of overlapping liveness intervals in X's usage interval never reaches the cache capacity; thus there is space for line X to reside in the cache, and OPTgen infers that X' would be a hit.

OPTgen can be implemented efficiently in hardware using a simple vector. More details about OPTgen's implementation are in the original paper [2].

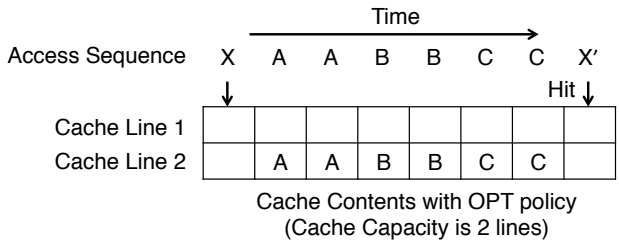

Figure 2: Intuition behind OPTgen.

2.2 The Hawkeye Predictor

The Hawkeye Predictor learns the behavior of the OPT policy on past memory references: If OPTgen determines that a line would be a cache hit under the OPT policy, then the PC that last accessed the line is trained positively; otherwise, the PC that last accessed the line is trained negatively. The Hawkeye Predictor has 2K entries, it uses 5-bit counters for training, and it is indexed by a 11-bit hashed PC.

2.3 Cache Replacement

On every cache access, the Hawkeye Predictor generates a prediction to indicate whether the line is likely to be cache-friendly or cache-averse. Cache-friendly lines are inserted with an RRIP value of 0, and cache-averse lines are inserted with an RRIP value of 7. When a cache-friendly line is inserted in the cache, the RRIP counters of all other cache-friendly lines are aged.

On a cache replacement, any line with an RRIP value of 7 (cache-averse line) is chosen as an eviction candidate. If no line has an RRIP value of 7, then Hawkeye evicts the line with the highest RRIP value (oldest cache-friendly line) and detrains its corresponding load instruction if the evicted line is present in the sampler.

3. PREFETCH-AWARE HAWKEYE

To accommodate prefetch requests, we introduce changes to OPTgen and the Hawkeye predictor, which we now discuss in turn.

3.1 Prefetch-Aware OPTgen

As illustrated in Section 2, OPTgen relies on the notion of liveness intervals to emulate Belady's solution. In the absence of prefetching, both endpoints of any liveness interval are demand accesses. However, in the presence of prefetching, four different kinds of liveness intervals are possible: Demand-Demand interval (D-D), Demand-Prefetch interval (D-P), Prefetch-Demand interval (P-D), and Prefetch-Prefetch interval (P-P).

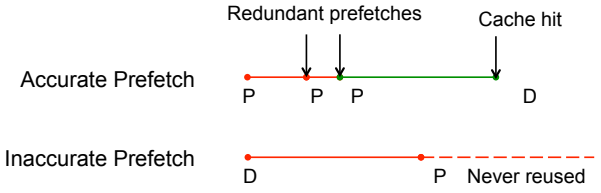

Figure 3: Liveness intervals that end with a prefetch are not cached by prefetch-aware OPTgen.

If all four types of intervals were treated identically, OPTGen would make poor caching decisions for past references. For example, P-P and D-P intervals that are deemed cache-friendly consume cache resources but do not improve the cache's hit rate because they represent redundant prefetches. The top part of Figure 3 shows that P-D intervals, not P-P intervals (redundant prefetches) contribute to cache hits for accurate prefetches. Moreover, it can be counter-productive to cache P-P and D-P intervals if the interval is not followed by a subsequent demand access. The bottom part of Figure 3 shows this case where an inaccurate prefetch can result in a

Component	Parameters	Budget
Sampler	2800 entries; 4-byte entry	11.2KB
Hawkeye Predictor	2 predictors (2K entries, 5-bit counter each)	2.56KB
Occupancy Vector	64 vectors, 128 entries each 4-bit entry	4KB
Replacement state per line	3-bit RRIP value	12KB
Additional state for sampled lines for sampled lines	64 sets $\times$ 16 ways 1-bit prefetch per line 11-bit signature per line	1.5KB

Table 1: Hawkeye hardware budget (16-way 2MB LLC)

D-P interval that is never followed by a subsequent demand. Thus, we modify OPTgen to only consider liveness intervals that end with a demand access, namely, P-D and D-D intervals. The former represents useful prefetches, and the latter represents reuse of demand accesses. By not allocating cache capacity to redundant and potentially inaccurate prefetches, OPTgen is able to make better use of the cache and to improve hit rate.

While ignoring redundant prefetches maximizes cache utilization, it can drastically increase prefetch traffic for some benchmarks. For example, in the top of Figure 3, if the two P-P intervals in red were not cached, the prefetcher would generate 3 memory requests, while it would generate just a single memory request if the P-P intervals were cached. Thus, redundant prefetches present a tradeoff between memory traffic and cache utilization. Our solution resolves this tradeoff by allowing redundant prefetches to be candidates for caching only if they have short liveness intervals, since these prefetches will reduce memory traffic without consuming cache space for long periods of time.

3.2 Prefetch-Aware Predictor

We also separate the prediction for demand and prefetch requests by employing two different predictors. As a result, we are able to better learn the caching behavior of load instructions that result in both demand and prefetch accesses. For example, a load instruction that loads cache-friendly demand accesses but issues inaccurate prefetches will be classified as cache-friendly by the demand predictor and cache-averse by the prefetch predictor.

3.3 Hardware Budget

Table 1 shows our configuration parameters for single-core configurations, as well as the hardware budget per core. For multi-core configurations, all meta-data structures except the RRIP values are scaled in proportion to the cache size. Thus, Hawkeye's hardware budget is 31.8 KB for single-core configurations and 90.2 KB for four-core configurations.

4. EVALUATION

We now describe our evaluation in the Cache Replacement Championship infrastructure.

4.1 Methodology

The CRC framework exposes four different configurations:

- Single core with 2 MB LLC without a prefetcher.
- Single core with 2 MB LLC with L1/L2 data prefetchers.
- A 4-core configuration with 8 MB of shared LLC without a prefetcher.
- A 4-core configuration with 8 MB of shared LLC with L1/L2 prefetchers.

Benchmarks. We evaluate Hawkeye on the SPEC2006 benchmarks that are sensitive to the replacement policy, that is, that show more than 2% improvement with the OPT policy. We compile the benchmarks using gcc-4.2 with the -O2 option. We run the benchmarks using the reference input set, we use SimPoint [5, 1] to generate for each benchmark a single sample of 1 billion instructions. We warm the cache for 50 million instructions and measure the behavior of 250 million instructions.

Multi-Core Workloads. Our multi-core results simulate four benchmarks running on 4 cores, choosing all combinations of the 19 most replacement-sensitive SPEC2006 benchmarks. Since simulating all combinations is extremely expensive, we randomly choose 600 of all the workload mixes. For each combination, the championship infrastructure simulates the simultaneous execution of the SimPoint samples of the constituent benchmarks until each benchmark has executed at least 250M instructions. If a benchmark finishes early, it is rewound and continues execution until every other application in the mix has finished running 250M instructions. Thus, all the benchmarks in the mix run simultaneously throughout the sampled execution.

To evaluate performance, we report the weighted speedup normalized to LRU for each benchmark combination. This metric is commonly used to evaluate shared caches [4, 3, 7, 9] because it measures the overall progress of the combination and avoids being dominated by benchmarks with high IPC. The metric is computed as follows. For each program sharing the cache, we compute its IPC in a shared environment (IPC_{shared}) and its IPC when executing in isolation on the same cache (IPC_{single}). We then compute the weighted IPC of the combination as the sum of $IPC_{shared}/IPC_{single}$ for all benchmarks in the combination, and we normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

Evaluated Caching Systems. We compare Hawkeye against SHiP [8], a state-of-the-art cache replacement algorithm; like Hawkeye, SHiP learns caching priorities for each load PC. For SHiP, we use a 16K entry Signature Hit Counter Predictor with 3-bit counters, and we use 2-bit RRIP counters for each line. Hawkeye's configuration parameters are listed in Table 1. For our multi-core evaluation, the replacement policies use common predictor structures for all cores. In particular, Hawkeye uses a single occupancy vector and a

single predictor to reconstruct and learn OPT's solution for the interleaved access stream from the past.

4.2 Config 1: Single-Core, No Prefetching

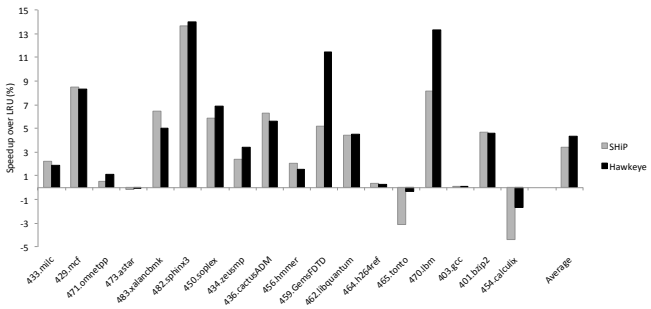

Figure 4: Speedup comparison for configuration 1 for SPEC CPU 2006 benchmarks.

Figure 4 shows that Hawkeye outperforms SHiP on a single-core system without prefetching. In particular, Hawkeye sees an average speedup of 4.5%, while SHiP sees an average speedup of 3.4%. Hawkeye also performs consistently well across benchmarks except for calculix where Hawkeye loses in comparison with LRU.

4.3 Config 2: Single-Core, With Prefetching

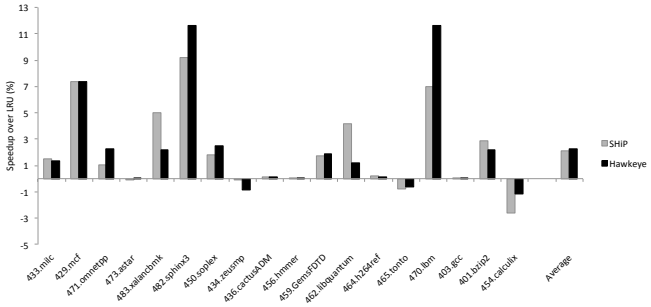

Figure 5: Speedup comparison for configuration 2 for SPEC CPU 2006 benchmarks.

Figure 5 shows that Hawkeye performs marginally better than SHiP on a single-core system with prefetching. In particular, Hawkeye sees an average speedup of 2.25%, while SHiP sees an average speedup of 2.09%. Hawkeye's loss on xalancbmk can be explained by our observation that xalanc benefits from long reuse intervals and therefore requires a window greater than $8\times$, which could not be accommodated within the CRC budget.

4.4 Config 3: Multi-Core, No Prefetching

Figure 6 shows that Hawkeye's performance scales with core count as its improvement over SHiP increases (9.2% vs. 7.1% speedup over LRU).

4.5 Config 4: Multi-Core, With Prefetching

Hawkeye and SHiP perform similarly for configuration 4.

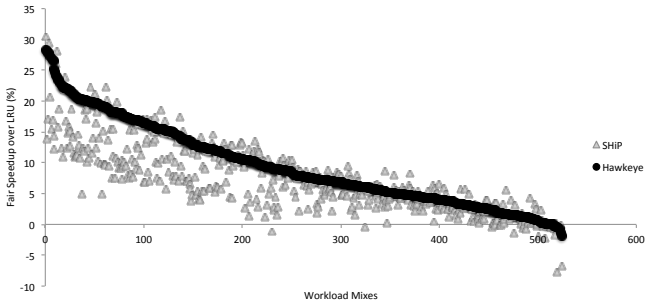

Figure 6: Speedup comparison for configuration 3 for multi-programmed SPEC CPU 2006 benchmarks.

5. CONCLUSIONS

This paper has shown that to perform well in the presence of a data prefetcher, the Hawkeye cache replacement policy benefits from two simple changes that allow it to distinguish among accurate, inaccurate, and redundant prefetches. However, even with these changes, Hawkeye's performance benefit over SHiP is quite small, suggesting that we have not solved the deeper question: What is the optimal cache solution in the presence of prefetches? Once the answer to this question is known, a Hawkeye-like solution should perform much better.

6. REFERENCES

[1] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[2] A. Jain and C. Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *43rd Annual IEEE/ACM International Symposium on Computer Architecture (ISCA)*, June 2016.

[3] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 208–219, 2008.

[4] S. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–186, 2010.

[5] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 318–319, 2003.

[6] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way cache: demand-based associativity via global replacement. In *International Symposium on Computer Architecture (ISCA)*, pages 544–555, 2005.

[7] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[8] 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 *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.

[9] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *the 36th International Symposium on Computer Architecture (ISCA)*, pages 174–183, 2009.