Back to the Future: 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

Belady's algorithm is optimal but infeasible because it requires knowledge of the future. This paper explains how a cache replacement algorithm can nonetheless learn from Belady's algorithm by applying it to past cache accesses to inform future cache replacement decisions. We show that the implementation is surprisingly efficient, as we introduce a new method of efficiently simulating Belady's behavior, and we use known sampling techniques to compactly represent the long history information that is needed for high accuracy. For a 2MB LLC, our solution uses a 16KB hardware budget (excluding replacement state in the tag array). When applied to a memory-intensive subset of the SPEC 2006 CPU benchmarks, our solution improves performance over LRU by 8.4%, as opposed to 6.2% for the previous state-of-the-art. For a 4-core system with a shared 8MB LLC, our solution improves performance by 15.0%, compared to 12.0% for the previous state-of-the-art.

Keywords: Cache replacement, Belady's Algorithm

1. INTRODUCTION

Caches are important mechanisms for reducing the long latencies of DRAM memory accesses, and their effectiveness is significantly influenced by their replacement policy. Unfortunately, cache replacement is a difficult problem. Unlike problems such as branch prediction, in which the definitive answer to the question, "Will this branch be taken?", will be readily available in a few cycles, it is difficult to get the definitive answer to the question, "Which cache line should be evicted?"

In the absence of definitive feedback, existing replacement policies build on heuristics, such as Least Recently Used (LRU) and Most Recently Used (MRU), which each work well for different workloads. However, even with increasingly clever techniques for optimizing and combining these policies, these heuristic-based solutions are each limited to specific classes of access patterns and are unable to perform well in more complex scenarios. As a simple example, consider the naive triply nested loop algorithm for computing matrix multiplication. As depicted in Figure 1, the elements of the C matrix enjoy short-term reuse, while those of the A matrix enjoy medium-term reuse, and those of the B matrix see long-term reuse. Figure 1 shows that existing replacement policies can capture some subset of the available reuse, but only Belady's algorithm [2] can effectively exploit all three forms of reuse.

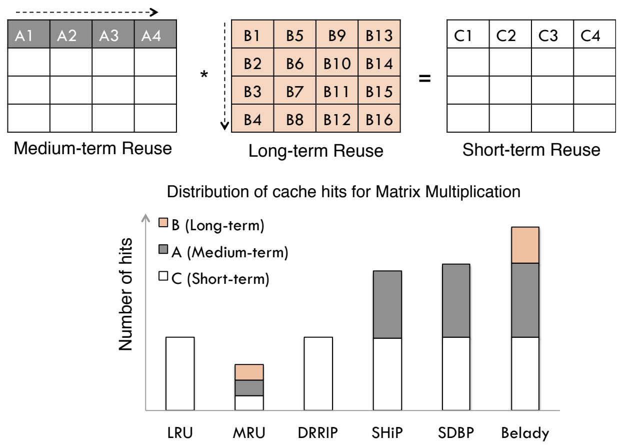


Figure 1: Existing replacement policies are limited to a few access patterns and are unable to cache the optimal combination of A, B and C.

In this paper, we present a fundamentally different approach, one that is not based on LRU, on MRU, or on any heuristic that is geared towards any particular class of access patterns. Our algorithm is instead based on Belady's algorithm: While Belady's algorithm is impractical because it requires knowledge of the future, we show that it is possible to apply a variant of Belady's algorithm to the history of past memory accesses. If past behavior is a good predictor of future behavior, then our replacement policy will approach the behavior of Belady's algorithm. We refer to the decisions made by Belady's algorithm as OPT. To learn the past behavior of Belady's algorithm, we observe that if with the OPT solution a load instruction has historically brought in lines that produce cache hits, then in the future, the same load instruction is likely to bring in lines that will also produce cache hits.

Our new cache replacement strategy thus consists of two components. The first reconstructs Belady's optimal solution for past cache accesses. The second is a predictor that learns OPT's behavior of past PCs to inform eviction decisions for future loads by the same PCs.

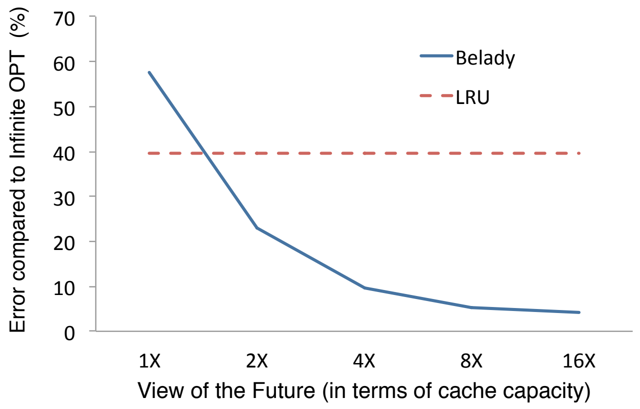


Figure 2: Belady's algorithm requires a long view of the future.

One concern with this idea is that Belady's algorithm looks arbitrarily far into the future, so our solution would theoretically need to remember an arbitrarily long history of past events. However, Figure 2 shows the impact of limiting this window of the future. Here, 1× represents a window that consists of accesses to *k* cache lines, where *k* is the capacity of the cache. We see that while Belady's algorithm performs better when it can see farther into the future, it approaches the performance of a true OPT policy when given a reuse window of 8× the cache size. Thus, we dub our new replacement policy Hawkeye[1].

Our use of OPT introduces two technical challenges. First, we need an efficient mechanism of reconstructing OPT. Second, a long history is needed to compute OPT. We solve the first problem by using the notion of liveness intervals (see Section 3), which leads to a simple and efficient solution. The use of liveness intervals is novel for cache replacement, because it explicitly conveys information about both reuse distance and the demand on the cache, which are both essential for making proper eviction decisions. We solve the second problem by using Set Dueling [25] to sample a small subset of all cache lines.

The result is that with 16 KB of additional storage (plus tag storage), Hawkeye can compute OPT's solution for past accesses with 99% accuracy. Of course, past behavior does not always model future behavior, so Hawkeye's performance does not match OPT's. Nevertheless, as shown in Figure 3, Hawkeye performs significantly better than previous policies on a memory-intensive SPEC CPU 2006 benchmarks.

To summarize, this paper makes the following contributions:

[1]Hawks are known for their excellent long-range vision and can see up to 8× more clearly than the best humans.

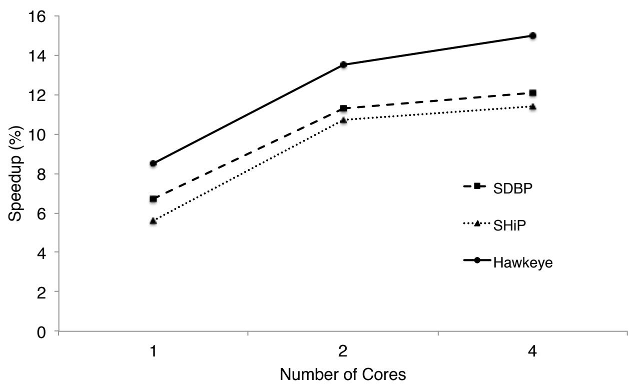


Figure 3: Speedup over LRU for 1, 2, and 4 cores.

- We introduce the Hawkeye cache replacement policy, which learns Belady's optimal solution (OPT) for past accesses to guide future replacement decisions.
- We introduce the OPTgen algorithm for efficiently computing OPT for a history of past cache accesses. OPTgen builds on three critical insights: (1) OPT's decision depends not only on a cache line's reuse interval but also on the overlap of reuse intervals, which represents the demand on the cache; (2) OPT's decision for a past access can be determined at the time of its next use; (3) a reuse window of 8× is necessary to generate OPT's solution accurately.
- To allow Hawkeye to practically simulate OPT, we use Set Dueling [25] to capture long-term behavior with a small 12KB hardware budget.
- We evaluate Hawkeye using the Cache Replacement Championship simulator [1] and show that Hawkeye substantially improves upon the previous state-of-the art. On the SPEC CPU 2006 benchmark suite, Hawkeye obtains a 17.0% miss rate reduction over LRU, compared with 11.4% for Khan, et al.'s SDBP policy [16]. In terms of performance, Hawkeye improves IPC over LRU by 8.4%, while SDBP improves IPC by 6.2%. On a 4-core system, Hawkeye improves speedup over LRU by 15.0%, while SDBP improves speedup by 12.0%.

This paper is organized as follows. Section 2 discusses related work. Sections 3 and 4 describe and evaluate our solution. We conclude in Section 5.

2. RELATED WORK

Since Belady's optimal cache replacement algorithm [2] was introduced in 1966, there has been considerable work in the development of practical replacement algorithms [4, 39, 28, 6, 35, 19, 32, 36, 12, 25, 13, 16, 37, 17, 20, 29, 8, 30, 7]. Here, we focus on work that is most closely related to Hawkeye, organized based on the type of information that they use to make decisions.

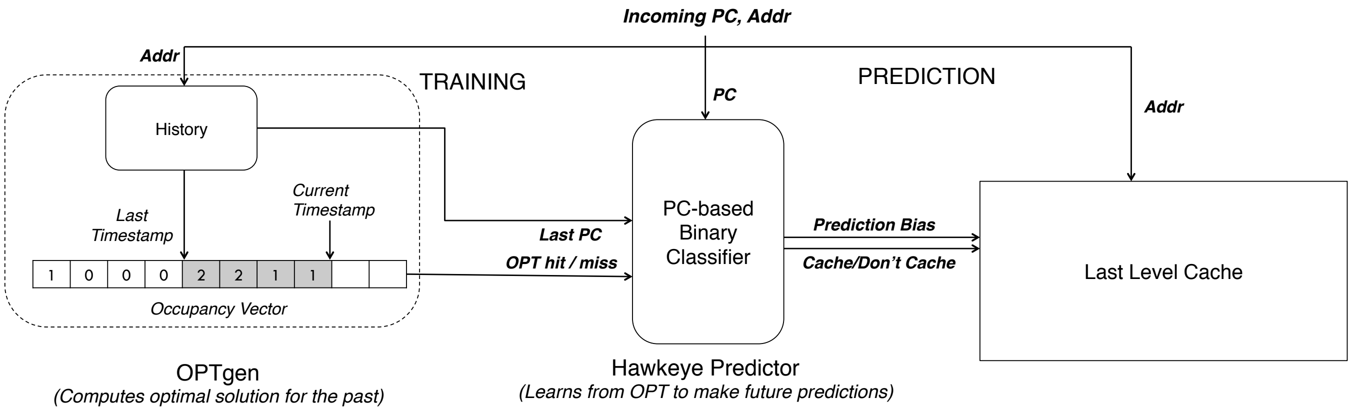


Figure 4: Block diagram of the Hawkeye replacement algorithm.

Short-Term History Information.

Many solutions use short-term information that reflects the current state of the cache, ignoring any information about cache lines that have been evicted.

Such solutions typically uses heuristics that cater to specific types of cache access patterns. For example, *recency-friendly* policies such as LRU and its variations [13, 36, 32] prioritize recently used lines under the assumption that they will soon be used again. Other policies favor lines with high access-frequency under the assumption that frequently used lines will soon be used again [29, 8, 27, 21, 18]. Jaleel et al. enhance recency-friendly policies by cleverly using 2-bits of re-reference interval prediction (RRIP) [12] to eliminate cache pollution due to streaming accesses. Hawkeye uses RRIP's idea of aging to adapt to changes in phase behavior.

To avoid the pathological behavior of recency-friendly policies on workloads that exhibit large reuse distances, *thrash-resistant* policies [25, 30] discard the most recently used line instead of the least recently used line, thereby retaining a portion of the active working set. Unfortunately, thrash-resistant policies perform poorly in the presence of recency-friendly or streaming accesses.

Because different replacement policies favor different cache access patterns, hybrid solutions have been developed [34, 26, 25] to dynamically select among competing policies. The key challenges with hybrid replacement are the management of additional information and the high hardware cost for dynamic selection. Qureshi et al. introduce Dynamic Set Sampling (DSS) [25], an inexpensive mechanism that chooses the best policy by sampling a few dedicated sets to assess the efficacy of the desired policy. Thus, DSS allows the policy to change over time, but it selects a single policy for all cache lines. By contrast, Hawkeye can use different policies for each load instruction.

Long-Term History Information.

Recent work exploits long-term information, including information about lines that have been evicted from the cache. For example, some policies [15, 5] predict reuse distances for incoming lines based on past reuse distributions, but such policies are expensive. Moreover, unlike Hawkeye's liveness intervals, reuse distance alone leads to inaccurate decisions because it does not account for the demand on the cache. For example, a line with a long reuse interval can remain in the cache if there is low demand on the cache, while at some other point in time, a line with a short reuse distance can be evicted from the cache if there is high demand for the cache. (See Section 3.1).

Hawkeye builds on recent work that learns the caching behavior of past load instructions to guide future caching decisions: SHiP [37] uses a predictor to identify instructions that load streaming accesses, while SDBP [16] uses a predictor to identify lines that are likely to be evicted by the LRU policy. Thus, SHiP and SDBP improve cache efficiency by not dedicating cache resources to lines that are likely to be evicted. However, these policies can be inaccurate because they learn the behavior of heuristic-based replacement policies (LRU and RRIP), which perform well for a limited class of access patterns. By contrast, Hawkeye simulates and learns from the past behavior of OPT, which makes no assumptions about access patterns.

Hawkeye considers a longer history of operations than either SHiP or SDBP, maintaining a history that is 8 times the size of the cache. To simulate OPT's behavior, we introduce an algorithm which bears resemblance to the Linear Scan Register Allocator [23] but solves a different problem.

Future Information.

Another class of replacement policies takes inspiration from victim caches [14] and defers replacement decisions to the future when more information is available. For example, the Shepherd Cache [28] emulates OPT by deferring replacement decisions until future reuse can be observed, but it cannot emulate OPT accurately because it uses an extremely limited window into the future; larger windows would be expensive because unlike Hawkeye's history of past references, the Shepherd Cache must store the contents of the lines that make up its window into the future. Other solutions [24, 30] use slightly longer windows (2×) into the future, but these solutions do not model OPT. In general, these solutions make a tradeoff between the window size and the precision of their replacement decisions.

Other Types of Information.

Cache performance can be improved by not only reducing the number of misses but by selectively eliminating expensive misses. For example, MLP [26] and prefetch-friendliness [38] can be used to reduce the overall performance penalty of LLC misses. The Hawkeye policy focuses on cache misses and is complementary to these techniques.

3. OUR SOLUTION

Conceptually, we view cache replacement as a binary classification problem, where the goal is to determine if an incoming line is *cache-friendly* or *cache-averse:* Cache-friendly lines are inserted with a high priority, while cache-averse lines are marked as eviction candidates for future conflicts. To determine how incoming lines should be classified, Hawkeye reconstructs Belady's optimal solution for past accesses to learn the behavior of individual load instructions.

Figure 4 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 (not shown), which reduces the amount of state required to reconstruct OPT's behavior. We now describe each component in more detail.

3.1 OPTgen

OPTgen determines what would have been cached if the OPT policy had been used. Starting from the oldest reference and proceeding forward in time, OPTgen assigns available cache capacity to lines in the order that they are reused. To assign cache capacity to old references, OPTgen repeatedly answers the following question: Given a history of memory references that includes a reference to cache line X, would the next reference to the same line, which we refer to as X', be a hit or a miss under the OPT policy?

To answer this question, we observe that OPT's decision for any past reference X can be determined at the time of its next reuse X' because any later reference is farther into the future than X', so Belady's algorithm would favor X' over that other line [3]. Thus, we define the time period that starts with a reference to X and proceeds up to (but not including) its next reference X' to be X's *usage interval*. Intuitively, X's usage interval represents its demand on the cache, which allows OPTgen to determine whether the reference to X' would result in a cache hit.

If we further define a cache line's *liveness interval* to be the time period during which that line resides in the cache under the OPT policy, then X would be a cache miss if at any point in its usage interval the number of overlapping liveness intervals matches the cache's capacity. Otherwise, X would be a cache hit.

For example, consider the sequence of accesses in Figure 5, which includes X's usage interval. Here, the cache capacity is two. We assume that OPTgen has already determined the liveness intervals of A, B, and C, 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 uses an *occupancy vector* to record the occupied

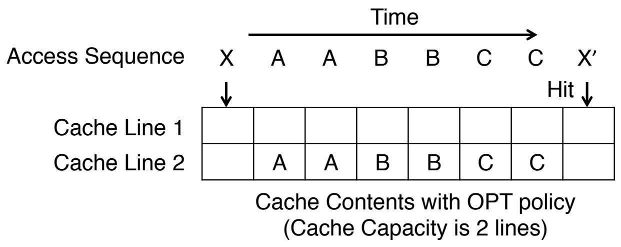


Figure 5: Intuition behind OPTgen.

cache capacity over time; each entry of this vector contains the number of liveness intervals that overlap at a particular time. To understand OPTgen's use of the occupancy vector, consider the example access stream in Figure 6(a) and OPT's solution for this access stream in Figure 6(b). Figure 6(c) shows how the occupancy vector is computed over time. In particular, the top of Figure 6(c) shows the sequence of lines that is accessed over time. For example, line B is accessed at Times 1 and 2. Each row in Figure 6(c) represents the state of the occupancy vector at a different point in time, so, for example, the third row (T=2) illustrates the state of the occupancy vector after Time 2, i.e., after the second access of line B and after OPTgen has determined that OPT would have placed B in the cache at Time 1.[2]

For an access to X, the occupancy vector for the usage interval (shown in gray) is updated as follows:

- The most recent entry of the occupancy vector (corresponding to this access to X) is set to 0.
- When line X is loaded for the first time, no further changes to the occupancy vector are made, reflecting the fact that OPT makes decisions based on the next reuse of the line.
- If X is not a first-time load, OPTgen checks to see if every element corresponding to the usage interval is less than the cache capacity: If so, then OPT would have placed X in the cache, so the shaded portions of the occupancy vector are incremented; if not, then X would have been a cache miss, so the occupancy vector is not modified.

For example, in Figure 6(c), consider the access of D at Time 8. Using the occupancy vector before T=8 (same as the occupancy vector at T=7 with a 0 added for Time 8), OPTgen sees that the elements in the usage interval (the values at positions 4 through 7) are all less than the cache capacity (2), so it concludes that D would be found in the cache at Time 8, and it increments the elements in positions 4 through 7.

As another example, consider the access to C at Time 11; some of the shaded elements have value 2, so OPTgen concludes that this access to C would have been a cache miss, so it does not increment the shaded elements of the occupancy vector. We see that by not incrementing any of the shaded

[2]In this discussion, we will use "T=1" to refer to rows of the figure, and we will use "Time 1" to refer to events in the Access Sequence, ie, columns in the figure.

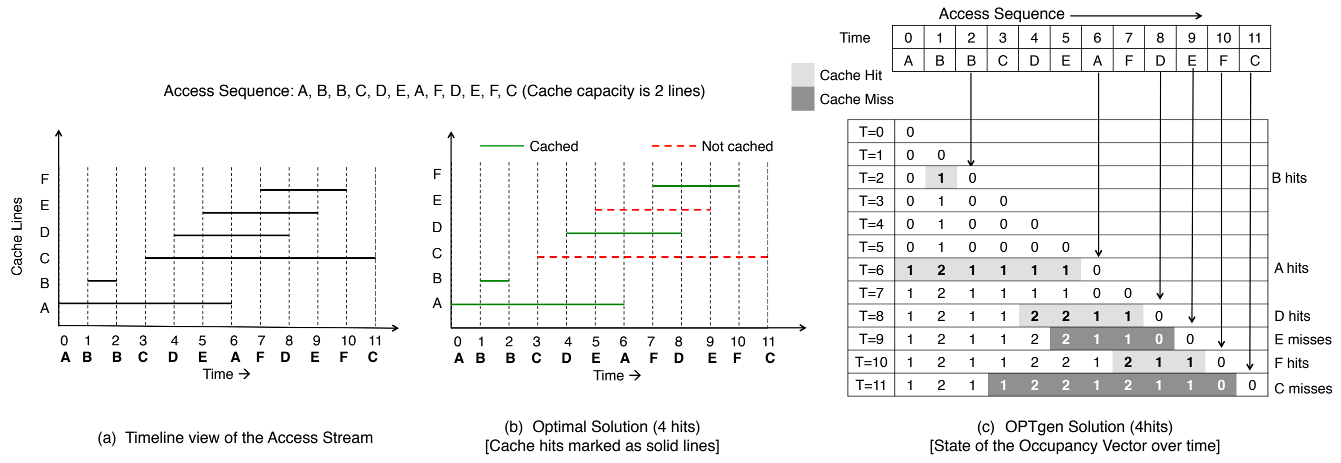


Figure 6: Example to illustrate OPTgen.

elements of the occupancy vector for cache misses, OPTgen assumes that misses will bypass the cache. If we wanted OPTgen to instead assume a cache with no bypassing, then the most recent entry (corresponding to the current access) would have been initialized to 1 instead of 0.

The example in Figure 6 highlights two important points. First, by reconstructing OPT, OPTgen is able to recognize both long-term and short-term reuse that is cache friendly. For example, both *A* and *B* hit in the cache even though the reuse interval of *A* far exceeds the cache capacity. Second, OPTgen can be implemented in hardware with very little logic because the occupancy vector can be maintained with simple read, write, and compare operations.

OPTgen for Set-Associative Caches.

For set-associative caches, OPTgen maintains one occupancy vector for each cache set such that the maximum capacity of any occupancy vector entry never exceeds the cache associativity. Occupancy vectors measure time in terms of cache accesses to the corresponding set, and they include enough entries to model 8× the size of the set (or the associativity). Thus, for a 16-way set-associative cache, each occupancy vector has 128 entries (corresponding to 8× the capacity of the set), and each occupancy vector entry is 4 bits wide, as its value cannot exceed 16.

3.2 Reducing the Size of OPTgen

So far, our discussion of OPTgen has not considered resource constraints, as we have assumed that the occupancy vector measures time in terms of individual cache accesses. We have also assumed that OPTgen has knowledge of liveness intervals that extend back 8× the size of the cache, which for a 16-way 2MB cache requires OPTgen to track over 260K entries in both the occupancy vectors and the history. This section describes two techniques that reduce these hardware requirements.

3.2.1 Granularity of the Occupancy Vector

To reduce the size of the occupancy vector, we increase its granularity so that each element represents a *time quantum*, a unit of time as measured in terms of cache accesses. Our sensitivity studies (Section 4) show that a time quantum of 4 cache accesses works well, which for a 16-way set-associative cache reduces the size of the occupancy vector from 128 to 32 entries.

Since occupancy vector entries for 16-way set-associative caches are 4 bits wide, the occupancy vector for each set requires 16 bytes of storage, which for a 2MB cache would still amount to 32KB storage for all occupancy vectors (2048 sets × 16 bytes per set).

3.2.2 Set Dueling

To further reduce our hardware requirements, we use the idea of Set Dueling [25], which monitors the behavior of a few randomly chosen sets to make predictions for the entire cache. To extend Set Dueling to Hawkeye, OPTgen reconstructs the OPT solution for only 64 randomly chosen sets. Section 4.3 shows that reconstructing the OPT solution for 64 sets is sufficient to emulate the miss rate of an optimal cache and to train the Hawkeye Predictor appropriately.

Set Dueling reduces Hawkeye's storage requirements in two ways. First, since OPTgen now maintains occupancy vectors for 64 sets, the storage overhead for all occupancy vectors is only 1 KB (64 occupancy vectors × 16 bytes per occupancy vector). Second, it dramatically reduces the size of the history, which now tracks usage intervals for only 64 sampled sets.

To track usage intervals for the sampled sets, we use a Sampled Cache. The Sampled Cache is a distinct structure from the LLC, and each entry in the Sampled Cache maintains a 2-byte address tag, a 2-byte load instruction PC, and a 1-byte timestamp. For 64 sets, the Sampled Cache would need to track a maximum of 8K addresses to capture usage intervals spanning a history of 8× the size of the cache, but we find that because of repeated accesses to the same address, 2400 entries in the Sampled Cache are enough to provide an 8× history of accesses. Thus the total size of the Sampled Cache is 12KB, and we use an LRU policy for eviction when the Sampled Cache is full.

3.3 The Hawkeye Predictor

The second major component of Hawkeye is a predictor that classifies the lines loaded by a given PC as either cache-friendly or cache-averse. This predictor builds on the observation that the majority of OPT's decisions for loads by a given PC are similar and therefore predictable. Figure 7 quantifies this observation, showing that for SPEC2006, the average per-PC *bias*—the probability that loads by the same PC have the same caching behavior as OPT—is 90.4%.

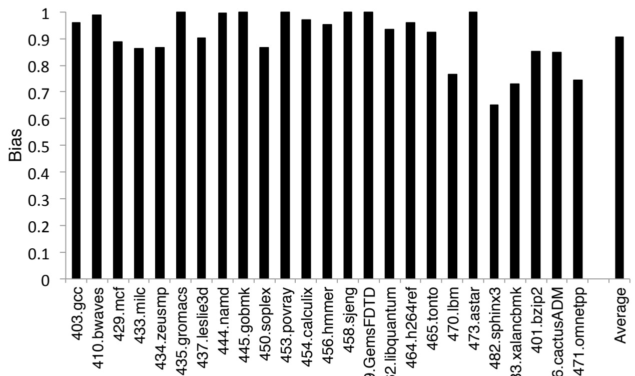


Figure 7: Bias of OPT's decisions for PCs.

Thus, the Hawkeye Predictor learns whether loads by a given instruction would have resulted in hits or misses under the OPT policy: If OPTgen determines that a line *X* would be a cache hit under the OPT policy, then the PC that last accessed *X* is trained positively; otherwise, the PC that last accessed *X* is trained negatively. The Hawkeye Predictor has 8K entries, it uses 3-bit counters for training, and it is indexed by a 13-bit hashed PC.

For every cache access, the predictor is indexed by the current load instruction, and the high-order bit of the corresponding 3-bit counter indicates whether the line is cache-friendly (1) or cache-averse (0). As we explain in Section 3.4, this prediction determines the line's replacement state.

Occasionally, load instructions will have a low bias, which will result in inaccurate predictions. Our evaluation shows that we can get a small performance gain by augmenting Hawkeye's predictions to include confidence, but the gains are not justified by the additional hardware complexity, so we do not evaluate this feature in this paper.

3.4 Cache Replacement

Our overall cache replacement goal is to use Hawkeye's predictions within a phase and to use an LRU strategy at phase change boundaries, when Hawkeye's predictions are likely to be incorrect. Thus, Hawkeye first chooses to evict cache-averse lines, as identified by the Hawkeye Predictor. If no lines are predicted to be cache-averse, then the oldest cache-friendly line (LRU) is evicted, allowing Hawkeye to adapt to phase changes. This scheme is likely to evict cache-averse lines from the new working set before evicting cache-friendly lines from the old working set, but this behavior is harmless because cache-averse lines from the new working set are likely to be evicted anyway. To correct the state of the predictor after a phase change, the predictor is detrained when cache-friendly lines are evicted. In particular, when a cache-friendly line is evicted, the predictor entry corresponding to the last load PC of the evicted line is decremented if the evicted line is present in the sampler.

Hawkeye Prediction \ Hit or Miss	Cache Hit	Cache Miss
Cache-averse	RRIP = 7	RRIP = 7
Cache-friendly	RRIP = 0	RRIP = 0; Age all lines: if (RRIP < 6) RRIP++;

Table 1: Hawkeye's Update Policy.

To implement this policy efficiently, we associate all cache-resident lines with 3-bit RRIP counters [12] that represent their eviction priorities; lines with a high eviction priority have a high RRIP value, and lines with low eviction priorities have a low RRIP value. Conceptually, the RRIP counter of a line combines information about Hawkeye's prediction for that line and its age. On every cache access (both hits and misses), the Hawkeye predictor generates a binary prediction to indicate whether the line is cache-friendly or cache-averse, and this prediction is used to update the RRIP counter as shown in Table 1. In particular, rows in Table 1 represent Hawkeye's prediction for a given access, and columns indicate whether the access was a cache hit or miss. For example, if the current access hits in the cache and is predicted to be cache-averse, then its RRIP value is set to 7. As another example, when a newly inserted line (cache miss) is predicted to be cache-friendly, its RRIP value is set to 0, and the RRIP values of all other cache-friendly lines are incremented to track their relative age. In general, cache-friendly lines are assigned an RRIP value of 0, and cache-averse lines are assigned an RRIP value of 7.

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 load instruction if the evicted line is present in the sampler.

Hawkeye's insertion policy differs in three ways from other RRIP-based policies [12, 37]. First, lines that are predicted to be cache-friendly are never saturated to the highest value, which ensures that cache-averse lines are always prioritized for eviction. Second, lines that are predicted to be cache-friendly are always assigned an RRIP value of 0 regardless of whether they were hits or misses. And finally, cache hits are promoted to an RRIP value of 0 only if they are predicted to be cache-friendly. These differences are designed to give the Hawkeye Predictor greater influence over the RRIP position than cache hits or misses.

L1 I-Cache	32 KB 4-way, 1-cycle latency
L1 D-Cache	32 KB 4-way, 1-cycle latency
L2 Cache	256KB 8-way, 10-cycle latency
Last-level Cache	2MB, 16-way, 20-cycle latency
DRAM	200 cycles
Two-core	4MB shared LLC (25-cycle latency)
Four-core	8MB shared LLC (30-cycle latency)

Table 2: Baseline configuration.

4. EVALUATION

4.1 Methodology

We evaluate Hawkeye using the simulation framework released by the First JILP Cache Replacement Championship (CRC) [1], which is based on CMP$im [10] and models a 4-wide out-of-order processor with an 8-stage pipeline, a 128-entry reorder buffer and a three-level cache hierarchy. The parameters for our simulated memory hierarchy are shown in Table 2. The infrastructure generates cache statistics as well as overall performance metrics, such as IPC.

Benchmarks. We evaluate Hawkeye on the entire SPEC2006 benchmark suite.[3] For brevity, Figures 8 and 9 show averages for all the benchmarks but only include bar charts for the 20 replacement-sensitive benchmarks 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, and as with the CRC, we use SimPoint [22, 9] to generate for each benchmark a single sample of 250 million instructions. We warm the cache for 50 million instructions and measure the behavior of the remaining 200 million instructions.

Multi-Core Workloads. Our multi-core results simulate either two benchmarks running on 2 cores or four benchmarks running on 4 cores, choosing all combinations of the 12 most replacement-sensitive SPEC2006 benchmarks. For 2 cores, we simulate all possible combinations, and for 4 cores, we randomly choose one tenth of all the workload mixes, resulting in a total of 136 combinations. For each combination, we simulate 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 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. Our multi-core simulation methodology is similar to the methodologies used by recent work [12, 37, 16] and the Cache Replacement Championship [1].

To evaluate performance, we report the weighted speedup normalized to LRU for each benchmark combination. This metric is commonly used to evaluate shared caches [16, 11, 33, 39] 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 two state-of-the-art cache replacement algorithms, namely, SDBP [16] and SHiP [37]; like Hawkeye, both SHiP and SDBP learn caching priorities for each load PC. We also compare Hawkeye with two policies that learn global caching priorities, namely, Dueling Segmented LRU with Adaptive Bypassing [6] (DSB, winner of the 2010 Cache Replacement Championship) and DRRIP [12].

DRRIP and SHiP use 2-bit re-reference counters per cache line. For SHiP, we use a 16K entry Signature Hit Counter Predictor with 3-bit counters. For SDBP, we use a 1-bit dead block prediction per cache line, 8KB sampler, and 3 prediction tables, each with 4K 2-bit counters. Our SDBP and SHiP implementation is based on the code provided by the respective authors with all parameters tuned for our execution. For DSB, we use the code provided on the CRC website and explore all tuning parameters. To simulate Belady's OPT, we use an in-house trace-based cache simulator. Hawkeye's configuration parameters are listed in Table 3.

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.

Power Estimation. We use CACTI [31] to estimate the dynamic energy consumed by the various replacement policies. Our energy estimates are limited to the additional components introduced by the replacement policy and do not consider the impact of improved cache performance on system-wide energy consumption.

4.2 Comparison with Other Policies

Figure 8 shows that Hawkeye significantly reduces the LLC miss rate in comparison with the two state-of-the-art replacement policies. In particular, Hawkeye achieves an average miss reduction of 17.0% on the 20 memory-intensive SPEC benchmarks, while SHiP and SDBP see miss reductions of 11.7% and 11.4%, respectively. Figure 9 shows that Hawkeye's reduced miss rate translates to a speedup of 8.4% over LRU. By contrast, SHiP and SDBP improve performance over LRU by 5.6% and 6.2%, respectively.

Figure 9 demonstrates two important trends. First, SDBP and SHiP each perform well for different workloads, but their performance gains are not consistent across benchmarks. For example, SHiP achieves the best performance for cactus, mcf, and sphinx but performs poorly on gems and tonto. By contrast, Hawkeye performs consistently well on all the workloads. Second, in contrast with the other re-

[3]We currently cannot run perl on our platform, leaving us with 28 benchmark programs.

[4]The results for the replacement-insensitive benchmarks (bwaves, milc, povray, dealII, sjeng, wrf, gamess and namd) are averaged in the a single bar named "Rest of SPEC".

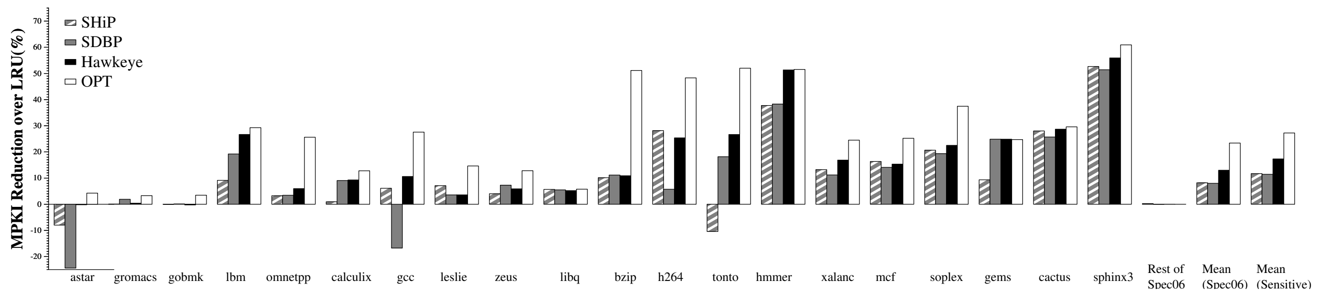


Figure 8: Miss rate reduction for all SPEC CPU 2006 benchmarks.[4]

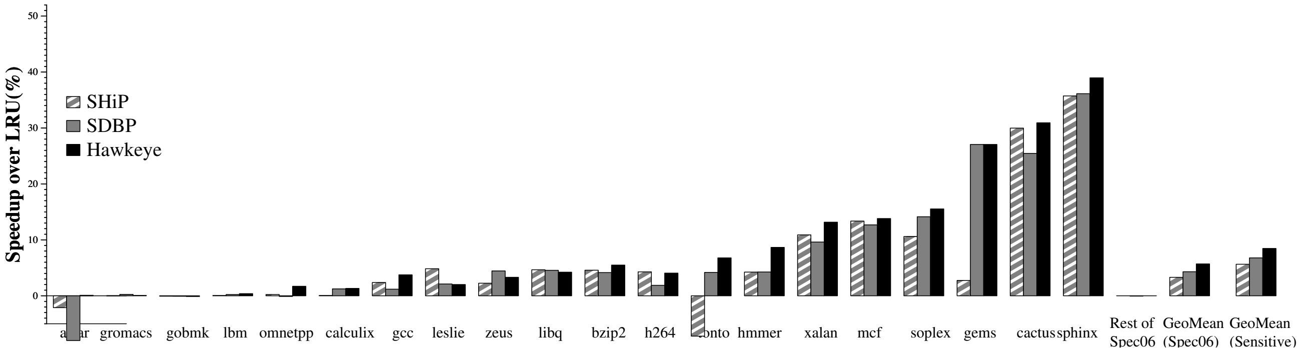


Figure 9: Speedup comparison for all SPEC CPU 2006 benchmarks.

placement policies, Hawkeye does not perform worse than the LRU baseline on any of the benchmarks. For example, SHiP and SDBP both slow down astar, and they increase the miss rates of tonto and gcc, respectively. These results reinforce our claim that previous replacement policies are geared to specific classes of access patterns, whereas by learning from OPT, Hawkeye can adapt to any workload.

Finally, Figure 10 shows that Hawkeye's performance improvement over LRU is much greater than DRRIP's (3.3% vs. 8.4%) and almost twice as large as DSB's (4.2% vs 8.4%). To understand Hawkeye's benefit, we observe that DRRIP learns a single policy for the entire cache, while DSB learns a single bypassing priority. By contrast, Hawkeye can learn a different priority for each load PC. Since it is common for cache-friendly and cache-averse lines to occur simultaneously, any global cache priority is unlikely to perform as well as Hawkeye.

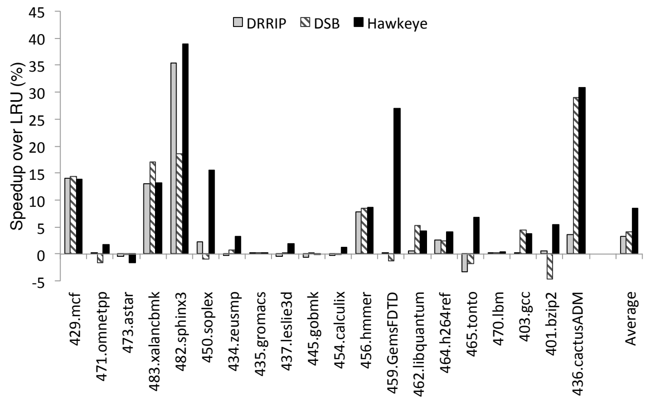


Figure 10: Comparison with DRRIP and DSB.

Sensitivity to Cache Associativity. Higher associativity gives a replacement policy more options to choose from, and since Hawkeye is making more informed decisions than the other policies, its advantage grows with higher associativity, as shown in Figure 11.

Except for SDBP, all of the replacement policies benefit from higher associativity. SDBP deviates from this trend because it uses a decoupled sampler[5] to approximate LRU evictions from the cache itself, and its performance is sensitive to a mismatch in the rate of evictions from the cache and the sampler. Hawkeye's decoupled sampler prevents this by employing a strategy that is independent of the cache configuration.

Hardware Overhead. Table 3 shows the hardware budget for Hawkeye's three memory components, namely, the sampler, the occupancy vector, and the PC-based predictor. For a 2MB cache, Hawkeye's total hardware budget is 28KB, including the per-cache-line replacement state in the tag array. Table 4 compares the hardware budgets for the evaluated replacement policies. We note that Hawkeye's hardware requirements are well within the 32KB budget constraint for the Cache Replacement Championship [1].

Finally, we also observe that like other set dueling based replacement policies, such as SDBP and SHiP, Hawkeye's

[5]For each data point in Figure 11, we choose the best performing sampler associativity.

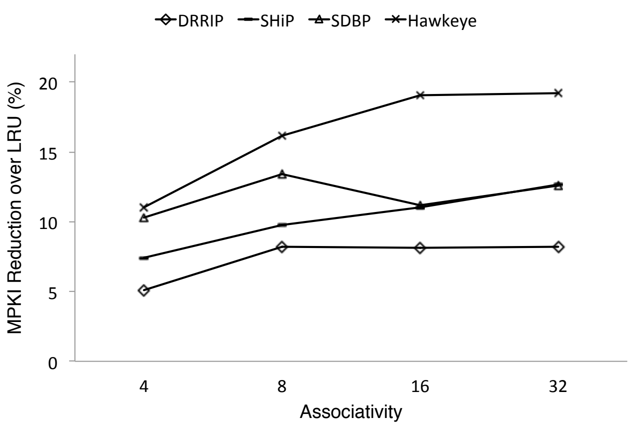


Figure 11: Sensitivity to cache associativity.

Component	Parameters	Budget
Sampler	2400 entries; 5-byte entry	12KB
Occupancy Vector	64 vector, 32 entries each 4-bit entry Quantum=4 accesses	1KB
Hawkeye Predictor	8K entries; 3-bit counter	3KB
Replacement State per line	3-bit RRIP value	12KB

Table 3: Hawkeye hardware budget (16-way 2MB LLC)

hardware budget for meta-data storage (Occupancy Vectors and Sampled Cache) does not increase with additional cores or larger caches.

Hardware Complexity. Because every occupancy vector update can modify up to 32 entries, OPTgen would appear to perform 32 writes on every cache access, which would consume significant power and complicate queue management. In fact, the number of updates is considerably smaller for three reasons. First, the occupancy vector is updated only on sampler hits, which account for only 5% of all cache accesses. Second, we implement the occupancy vector as an array of 32-bit lines, such that each line contains eight 4-bit entries of the occupancy vector. On a sampler hit, all eight entries in the cache line can be updated in parallel using a modified 32-bit adder (which performs addition on 4-bit chunks of the line). As a result, a sampler hit requires at most 4 writes to the occupancy vector. Third, Figure 12 shows the distribution of the number of entries that are updated on each sampler hit: 85% of these accesses update 16 entries or fewer, which means that they modify no more than 2 lines and can complete in 2 cycles. Moreover, 65% of the accesses update 8 entries or fewer, so they modify no more than 1 line. Because occupancy vector updates are not on the critical path, these latencies do not affect performance.

Energy Consumption. Hawkeye does not increase the energy consumption of cache lookups or evictions, but it consumes extra energy for the sampler, the predictor, and the occupancy vector. We compute the dynamic energy consumption of each of these components by computing the energy per operation using CACTI and by computing the number of probes to each component as a fraction of the total number of LLC accesses. While the predictor is probed on every LLC access, the sampler is only probed for LLC accesses belonging to the sampled cache sets, and the occupancy vector is accessed only for sampler hits. As shown in Figure 12, the great majority of the occupancy vector updates modify no more than 4 lines.

We find that the Hawkeye Predictor, sampler, and occupancy vector consume 0.4%, 0.5% and 0.1%, respectively, of the LLC's dynamic energy consumption, which results in a total energy overhead of 1% for the LLC. Thus, Hawkeye's energy overhead is similar to SDBP's (both Hawkeye and SDBP use a decoupled sampler and predictor), while SHiP's energy overhead is 0.5% of the LLC because it does not use a decoupled sampler.

Policy	Predictor Structures	Cache Meta-data	Hardware Budget
LRU	None	16KB	16KB
DRRIP	8 bytes	8KB	8KB
SHiP	4KB SHCT 2KB PC tags	8KB	14KB
SDBP	8KB sampler 3KB predictor	16KB	27KB
Hawkeye	12KB sampler 1KB OPTgen 3KB predictor	12KB	28KB

Table 4: Comparison of hardware overheads.

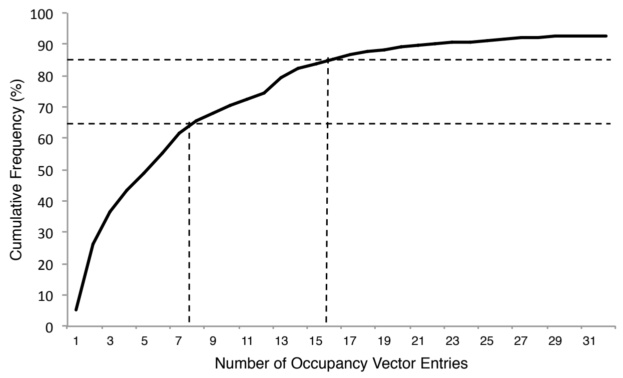


Figure 12: CDF of number of Occupancy Vector entries updated on sampler hits. 85% of the accesses update 16 entries or fewer, so they modify no more than 2 lines.

4.3 Analysis of Hawkeye's Performance

There are two aspects of Hawkeye's accuracy: (1) OPTgen's accuracy in reconstructing the OPT solution for the past, and (2) the Hawkeye Predictor's accuracy in learning the OPTgen solution. We now explore both aspects.

OPTgen Simulates OPT Accurately. Recall from Section 3 that OPTgen maintains occupancy vectors for only 64 sampled sets, and each entry of the occupancy vector holds the number of cache-resident lines that the OPT policy would retain in a given time quantum.

Figure 13 shows that OPTgen is accurate when it models occupancy vectors for all sets and uses a time quantum of 1. When sampling 64 sets, OPTgen's accuracy decreases by 0.5% in comparison with the true OPT solution, and with a time quantum of 4 cache accesses, its accuracy further decreases by only 0.3%. Thus, when using a time quantum of 4, OPTgen can achieve 99% accuracy in modeling the OPT solution with 64 occupancy vectors.

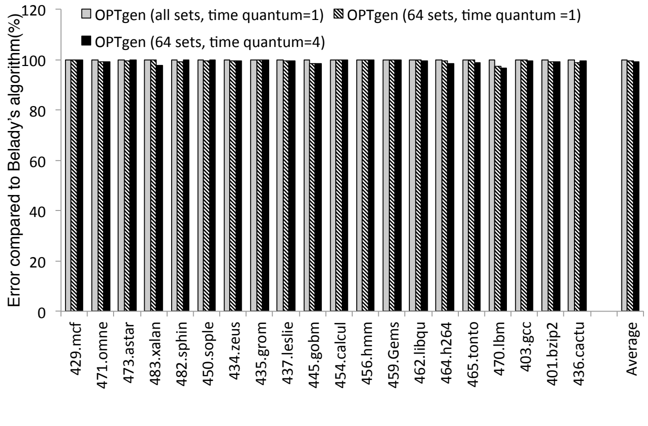


Figure 13: OPTgen simulates OPT accurately.

Predictor Accuracy. Figure 14 shows that the Hawkeye Predictor is 81% accurate in predicting OPT's decisions for future accesses. There are two sources of inaccuracy: (1) Optimal decisions of the past may not accurately predict the future; (2) the predictor may learn slowly or incorrectly due to resource limitations and training delay. Since the average bias of the OPT solution for load instructions is 91%, we conclude that the predictor contributes to the remaining loss.

Sampler Accuracy. Figure 15 shows that on average, a sampled history has little impact on Hawkeye's performance. However, the impact of sampling on Hawkeye's performance varies with benchmark. For benchmarks such as bzip2, calculix, and tonto, sampling actually improves performance because the sampled history not only filters out noisy training input, but it also accelerates training by aggregating many training updates into a single event. For zeusmp, soplex, and h264, performance is degraded with a sampled history because the samples are unable to adequately represent the past behavior of these benchmarks.

Distribution of Eviction Candidates. Recall that when all eviction candidates are predicted to be cache-friendly, Hawkeye evicts the oldest line (LRU). Figure 16 shows the frequency with which the LRU candidate is evicted. We see that the Hawkeye Predictor accounts for 71% of the overall evictions, though the distribution varies widely across

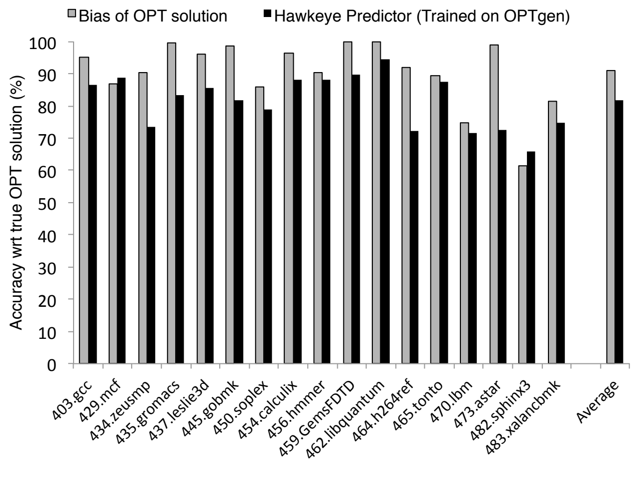


Figure 14: Accuracy of the Hawkeye Predictor.

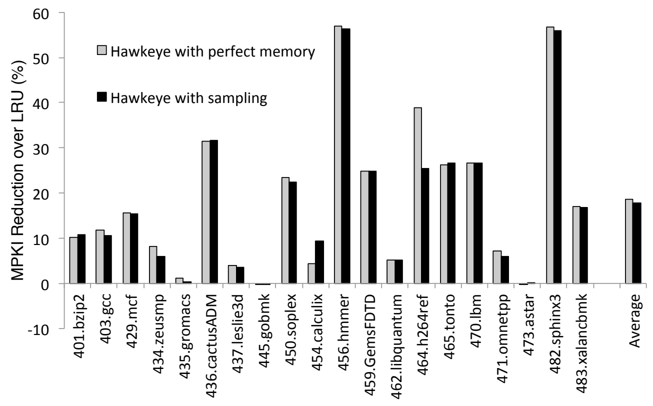


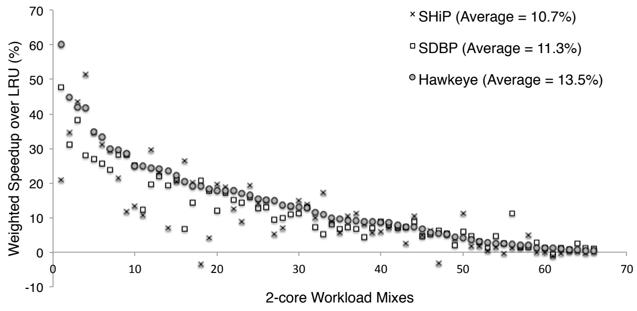
Figure 15: Impact of sampling on performance.

benchmarks. For benchmarks that have a cache-resident working set, such as astar, gromacs, and gobmk, Hawkeye learns that most accesses are cache hits, so it typically defaults to the LRU policy. By contrast, for benchmarks that have a complex mix of short-term and long-term reuse, such as mcf, xalanc, and sphinx, the Hawkeye Predictor accounts for a majority of the evictions, and the LRU evictions occur only during infrequent working set transitions.

4.4 Multi-Core Evaluation

The left side of Figure 17 shows that on a 2-core system with a shared LLC, Hawkeye's advantage over SDBP and SHiP increases, as Hawkeye achieves a speedup of 13.5%, while SHiP and SDBP see speedups of 10.7% and 11.3%, respectively. The right side of Figure 17 shows that Hawkeye's advantage further increases on a 4-core system, with Hawkeye improving performance by 15%, compared with 11.4% and 12.1% for SHiP and SDBP, respectively. On both 2-core and 4-core systems, we observe that while SHiP outperforms both SDBP and Hawkeye on a few workload mixes, its average speedup is the lowest among the three policies, which points to large variations in SHiP's performance.

Figures 18 and 3 summarize Hawkeye performance as we



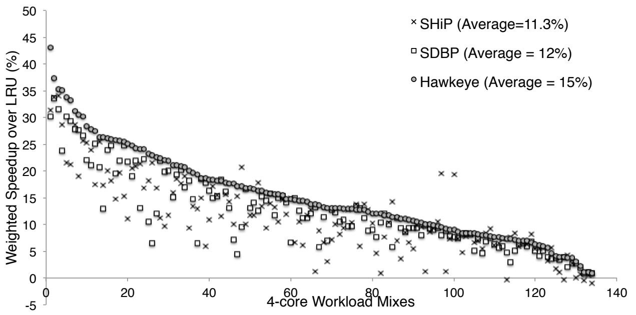


Figure 17: Weighted speedup for 2 cores with shared 4MB LLC (left) and 4 cores with shared 8MB LLC (right).

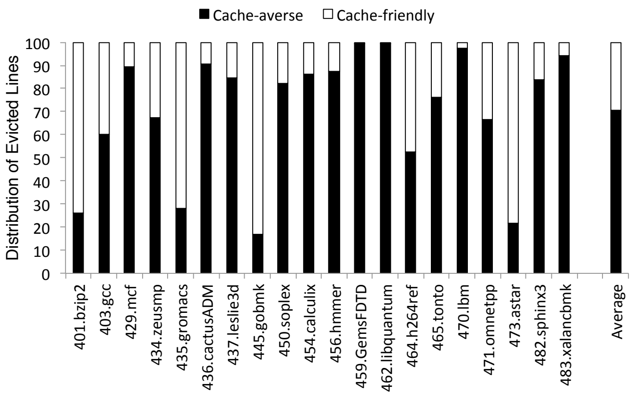


Figure 16: Distribution of evicted lines.

increase the number of cores from 1 to 4. We see that Hawkeye's relative benefit over SDBP increases with more cores. We also see that the gap between SHiP and SDBP diminishes at higher core counts.

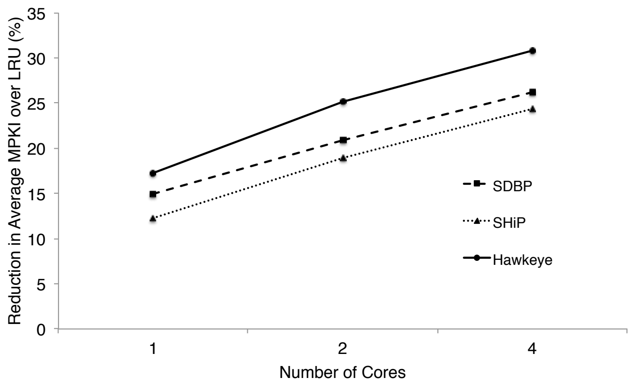


Figure 18: Miss reduction over LRU for 1, 2, and 4 cores.

Scheduling Effects. A key challenge in learning the caching behavior of multi-core systems is the variability that can arise from non-deterministic schedules. Thus, the optimal solution of the past memory access stream may not represent the optimal caching decisions for the future memory access stream. However, our evaluation shows that for multi-programmed workloads, the average bias of OPT's decisions is 89%, which explains why Hawkeye is beneficial for shared caches.

5. CONCLUSIONS

In this paper, we have introduced the Hawkeye cache replacement policy and shown that while it is impossible to look into the future to make replacement decisions, it *is* possible to look backwards over a sufficiently long history of past memory accesses to learn and mimic the optimal behavior.

The advantage of learning from OPT is that OPT can exploit reuse for any workload, so unlike existing policies, it is not focused on certain types of reuse—e.g., short-term and medium-term. This claim is supported by our empirical results: Unlike other policies, which for some workloads increase the number of cache misses (when compared against LRU), Hawkeye does not increase the number of cache misses for any of our evaluated workloads.

Conceptually, Belady's algorithm is superior to work that focuses on reuse distance, because Belady's algorithm directly considers both reuse distance *and* the demand on the cache. Concretely, by learning from OPT, Hawkeye provides significant improvements in miss reductions and in speedup for both single-core and multi-core settings.

More broadly, we have introduced the first method of providing an oracle for training cache replacement predictors. As with the trend in branch prediction, we expect that future work will enhance cache performance by using more sophisticated predictors that learn our oracle solution more precisely. Indeed, given the 99% accuracy with which OPTgen reproduces OPT's behavior, the greatest potential for improving Hawkeye lies in improving its predictor. Finally, we believe that Hawkeye's long history provides information that will be useful for optimizing other parts of the memory system, including thread scheduling for shared memory systems, and the interaction between cache replacement policies and prefetchers.

Acknowledgments.
We thank John Colby, Curtis Dunham, Don Fussell, Ibrahim Hur, Daniel Jimenez, Hao Wu, and the anonymous referees for their valuable feedback on early drafts of this paper. We also thank Aamer Jaleel for providing access to

his implementation of SHiP. This work was funded in part by NSF grants CNS-1138506 and DRL-1441009 and by a gift from the Qualcomm Foundation.

6. REFERENCES

[1] A. R. Alameldeen, A. Jaleel, M. Qureshi, and J. Emer. 1st JILP workshop on computer architecture competitions (JWAC-1) cache replacement championship. 2010.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, pages 78–101, 1966.

[3] L. A. Belady and F. P. Palermo. On-line measurement of paging behavior by the multivalued MIN algorithm. *IBM Journal of Research and Development*, 18:2–19, 1974.

[4] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, pages 401–412, 2009.

[5] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 389–400, 2012.

[6] H. Gao and C. Wilkerson. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *JWAC 2010-1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[7] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *International Conference on Supercomputing*, pages 338–347, 1995.

[8] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *International Symposium on Computer Architecture (ISCA)*, pages 107–116, 2000.

[9] 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.

[10] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. Cmp$im: A Pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[11] 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.

[12] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *International Symposium on Computer Architecture (ISCA)*, pages 60–71. ACM, 2010.

[13] D. A. Jiménez. Insertion and promotion for tree-based PseudoLRU last-level caches. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 284–296, 2013.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990.

[15] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *25th International Conference on Computer Design (ICCD)*, pages 245–250, 2007.

[16] 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.

[17] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 61–68, 2005.

[18] C. S. Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, pages 1352–1361, 2001.

[19] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *the 27th International Symposium on Computer Architecture (ISCA)*, pages 139–148, 2000.

[20] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 222–233, 2008.

[21] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD Record*, pages 297–306. ACM, 1993.

[22] 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.

[23] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, pages 895–913, 1999.

[24] T. R. Puzak. *Analysis of Cache Replacement-algorithms*. PhD thesis, University of Massachusetts Amherst, 1985.

[25] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *International Symposium on Computer Architecture (ISCA)*, pages 381–391. ACM, 2007.

[26] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 167–178, 2006.

[27] 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.

[28] K. Rajan and R. Govindarajan. Emulating optimal replacement with a shepherd cache. In *the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 445–454, 2007.

[29] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *the ACM Conference on Measurement and Modeling Computer Systems (SIGMETRICS)*, pages 134–142, 1990.

[30] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *the 21st Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 355–366, 2012.

[31] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[32] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *ACM SIGMETRICS Performance Evaluation Review*, pages 122–133, 1999.

[33] 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.

[34] R. Subramanian, Y. Smaragdakis, and G. H. Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 385–396. IEEE Computer Society, 2006.

[35] Z. Wang, K. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Parallel Architectures and Compilation Techniques*, pages 199–208, 2002.

[36] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *High-Performance Computer Architecture, 2000*, pages 49–60, 2000.

[37] 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.

[38] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. PACMan: prefetch-aware cache management for high performance caching. In *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–453, 2011.

[39] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, pages 174–183, 2009.