**To Appear in ISCA 2018**
**Final Version In Preparation.**

# Rethinking Belady's Algorithm to Accommodate Prefetching

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 shows that in the presence of data prefetchers, cache replacement policies are faced with a large unexplored design space. In particular, we observe that while Belady's MIN algorithm minimizes the total number of cache misses—including those for prefetched lines—it does not minimize the number of *demand* misses. To address this shortcoming, we introduce Demand-MIN, a variant of Belady's algorithm that minimizes the number of demand misses at the cost of increased prefetcher traffic. Together, MIN and Demand-MIN define the boundaries of an important design space, with many intermediate points lying between them.

To reason about this design space, we introduce a simple conceptual tool, which we use to define a new cache replacement policy called Prefetch-Aware Hawkeye. Our empirical evaluation shows that for a mix of SPEC 2006 benchmarks running on a 4-core system with a stride prefetcher, Prefetch-Aware Hawkeye improves IPC by 7.7% over an LRU baseline, compared to 6.4% for the previous state-of-the-art. On an 8-core system, Prefetch-Aware Hawkeye improves IPC by 9.4% compared to 5.8% for the previous state-of-the-art.

## 1. INTRODUCTION

Although caches and data prefetchers have been around for decades [1, 2, 3], there has been surprisingly little research on the interaction between the two. Most such research focuses on identifying inaccurate prefetches so that they can be preferentially evicted. More recent work [4] also attempts to retain hard-to-prefetch lines, but we argue that much more can be done.

We start by asking the question, what is the optimal cache replacement policy if we assume the existence of a data prefetcher? It would seem natural to look to Belady's MIN algorithm [5] for guidance, since it provably minimizes the number of cache misses, which in turn minimizes memory traffic. However, in the face of a prefetcher, Belady's algorithm is incomplete because it ignores the distinction between prefetches and demand loads. Thus, it minimizes the total number of misses, including those for lines brought in by prefetches, but it does not minimize the number of demand misses.

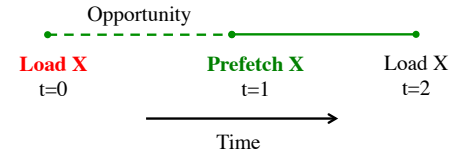


**Figure 1: Opportunity to improve upon MIN.**

As an alternative to MIN, this paper introduces Demand-MIN, a variant of Belady's algorithm that minimizes the number of demand misses at the cost of increased prefetcher traffic. Unlike MIN, which evicts the line that is reused furthest in the future, Demand-MIN evicts the line that is *prefetched* furthest in the future—and then falling back on MIN if no such line exists. For example, consider the accesses of line $x$ in Figure 1 (which ignores accesses to other lines). In the time interval between t=0 and t=1, Demand-MIN would allow line $x$ to be evicted, leaving more space to cache demand loads during this time interval. However, this improvement in demand misses comes with increased traffic, because the prefetch at time t=1 becomes a DRAM access instead of a cache hit. The reduction in demand misses can be significant: On a mix of SPEC 2006 benchmarks running on 4 cores, LRU yields an average MPKI of 29.8, MIN an average of 21.7, and Demand-MIN an average of 16.9.

What then is the optimal policy in terms of program performance? We observe that MIN and Demand-MIN define the extreme points of a design space, with MIN minimizing memory traffic, with Demand-MIN minimizing demand misses, and with the ideal replacement policy often lying somewhere in between. By plotting demand hit-rate (x-axis) against memory traffic (y-axis), Figure 2 shows that different SPEC benchmarks will prefer different policies within this space. Benchmarks such as astar (blue) and sphinx (orange) have lines that are close to horizontal, so they can enjoy the increase in demand hit rate that Demand-MIN provides while incurring little increase in memory traffic. By contrast, benchmarks such as tonto (light blue) and calculix (purple) have vertical lines, so Demand-MIN increases traffic but provides no improvement in demand hit rate. Finally, the re-

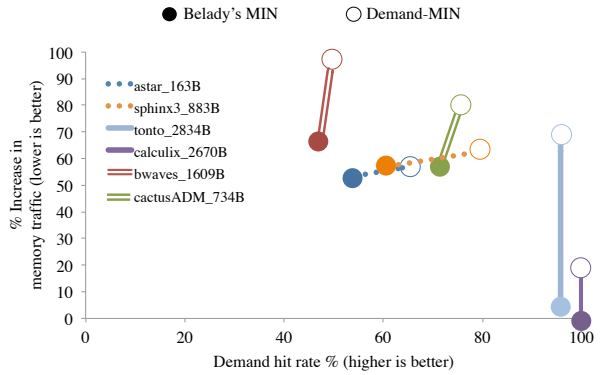


**Figure 2: With prefetching, replacement policies face a tradeoff between demand hit rate and prefetch traffic.**

maining benchmarks (bwaves and cactus) present tradeoffs that depend on the available bandwidth.

Unfortunately, it is difficult to identify the optimal policy for a given workload. First, for a given workload, it is difficult to know what the demand-hit-rate vs. increase-in-traffic curve looks like (the curves in Figure 2 were produced by simulating both the MIN and Demand-MIN solutions). Second, even if we had the curves, for the non-extreme cases, such as bwaves and cactus, it would be difficult to identify the most desirable point on the curve, because unlike miss rates, performance depends on many factors, such as the criticality of loads. Finally, even if we knew the desirable point on the curve, it would be difficult to identify lines that should be evicted to reach that point.

To navigate the space between MIN and Demand-MIN, this paper defines a simple new metric, Prefetches Evicted per Demand-Hit (PED), which serves as a proxy for the slope of the curves in Figure 2. This metric allows a policy to dynamically select a point in the space between MIN and Demand-MIN that is appropriate for a given workload. The result is Flex-MIN, a variant of MIN that is parameterized to represent different solutions within the space between MIN and Demand-MIN.[1]

Of course, Demand-MIN, Flex-MIN, and MIN are impractical because they rely on knowledge of the future, but the Hawkeye Cache [6] shows how Belady's MIN algorithm can be used in a practical setting: The idea is to train a PC-based predictor that learns from the decisions that MIN would have made on past memory references; Hawkeye then makes replacement decisions based on what the predictor has learned. In this paper, we use the architecture and machinery of the Hawkeye Cache (along with a small amount of added hardware to measure PED values), but instead of learning from Belady's MIN algorithm, our policy learns from Flex-MIN. The result is a new policy that we call Prefetch-Aware Hawkeye (PA-Hawkeye).

This paper makes the following contributions:

- We recognize that Belady's MIN algorithm is not ideal in the face of prefetching, and we introduce the Demand-MIN algorithm, which minimizes the number of demand misses. Together, MIN and Demand-MIN bracket a rich space of replacement policies.
- Because different workloads prefer different points in this design space, we introduce the Flex-MIN policy, which uses the notion of Prefetches Evicted per Demand-Hit to select an appropriate point in the space for a given workload.
- We encapsulate these ideas in a practical replacement policy, PA-Hawkeye, that in the presence of prefetching significantly improves upon the state-of-the-art. Using the ChampSIM simulation infrastructure [7] and multi-programmed SPEC 2006 benchmarks, we show that on 4 cores, PA-Hawkeye improves IPC over LRU by 7.7%, compared with 6.4% for an optimized version of the PACMAN policy [4]. On 8 cores, PA-Hawkeye sees 9.4% improvement over LRU, while optimized PACMAN sees 5.8% improvement.

The important new ideas in this paper are concentrated in Section 3, where we present the Demand-MIN policy and the PED metric. Section 4 explains the largely straightforward implementation of these ideas in the Hawkeye architecture, and Section 5 evaluates our solution. Related Work resides in its customary position in Section 2.

## 2. RELATED WORK

We now put our work in the context of prior work. We start by discussing variants of Belady's algorithm and follow with a discussion practical replacement policies.

### 2.1 Variants of Belady's MIN

In 1966, Belady proposed an algorithm to determine hits and misses under optimal cache replacement [5]. Mattson et al., proposed a different algorithm for optimal replacement [8], along with the first proof of optimality for a cache replacement algorithm. In 1974, Belady's MIN and Mattson's OPT were proven to be identical [9]. More recently, Michaud proposed a new way to reason about the optimal solution [10] and proved interesting mathematical facts about the optimal policy. None of this work consider prefetches.

Variants of MIN in the presence of prefetching typically focus on defining an optimal prefetching schedule assuming future knowledge. For example, Cao et al., propose two strategies for approaching an optimal caching and prefetching strategy [11, 12] for file systems. Temmam et al. [13] modify Belady's MIN to generate an optimal prefetch schedule that exploits both temporal and spatial locality. Our work differs by focusing on the cache replacement policy while assuming that the prefetcher remains fixed.

Finally, Jeong and Dubois [14] address Belady's assumption that all misses have uniform cost, presenting an exponential time cost-aware caching algorithm.

### 2.2 Practical Replacement Solutions

We now discuss practical cache replacement solutions, starting with prefetch-aware replacement policies, which is the subject of this paper. We then discuss advances in

[1] Despite its name, Flex-MIN is not optimal in any theoretical sense since it is built on a PED, which is a heuristic.

prefetch-agnostic cache replacement solutions and their implications for cache replacement in the presence of prefetching. Finally, we discuss solutions that modify the prefetcher to improve cache efficiency.

*Prefetch-Aware Cache Replacement.*

Unlike PA-Hawkeye, which focuses on accurate prefetches, most prior work in prefetch-aware cache management focuses on minimizing cache pollution due to inaccurate prefetches. Several solutions [15, 16] use prefetcher accuracy to decide the replacement priority of prefetched blocks. Ishii et al., instead use the internal state of the AMPM prefetcher [17] to inform the insertion priority of prefetched blocks [18]. PACMan [4] uses set dueling [19, 20] to determine whether prefetches should be inserted with high or low priority. The KPC [7] co-operative prefetching and caching scheme uses accuracy feedback to decide whether incoming prefetches are likely to be useful, and it uses timeliness feedback to determine the level of the cache hierarchy at which to insert the prefetch. PA-Hawkeye deals with inaccurate prefetches by using a PC-based predictor to learn the solution of Flex-MIN, which, like all variants of MIN, always discards inaccurate prefetches, since they are always reused furthest in the future.

The main goal of PA-Hawkeye is to evict lines that will be accurately prefetched in the future. PACMan [4] deprioritizes prefetch-friendly lines by not updating their insertion priority when they receive a hit due to a prefetch, but there are two fundamental differences between PA-Hawkeye and PACMan. First, PACMan does not consider the tradeoff between hit rate and traffic as it uniformly deprioritizes all prefetch-friendly lines, resulting in large traffic overheads (see Section 5.3). Second, because PACMan is triggered only on prefetches that hit in the cache, PACMan handles one of the three classes of references that we define in Section 3. By contrast, PA-Hawkeye learns from the past behavior of Flex-MIN to aggressively assign low priority to both demands and prefetches that are likely to be prefetched again. Section 5.3 provides a more detailed quantitative analysis of these differences as we observe that PACMan improves SHiP's performance by only 0.3% on four cores.

Finally, Seshadri et al., claim that prefetches are often dead after their first demand hit [16], so they propose that prefetched blocks be demoted after their first hit. While this strategy is likely to be effective for streaming workloads, it does not generalize to complex workloads and sophisticated prefetchers. Our optimized version of SHiP+PACMan includes this optimization, and we observe that it provides a 0.2% performance improvement on four cores.

*Advances in Cache Replacement.*

Much of the research in cache replacement policies has been prefetcher-agnostic [21, 22, 23, 24, 25, 26, 27, 28, 29, 20, 30, 31, 32, 33, 34, 35, 36, 37, 38].

Many replacement polices observe the reuse behavior for cache-resident lines to modulate their replacement priority [30, 28, 27, 35, 36, 39, 40, 41, 29], but by not distinguishing between demand loads and prefetches, such solutions are susceptible to cache pollution and are likely to mistake hits due to prefetches as a sign of line reuse.

Adaptive replacement policies [42, 19, 20] dynamically select among competing replacement policies and have the advantage that they can be easily modified to use different replacement schemes for prefetches to avoid cache pollution. For example, PACMan uses Set Dueling [20] to evaluate whether prefetches should be inserted with high or low priority. Unfortunately, such solutions treat all prefetches the same, assuming that they are all accurate or all inaccurate.

Recent solutions [32, 31, 6] use load instructions to learn the past caching behavior of demand accesses. Such solutions can be very effective in mitigating cache pollution if they distinguish between demands and prefetches and if a load instruction is provided for each prefetch request. For example, they can learn that prefetches loaded by a certain PC are more likely to be inaccurate than prefetches loaded by a different PC. Our SHiP+PACMan baseline builds on SHiP [32], and our solution (PA-Hawkeye) builds on Hawkeye [6], which both use PC-based predictors to provide replacement priorities for prefetches. One of the main contributions of this paper is the use of a PC-based predictor to determine whether a demand or a prefetch is likely to be prefetched and hence should be inserted with a low priority.

Finally, there are replacement solutions that revise their replacement decisions as they gather more information [37, 43, 23]. Existing solutions in this category do not distinguish between demands and prefetches.

*Prefetcher-Centric Solutions.*

Finally, there are solutions that reduce cache pollution by improving prefetcher accuracy [44, 45, 46, 47], and by dynamically controlling the prefetcher's aggressiveness [15, 48, 49, 50, 51, 52]. Such solutions are orthogonal to cache replacement and are likely to benefit from replacement policies that intelligently balance prefetch-friendly and hard to prefetch lines. Section 3.4 discusses the ways that prefetcher accuracy and coverage affect PA-Hawkeye.

## 3. DEMAND-MIN AND FLEX-MIN

This section defines our new Demand-MIN policy. We first develop intuition by showing a concrete example of how we can improve upon the MIN policy. We then describe the Demand-MIN policy, followed by the Flex-MIN policy.

### 3.1 Limitations of Belady's MIN algorithm

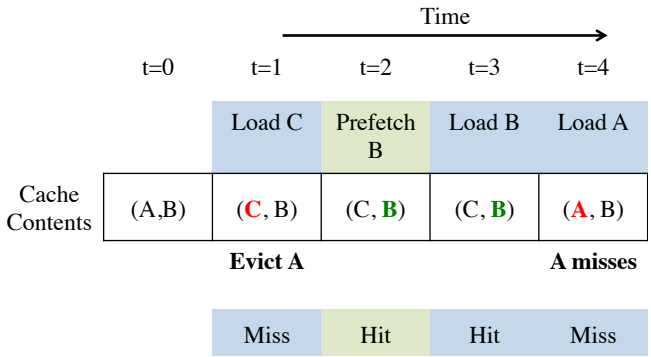


**Figure 3: Belady's MIN results in 2 demand misses.**

To see that Belady's MIN algorithm does not minimize demand misses, consider Figure 3, which shows an access sequence with demand accesses shown in blue and prefetch accesses shown in green.

For a cache that can hold 2 lines and initially holds lines *A* and *B*, we see that Belady's MIN algorithm produces two misses. The first miss occurs at t=1, when line *C* is loaded into a full cache. The MIN algorithm would evict *A* which is reused further in the future than *B*. The resulting cache contains lines *C* and *B*, so the prefetch to line *B* at time t=2 and the demand reference to *B* at time t=3 both hit in the cache. The second miss occurs at time t=4 when we load *A*.

Since MIN is optimal, we cannot do better than its two misses, but we *can* reduce the number of demand misses. The key observation is that *B* will be prefetched at time t=2, so the demand reference to *B* at t=3 will hit irrespective of our decision at time t=1, so we can decrease the number of demand misses as shown in Figure 4, where at time t=1, we evict *B* instead of *A*.

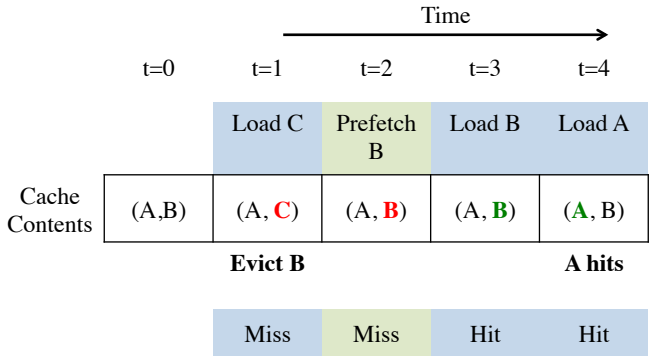


**Figure 4: Demand-MIN results in 1 demand miss.**

As a result, the prefetch to *B* at t=3 misses in the cache. The subsequent demand reference to *B* at t=3 still hits, but *A* now hits at t=4, which yields one more demand hit than Belady's MIN algorithm. Thus, this new caching strategy still results in 2 misses, but it exchanges a prefetch hit[2] for a demand hit, resulting in just a single demand miss (to *C*).

In this simple example, our improvement in demand hits did not increase overall memory traffic, but it is, of course, possible to trade multiple prefetch hits for a single demand hit, which can lead to extra prefetch traffic.

Note that even if MIN ignored prefetches, it would not arrive at the solution shown in Figure 4; it would still evict *A* at time t=1 because the load to *A* (t=4) is further in the future than the load to *B* (t=3).

## 3.2 The Demand-MIN Algorithm

To minimize demand misses, we modify Belady's MIN algorithm as follows:

> Evict the line that will be prefetched furthest in the future, and if no such line exists, evict the line that will see a demand request furthest in the future.

[2]We define a prefetch hit to be a prefetch request that hits in the cache and is not sent to memory.

In the example in Figure 4, we see that at time t=1, this policy will evict *B*, which is prefetched furthest in the future.

Intuitively, Demand-MIN preferentially evicts lines that do not need to be cached because they can be prefetched in the future. In particular, Demand-MIN creates room for demand loads by preferentially evicting the following three classes of accesses:

- *Repeated prefetches*: We define a *repeated prefetch* to be the second of a pair of prefetches of the same line without an intervening demand load.
- *Prefetch-friendly lines*: Lines that will be accurately prefetched in the future (Figure 5(b)) do not need to be cached to receive the subsequent demand hit. To understand the importance of evicting these lines, see the Venn diagram in Figure 6: For single-core SPEC 2006 benchmarks, 31.3% of lines are both cache-friendly and prefetch-friendly, and 34.6% are neither cache-friendly nor prefetch-friendly. Demand-MIN improves hit rate by evicting lines in the intersection of the Venn diagram (shown in dark gray) to make room for lines that lie outside both circles (shown in white).
- *Dead intervals*: Inaccurate prefetches of dead blocks (Figure 5(c)) create spurious demand on the cache.

The dashed lines in Figure 5 also illustrate a simple way to identify lines that can be evicted to improve hit rate. If we define the time period between two consecutive references to X to be X's *usage interval* [6], then there are four types of usage intervals—P-P, D-P, D-D, and P-D—depending on whether the endpoints are prefetches (P) or demand accesses (D). With this terminology, we see that Demand-MIN's benefit comes from evicting intervals that end with a prefetch, ie, the P-P intervals (Figure 5(a)) and D-P intervals (Figure 5(b) and (c)). For brevity, we collectively refer to P-P and D-P intervals as *-P intervals.

### 3.2.1 *On the Optimality of Demand-MIN*

The proofs of MIN's optimality are rather lengthy [8, 53, 54, 55, 56], so rather than provide a formal proof of the optimality of Demand-MIN, we instead give an informal argument.

The intuition behind Belady's MIN algorithm is simple: When faced with a set of eviction candidates, MIN chooses the one that is referenced furthest in the future because (1) the *benefit* of caching any of these candidates is the same, namely, the removal of one cache miss in the future, but (2) the *opportunity cost* is higher for lines accessed further in the future, since they occupy the cache for a longer period of time. Thus, MIN evicts the line with the largest opportunity cost, which is the line that is reused furthest in the future.

But if the goal is to minimize the number of demand misses, then we need to distinguish between lines that are next referenced by a demand load and lines that are next referenced by a prefetch. The caching of the former will reduce the number of demand misses by one, whereas the caching of the latter will not. Thus, Demand-MIN preferentially caches lines that are next referenced by demand loads over those that are next referenced by prefetches. And among lines referenced by prefetches, it again evicts the line

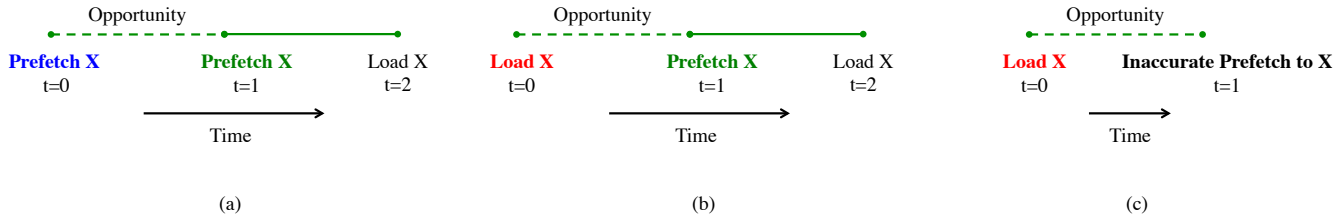


**Figure 5: Demand-MIN increases demand hit by evicting 3 classes of cache accesses.**

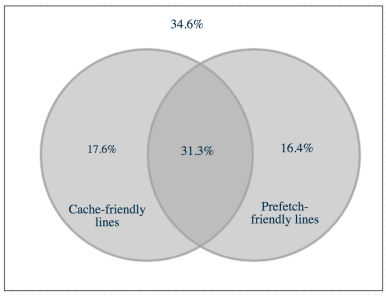


**Figure 6: Demand-MIN increases demand hit rate by using space allocated to prefetch-friendly lines (dark Gray) to instead cache hard-to-prefetch lines (white space).**

that is prefetched furthest in the future, because that line has the largest opportunity cost.

## 3.3 The Flex-MIN Algorithm

To realize a better tradeoff between demand hit rate and traffic, we introduce the notion of a *protected prefetch*, which is the endpoint of a *-P usage interval that should be cached because its eviction would generate traffic without providing considerable benefit in terms of hit rate. We then further modify Demand-MIN as follows:

> Evict the line that is prefetched furthest in the future and is not *protected*. If no such line exists, default to MIN.

Thus, Flex-MIN explores the design space between MIN and Demand-MIN. Flex-MIN is equivalent to MIN if all prefetches are protected, and it is equivalent to Demand-MIN if no prefetches are protected.

### *3.3.1 Protected Prefetches*

Protected prefetches are difficult to identify because their classification depends on both application characteristics and available bandwidth, so we define a heuristic for identifying protected prefetches.

Our definition of protected prefetches is based on two observations. First, it is more profitable to evict long *-P intervals than short *-P intervals because both generate 1 prefetch request (corresponding to the end of the interval), but their benefit in terms of freed cache space is proportional to the interval length. Thus, long *-P intervals free up more cache space per unit of extra traffic. Second, it is profitable to evict *-P intervals only if there are demand requests that can use the extra cache space. Even if there are many demand misses, they may require significant amounts of cache space to be freed before they become cache-friendly.

Thus, we define a protected prefetch to be a prefetch that lies at the end of a *-P usage interval whose length is below a given threshold. To determine the threshold, we compute the ratio of the average length of demand miss intervals to the average length of cache-friendly *-P intervals. We call this ratio Prefetches Evicted per Demand-Hit (PED). Intuitively, the ratio is a proxy for the slope of the line shown in Figure 2. A smaller ratio represents greater opportunity per unit traffic, and a large ratio indicates less opportunity per unit traffic.

As we will explain in Section 4, these ratios can be computed using a simple extension to Hawkeye.

## 3.4 Impact of the Prefetcher

Before moving on to a practical solution, we first discuss the impact that different prefetchers will have on Flex-MIN.

*Prefetcher Coverage.*

The design space introduced by Demand-MIN—and thus the utility of PA-Hawkeye—improves as prefetcher coverage increases, because higher coverage introduces more opportunities for Flex-MIN to evict prefetch-friendly lines and to create space for cache-averse demands.

*Prefetcher Accuracy.*

Inaccurate prefetches have a marginal effect on Flex-MIN, since the main idea behind Flex-MIN is to evict lines that can be accurately prefetched. A more accurate prefetcher will reduce the opportunity for PA-Hawkeye to evict dead intervals, but we find that these form only a small portion of all *-P intervals. Of course, lower accuracy requires the replacement policy to evict lines that it expects to be inaccurate, but modern replacement strategies, including both Hawkeye [6] and SHiP [32], already do this.

*Multiple Levels of Prefetching.*

Prefetchers can bring data into any of multiple levels of the cache. These decisions do not impact the design space defined by Demand-MIN, but from an implementation perspective, they have a huge impact on how this design space is explored. For example, if prefetches are being inserted into both the L1 cache and the last-level cache (LLC), then the LLC might not observe the intermediate demand access that are filtered by the L1 cache. Thus, replacement policies that rely on observing intermediate demand accesses between consecutive prefetches [16] will be unable to adapt to such scenarios. PA-Hawkeye is insensitive to this issue, as it focuses on *-P intervals and therefore does not need to

observe intermediate demand accesses.

# 4. PREFETCH-AWARE HAWKEYE

Like MIN, Flex-MIN is impractical because it requires knowledge of the future, but the recently proposed Hawkeye replacement policy [6] shows how MIN can be used as part of a practical replacement policy. In this section, we explain how Hawkeye can be modified to use Flex-MIN instead of MIN. Because Flex-MIN derives from MIN, straightforward changes to Hawkeye's basic implementation are sufficient to navigate the complex design space introduced by prefetches.

For those unfamiliar with Hawkeye, we first provide some necessary background.

## 4.1 Background: Hawkeye

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 [6], 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 into the cache, while lines that are predicted to be cache-averse are inserted with low priority.

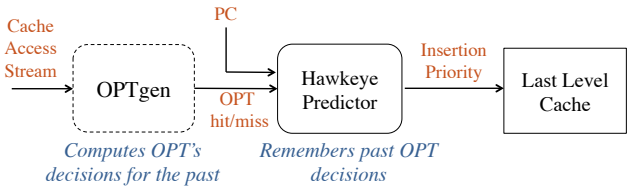


**Figure 7: Overview of the Hawkeye Cache.**

Figure 7 shows the overall structure of Hawkeye. Its main components are the Hawkeye Predictor, which makes insertion decisions, and OPTgen, which simulates OPT's behavior to produce inputs that train the Hawkeye Predictor.

*OPTgen.*

OPTgen determines what would have been cached if the OPT policy (MIN) had been used. The key insight behind OPTgen is that for a given cache access to line $x$, the optimal decision can be made when $x$ is next reused, because any later reference will be further in the future and would be a better eviction candidate for Belady's algorithm. Thus, OPTgen computes the optimal solution by assigning cache capacity to lines in the order in which they are reused.

To define OPTgen, Jain and Lin define a *usage interval* to be the time period that starts with a reference to some line $x$ and proceeds up to (but not including) its next reference, $x'$. If there is space in the cache to hold $x$ throughout the duration of this usage interval, then OPTgen determines that the reference to $x'$ would be a hit under Belady's policy.

For example, consider the sequence of accesses in Figure 8, which shows $x$'s usage interval. Here, assume that the cache capacity is two and that OPTgen has already determined the $a$, $b$, and $c$ can be cached. Since these intervals never overlap, the maximum number of overlapping liveness intervals in $x$'s usage interval never reaches the cache capacity, so there is space for line $x$ throughout the interval, and OPTgen infers that the load of $x'$ would be a hit.

OPTgen can be implemented efficiently in hardware using set sampling [39] and a simple vector representation of the usage intervals [6].

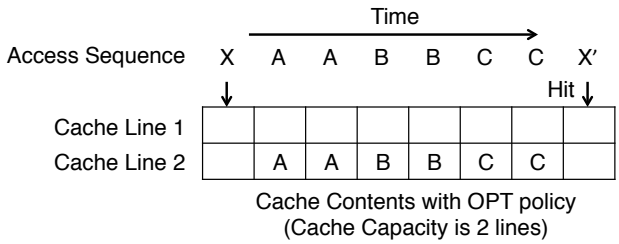


**Figure 8: Intuition behind OPTgen.**

*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 per core, it uses 5-bit counters for training, and it is indexed by a hash of the PC.

*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 high priority, i.e., an RRIP value [29] 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.

## 4.2 Prefetch-Aware Hawkeye

PA-Hawkeye modifies Hawkeye by learning from Flex-MIN instead of MIN. We first describe how we modify OPTgen to simulate Flex-MIN for past accesses, and we then describe changes to the Hawkeye predictor that allow it to better learn Flex-MIN's solution. Hawkeye's insertion and promotion policies remain unchanged.

*FlexMINgen.*

FlexMINgen determines what would have been cached if the Flex-MIN policy had been used. To simulate Flex-MIN, we modify OPTgen to distinguish between demands and prefetches, specifically, between *-P intervals (D-P and P-P) and *-D intervals (D-D and P-D).

FlexMINgen considers caching *-P intervals only if they

are shorter than some given *threshold*. As explained in Section 3.3, it is beneficial to cache short *-P intervals to avoid significant increase in prefetch traffic. FlexMINgen does not cache *-P intervals that are longer than the threshold so that the cache space can be used to cache a *-D interval. For *-D intervals, FlexMINgen follows the same policy as OPTgen.

The threshold for *-P intervals is computed dynamically. In particular, the threshold increases linearly with the PED metric, which we define to be the ratio of the average length of demand miss intervals to the average length of cache-friendly *-P intervals. We empirically find that the threshold should be set to 2.5 times the PED value.

To compute PED, we calculate the average length of demand miss intervals and the average length of cache-friendly *-P intervals using the following four counters.

- $DemandMiss_{total}$: This counter tracks the total length of all demand miss intervals. For every *-D interval that is determined to be a miss by FlexMINgen, this counter is incremented by the length of the interval.
- $DemandMiss_{count}$: This counter tracks the number of demand miss intervals and is incremented by 1 for every *-D interval that is determined to be a miss by FlexMINgen.
- $Supply_{total}$: This counter tracks the total length of all cache-friendly *-P intervals. For every *-P interval, FlexMINgen is probed to see if the *-P interval would have hit in the cache; if the answer is yes, this counter is incremented by the length of the *-P interval. This counter is incremented for all cache-friendly *-P intervals irrespective of their length.
- $Supply_{count}$: This counter tracks the number of cache-friendly *-P intervals and is incremented by 1 for every cache-friendly *-P interval.

In a multi-core environment, we compute the PED for each core and set each core's threshold individually. Since the length of usage intervals increases as the cache observes interleaved accesses from multiple cores, the threshold is scaled with the core count.

*PA-Hawkeye Predictor.*

The PA-Hawkeye predictor learns FlexMINgen's solution for past accesses. It differs from the Hawkeye predictor in two ways. First, we use separate predictors to learn FlexMIN's behavior for demands and prefetches. Second, to allow the predictor to learn that long *-P intervals should not be cached, the predictors are trained negatively when *-P intervals of length greater than the *threshold* are encountered. In particular, when we encounter a D-P interval that is longer than the threshold, we negatively train the demand predictor for the PC that loaded the left endpoint of the interval. Similarly, when we encounter a P-P interval that is longer than the threshold, we negatively train the prefetch predictor for the PC that loaded the left endpoint of the P-P interval.

*Hardware Overhead.*

PA-Hawkeye adds 32 bytes of hardware to Hawkeye. In particular, it needs four counters to compute the PED (see Section 4.2). While the counters can be updated using addition operations, While updating the counters requires simple addition operations, computing the PED involves an expensive division operation to compute the ratio between the length of demand intervals and the length of cache-friendly *-P intervals. We find that the division operation does not need to be precise and can be approximated using shift operators without any any effect on PA-Hawkeye's performance.

| | |
|---|---|
| L1 I-Cache | 32 KB 8-way, 4-cycle latency |
| L1 D-Cache | 32 KB 8-way, 4-cycle latency |
| L2 Cache | 256KB 8-way, 8-cycle latency |
| LLC per core | 2MB, 16-way, 20-cycle latency |
| DRAM | 13.5ns for row hits<br>40.5ns for row misses<br>800MHz, 12.8 GB/s |
| Single-core<br>Four-core<br>Eight-core | 2MB shared LLC<br>8MB shared LLC<br>16MB shared LLC |

**Table 1: Baseline configuration.**

# 5. EVALUATION

This section describes our empirical evaluation of FlexMIN and PA-Hawkeye, starting with our methodology.

## 5.1 Methodology

*Simulator.*

We evaluate our new policy using ChampSim [7], a trace-based simulator that includes an out-of-order core model with a detailed memory system. ChampSim models a 6-wide out-of-order processor with a 256-entry reorder buffer and a 3-level cache hierarchy. It models the memory effects of mispredicted branches and includes a perceptron-based branch predictor [57]. The simulator generates cache statistics as well as overall performance metrics, such as IPC.

The parameters for our simulated memory hierarchy are shown in Table 1. Caches include FIFO read and prefetch queues, with demand requests having priority over prefetch requests. MSHRs track outstanding cache misses, and if MSHRs are not available, further misses are stalled.

The L1 cache includes a next-line prefetcher, and the L2 cache includes a PC-based stride prefetcher. The L2 prefetcher can insert into either the L2 or the LLC. The prefetcher is invoked on demand accesses only. For our workloads, the L1 achieves 49% accuracy, while the L2 prefetcher achieves 51.9% accuracy. Together the prefetchers achieve 53% coverage.

The main memory is modeled in detail as it simulates data bus contention, bank contention, row buffer locality, and bus turnaround delays. The main memory read queue is processed out of order and uses a modified Open Row FR-FCFS policy. The DRAM core access latency for row hits is approximately 13.5ns and for row misses is approximately 40.5ns. Other timing constraints, such as tFAW and DRAM refresh, are not modeled.

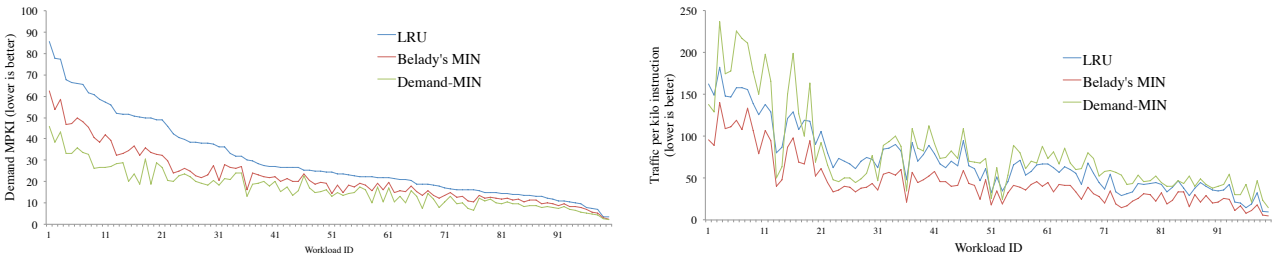


**Figure 9: Belady's MIN minimizes traffic, not Demand-MPKI (results for multi-programmed SPEC 2006)**

*Workloads.*

To stress the last-level cache, we use multi-programmed SPEC2006 benchmarks with 1, 4 and 8 cores. For 4-core results, we simulate 4 uniformly random benchmarks from the 20 most replacement-sensitive SPEC2006 benchmarks, and for 8-core results, we choose 8 uniformly random SPEC2006 benchmarks. For each individual benchmark, we use the reference input and trace the highest weighted SimPoint [58, 59]. Overall, we simulate 100 4-core mixes and 50 8-core mixes.

For each mix, we simulate the simultaneous execution of SimPoints of the constituent benchmarks until each benchmark has executed at least 1 billion instructions. To ensure that slow-running applications always observe contention, we restart benchmarks that finish early so that all benchmarks in the mix run simultaneously throughout the execution. Our multi-core simulation methodology is similar to the methodology used by recent work [29, 32, 31]. We warm the cache for 200 million instructions and measure the behavior of the next billion instructions.

*Metrics.*

To evaluate performance, we report the weighted speedup normalized to LRU for each benchmark combination. This metric is commonly used to evaluate shared caches [31, 60, 61, 22, 62] 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.

It is difficult to measure IPC for MIN and its variants, since they rely on knowledge of the future. We could apply the optimal decisions computed from a previous simulation to a re-run of the same simulation, but in a multi-core setting, this approach does not work because replacement decisions can alter the scheduling of each application, which will likely result in a different optimal caching solution.

Therefore, for MIN and its variants, we compute the average Demand MPKI of all applications in the mix; Demand MPKI is the total number of demand misses observed for every thousand instructions. To measure traffic overhead, we compute the overall traffic, including demand and prefetch misses, and we normalize it to every thousand instructions, yielding Traffic Per Kilo Instruction (TPKI).

*Baseline Replacement Policies.*

We compare PA-Hawkeye with two state-of-the-art replacement policies, namely, PACMan+SHiP [32, 4] and MultiPerspective Reuse Prediction (MPPPB) [62].

The PACMan insertion policy enhances existing replacement policies to account for prefetching by (1) avoiding cache pollution and (2) retaining cache lines that cannot be easily prefetched. We use as a baseline PACMan + SHiP, an optimized version of PACMan that uses SHiP [32] instead of PACMan's original DRRIP [29] policy. With respect to prefetches, PACMan + SHiP, which we obtained from its authors [63], includes several optimizations. (1) It uses a separate predictor to predict the insertion priority for prefetches on a miss. (2) It uses PACMan's policy of not updating the RRIP value on prefetch hits to allow prefetch-friendly lines to be evicted faster. (3) In the spirit of PACMan, when prefetches receive a subsequent demand hit, they are given a lower priority under the assumption that they are likely to be prefetched again [16].

MultiPerspective Reuse Prediction [62] uses sophisticated learning techniques to combine multiple features to predict whether a block is dead. The resulting replacement policy, called MultiPerspective Placement, Promotion and Bypass (MPPPB), outperforms Hawkeye in the presence of prefetching [62]. Its holistic approach to cache management makes it difficult to combine with PACMan.

For all replacement policies, we normalize the results to a baseline that uses LRU.

## 5.2 Demand-MIN and Flex-MIN

We first evaluate Demand-MIN and Flex-MIN. While these are unrealizable algorithms, we can evaluate them in a post-mortem fashion to measure demand miss rate and prefetcher traffic.

*Demand-MIN.*

The benefits of Demand-MIN are shown in the left graph of Figure 9, which compares the MPKI of LRU, MIN, and Demand-MIN for 100 4-core mixes of the SPEC2006 benchmarks. While Belady's MIN provides significantly lower MPKI than LRU, Demand-MIN can achieve even lower MPKI. In particular, the average MPKI is 29.8 for LRU, 21.7 for the MIN algorithm, and 16.9 for Demand-MIN.

The cost of Demand-MIN is shown in the right graph of

Figure 9, which compares prefetch traffic for these same policies. We see that MIN achieves the lowest traffic of 45.4 requests per kilo instructions, while the average traffic for Demand-MIN is as high as 79.4 requests per kilo instructions. In fact, we see that the traffic-overheads of MIN-Demand typically exceeds that of LRU.

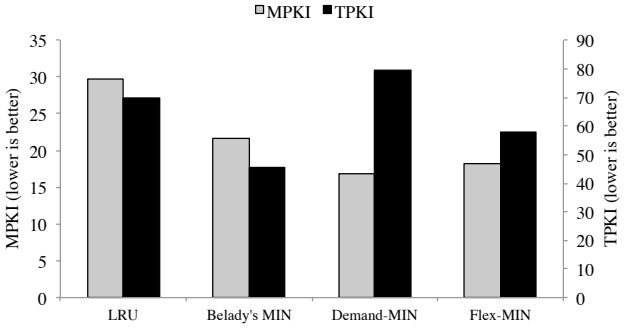


**Figure 10: Flex-MIN achieves a good tradeoff between MIN and Demand-MIN.**

*Flex-MIN.*

Figure 10 plots both average MPKI (on the left axis) and average Traffic Per Kilo Instruction (on the right axis), showing that Flex-MIN achieves an excellent tradeoff, as it approaches the benefits of Demand-MIN, and it approaches the traffic overhead of MIN. In particular, Flex-MIN's MPKI of 18.3 is closer to Demand-MIN's 16.9 than to MIN's 21.7 (and is much better than LRU's 29.8). Flex-MIN's TPKI of 58.1 is closer to MIN's 45.5 than to Demand-MIN's 79.4 (and is significantly better than LRU's 69.8).

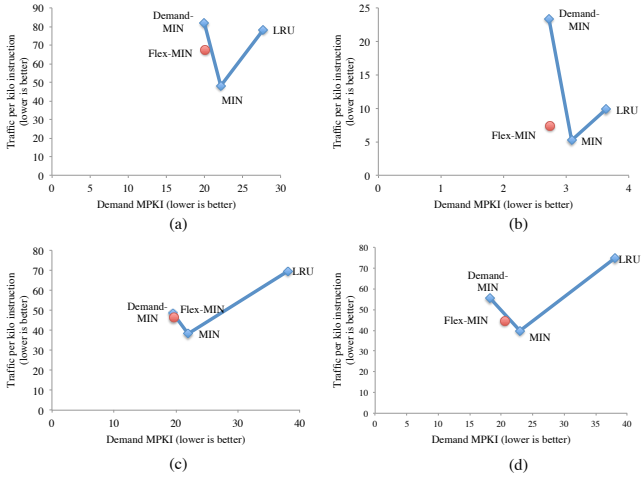


**Figure 11: Tradeoff between Belady's MIN and Demand-MIN is workload dependent.**

Figure 11 shows that the tradeoff between hit rate and prefetch traffic varies from one mix to another. For example, Figures 11(a) and (b) show mixes where Demand-MIN produces large traffic overheads without significant improvements in hit rate, so in these cases, a solution close to Belady's MIN is most desirable. We see that Flex-MIN picks attractive points for these mixes. In particular, for Figure 11(a), Flex-MIN picks a point midway between MIN and Demand-MIN, and for Figure 11(b), where the tradeoff is much more skewed, Flex-MIN picks a point very close to MIN. By contrast, Figures 11(c) and (d) show mixes where Demand-MIN achieves considerably better hit rates, and for these mixes, a solution closer to Demand-MIN is likely to be more desirable with respect to performance. We see that Flex-MIN makes good choices for these mixes as it picks a point close to Demand-MIN for Figure 11(c) and a point midway between MIN and Demand-MIN for Figure 11(d).

In general, we conclude that Flex-MIN is a good foundation upon which to build a practical cache replacement solution, which we now evaluate in the next section.

## 5.3 Prefetch-Aware Hawkeye

On a single-core system, PA-Hawkeye, PACMan+SHiP and MPPPB all improve performance by 2.5% over LRU.

On 4-cores, PA-Hawkeye outperforms both PACMan+SHiP and MPPPB, as shown in Figure 12(left). In particular, PA-Hawkeye improves performance by 7.7% over LRU, while PACMan+SHiP and MPPPB improve performance by 6.4% and 4.2%, respectively. The average MPKI for MPPPB, SHiP and PA-Hawkeye are 23.6, 22.6, and 21.9 respectively, and the average TPKI for the three policies are 76.2, 65.6, and 64.9 respectively.

On 8-cores (Figure 12(right)) PA-Hawkeye sees much larger performance improvements[3]: PA-Hawkeye improves performance by 9.4% over LRU, while PACMan+SHiP improves performance by 5.8%. PA-Hawkeye performs better because it reduces both average MPKI (18.8% vs. 22.6% for SHiP) and average TPKI (5.3% vs. −4.0% for SHiP).

Figure 13 shows that the performance gap between PA-Hawkeye and SHiP+PACMan grows with higher core count. There are two sources of PA-Hawkeye's advantage. First, in a multi-core system, cache management decisions taken by one core can significantly impact the cache performance of other cores. PA-Hawkeye considers the global impact of its decisions by solving Flex-MIN collectively for all applications instead of solving it for each core in isolation, so it successfully leverages the cache space freed by one application to improve hit rates for other applications. Second, as bandwidth becomes more constrained, it becomes more important to reason about the tradeoff between hit rate and traffic. Thus, the eviction of short *-P intervals has a larger performance impact in bandwidth-constrained environments.

## 5.4 Understanding PA-Hawkeye's Benefits

To understand PA-Hawkeye's performance advantage over PACMan+SHiP, Figure 14 compares PACMan+SHiP against vanilla SHiP and PA-Hawkeye against Hawkeye. These results show the performance gain that comes only from evicting prefetch-friendly lines. We see that PACMan only increases SHiP's performance from 6.1% to 6.4%, while the prefetch-aware aspects of PA-Hawkeye improve Hawkeye's performance from 6.3% to 7.7%. (PACMan does not combine well with Hawkeye, as it reduces Hawkeye's performance from 6.3% to 3.8%.)

[3]The MPPPB simulations for 8 cores did not finish in time.

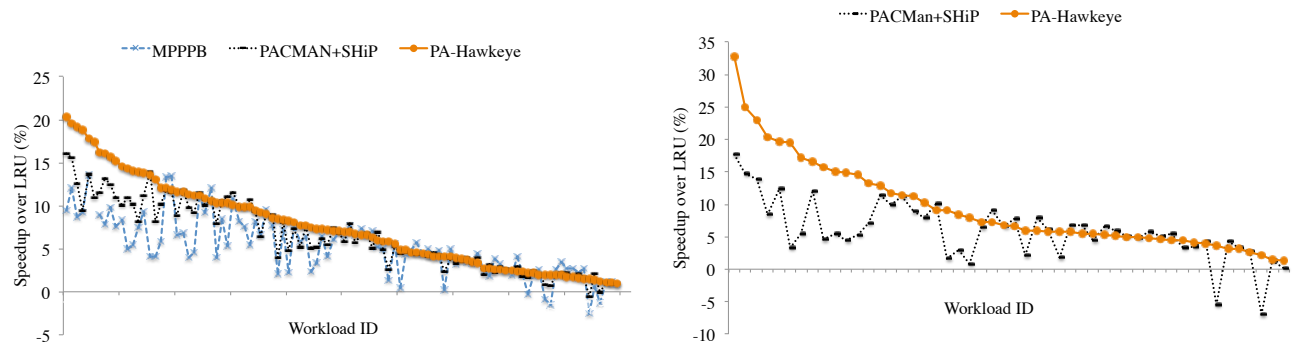


**Figure 12: PA-Hawkeye outperforms PACMan+SHiP and MPPPB on both 4 cores (left) and 8 cores (right).**

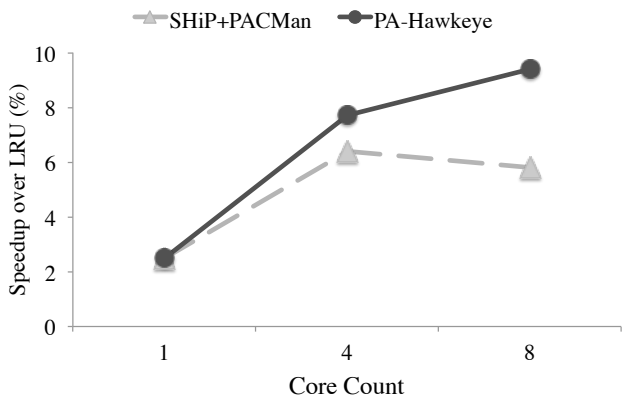


**Figure 13: PA-Hawkeye's advantage increases with more cores.**

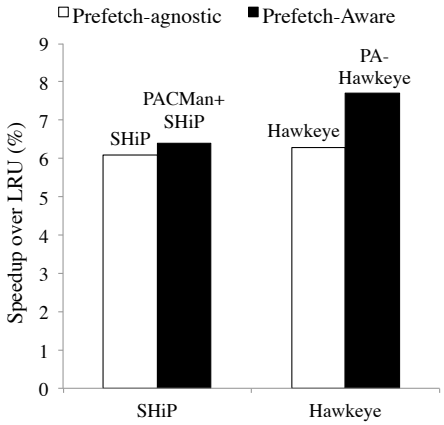


**Figure 14: PA-Hawkeye is more effective at retaining hard-to-prefetch lines than PACMan.**

Figure 15 sheds insight into PA-Hawkeye's advantage. The left graph plots PACMan+SHiP's MPKI reduction over vanilla SHiP on the x-axis and its traffic reduction over SHiP on the y-axis (each dot represents a workload mix). Similarly, the right graph plots PA-Hawkeye's MPKI reduction and traffic reduction over vanilla Hawkeye. Two observations are noteworthy. First, many blue dots in the left graph increase hit rate at the expense of significant traffic (bottom right quadrant). By contrast, only 1 of 100 red dots in the right graph resides in that quadrant. Second, if we focus on the positive cases in each graph, PA-Hawkeye explores a much larger part of the design space than PACMan+SHiP, as it can reduce absolute MPKI by up to 8 points and reduce traffic by up to 13 points. By contrast, PACMan + SHiP spans a smaller portion of the design with a maximum MPKI reduction of 5.6 and a maximum traffic reduction of 2.7.

The smaller expanse of the blue points in the left graph is not surprising, because of the three classes of references shown in Figure 5, PACMan only optimizes the first class. In particular, PACMan is triggered only when a prefetch hits in the cache, which means that its benefit is restricted to intervals that start with a prefetch.

The undesirable blue points in the left graph of Figure 15 can be explained by realizing that (1) PACMan does not consider the tradeoff between hit rate and traffic in evicting prefetch-friendly lines, so it is can increase prefetch traffic significantly for small improvements in demand hit rate, and (2) PACMan uniformly deprioritizes all prefetches that receive hits, so it can inadvertently evict useful P-D intervals.

## 5.5 Flex-MIN's Impact on Performance

Because MIN requires knowledge of the future, we cannot measure the IPC of it or its variants, but we *can* use Hawkeye as a tool to measure their impact indirectly by creating two new versions of PA-Hawkeye, one that learns from MIN and another that learns from Demand-MIN. We call these versions PA-Hawkeye-MIN and PA-Hawkeye-Demand-MIN. Figure 16 shows that on 4 cores, PA-Hawkeye outperforms both PA-Hawkeye-MIN and PA-Hawkeye-Demand-MIN, achieving a performance improvement of 7.7% over LRU, while PA-Hawkeye-MIN and PA-Hawkeye-Demand-MIN improve performance by 6.3% and 4.5%, respectively. Not surprisingly, PA-Hawkeye achieves the best tradeoff between hit rate and traffic, reducing MPKI by 26.6% and reducing traffic by 9%. By contrast, PA-Hawkeye-Demand-MIN achieves the highest MPKI reduction of 26.6%, but it increases traffic by 4.4%, while PA-Hawkeye-MIN reduces traffic by 6.4%, but it reduces MPKI by only 19.9%.

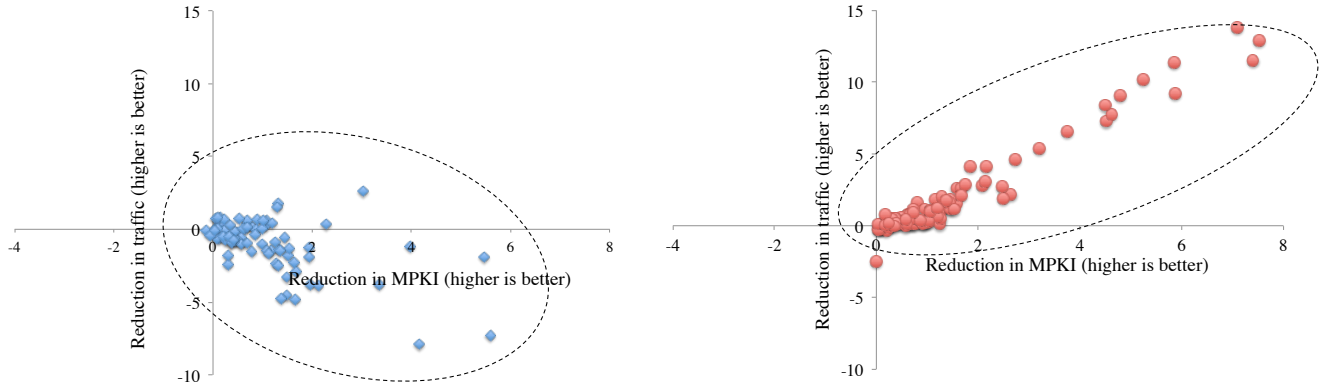


**Figure 15: Our scheme (right) explores a larger and more attractive part of the design space than PACMan (left).**

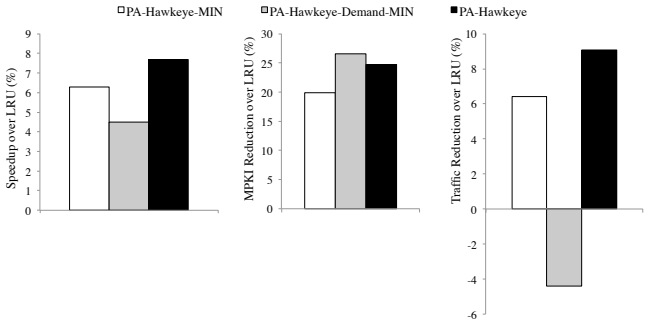


**Figure 16: Three Versions of PA-Hawkeye.**

*PA-Hawkeye Predictor Accuracy.*

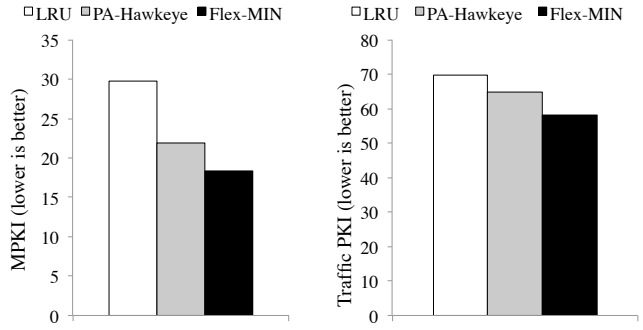


**Figure 17: A better predictor will shrink the gap between PA-Hawkeye and Flex-MIN.**

While PA-Hawkeye outperforms SHiP+PACMan, as shown in Figure 17, it does not match the miss reduction and traffic of Flex-MIN. In particular, Flex-MIN's average MPKI is 18.3, while PA-Hawkeye's average MPKI is 21.9. At the same time, Flex-MIN's traffic overhead in TPKI is 58.1, while PA-Hawkeye's TPKI is 64.9. Thus, not only does Flex-MIN have lower MPKI and TPKI, it achieves an overall better tradeoff between hit rate and traffic.

Since PA-Hawkeye learns from Flex-MIN, we conclude that the performance gap stems from inaccuracy in PA-Hawkeye's predictor, which ranges from 80-90%, with an average across all workloads of 87%. Thus, we conclude that to match Flex-MIN, improvements in PA-Hawkeye predictor's accuracy are critical. Since an inaccuracy of 13% results in a significant gap between PA-Hawkeye and Flex-MIN, we expect even small accuracy improvements to have a big impact on performance.

## 6. CONCLUSIONS

Data caches and data prefetchers have been mainstays of modern processors for decades, and while there has been considerable work in modulating memory traffic from the perspective of a prefetcher, we have shown in this paper that the cache replacement policy can also play a role in modulating memory traffic. In particular, we have introduced a new cache replacement policy that selectively *increases* memory traffic—in the form of extra prefetch traffic—to reduce the number of demand misses in the cache.

In particular, we have identified a new design space that resides between Belady's MIN algorithm and our new Demand-MIN algorithm. We have then shown that the best solution often resides somewhere between the two extreme points, depending on the workload. Finally, we have introduced the Flex-MIN policy, which uses the notion of *-P intervals to find desirable points within this design space.

Finally, we have shown how the Hawkeye Cache can be modified to use Flex-MIN instead of MIN, yielding PA-Hawkeye. Our results show that PA-Hawkeye explores a larger design space than PACMan+SHiP and that our solution scales well with the number of cores: For a mix of SPEC2006 benchmarks running on 8 cores, PA-Hawkeye achieves an average speedup over LRU of 9.4%, compared to 5.8% for PACMan+SHiP.

## 7. REFERENCES

[1] A. J. Smith, "Cache memories," *ACM Computing Surveys*, pp. 473–530, 1982.

[2] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pp. 124–131, 1983.

[3] 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)*, pp. 364–373, 1990.

[4] 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)*, pp. 442–453, 2011.

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

[6] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *43rd International Symposium on Computer Architecture (ISCA)*, June 2016.

[7] 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 *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 737–749, ACM, 2017.

[8] R. Mattson, J. Gegsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

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

[10] P. Michaud, "Some mathematical facts about optimal cache replacement," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 50, 2016.

[11] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," in *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, pp. 188–197, 1995.

[12] T. Kimbrel and A. R. Karlin, "Near-optimal parallel prefetching and caching," *SIAM Journal on computing*, vol. 29, no. 4, pp. 1051–1082, 2000.

[13] O. Temam, "An algorithm for optimally exploiting spatial and temporal locality in upper memory levels," *IEEE transactions on computers*, vol. 48, no. 2, pp. 150–158, 1999.

[14] J. Jeong and M. Dubois, "Cache replacement algorithms with nonuniform miss costs," *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 353–365, 2006.

[15] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pp. 63–74, 2007.

[16] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, p. 51, 2015.

[17] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," in *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–24, 2011.

[18] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 267–278, ACM, 2012.

[19] 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)*, pp. 167–178, 2006.

[20] 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)*, pp. 381–391, ACM, 2007.

[21] 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)*, pp. 401–412, 2009.

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

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

[24] 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.

[25] 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*, pp. 199–208, 2002.

[26] 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)*, pp. 139–148, 2000.

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

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

[29] 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)*, pp. 60–71, ACM, 2010.

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

[31] 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)*, pp. 175–186, 2010.

[32] 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)*, pp. 430–441, 2011.

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

[34] 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)*, pp. 222–233, 2008.

[35] 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)*, pp. 134–142, 1990.

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

[37] 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*, pp. 355–366, 2012.

[38] 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*, pp. 338–347, 1995.

[39] 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)*, pp. 544–555, 2005.

[40] 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*, pp. 297–306, ACM, 1993.

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

[42] 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)*, pp. 385–396, IEEE Computer Society, 2006.

[43] N. Beckmann and D. Sanchez, "Maximizing cache performance under uncertainty," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 109–120, IEEE, 2017.

[44] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2013.

[45] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 623–634, IEEE, 2014.

[46] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *ISCA*, pp. 69–80, 2009.

[47] P. Diaz and M. Cintra, "Stream chaining: exploiting multiple levels of correlation in data prefetching," in *ISCA*, pp. 81–92, 2009.

[48] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *IEEE PACT*, pp. 135–145, 2004.

[49] X. Zhuang and H.-H. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," in *International Conference on Parallel Processing*, pp. 286–293, 2003.

[50] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62–73, 1992.

[51] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 316–326, 2009.

[52] B. Panda, "Spac: A synergistic prefetcher aggressiveness controller for multi-core systems," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3740–3753, 2016.

[53] L. A. McGeoch and D. D. Sleator, "A strongly competitive randomized paging algorithm," *Algorithmica*, vol. 6, pp. 816–825, 1991.

[54] A. Cohen and W. Burkhard, "A proof of optimality of the MIN paging algorithm using linear programming duality," *Operations Research Letters*, vol. 18, pp. 7–13, August 1995.

[55] B. V. Roy, "A short proof for the optimality of the MIN paging algorithm," *Information Processing Letters*, vol. 102, pp. 72–73, April 2007.

[56] M.-K. Lee, P. Michaud, J. S. Sim, and D. Nyang, "A simple proof for the optimality of the MIN cache replacement policy," *Information Processing Letters*, vol. 116, pp. 168–170, February 2016.

[57] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7$^{th}$ Int'l Symposium on High Performance Computer Architecture*, pp. 197–206, January 2001.

[58] 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)*, pp. 318–319, 2003.

[59] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[60] 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*, pp. 208–219, 2008.

[61] 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*, pp. 234–244, 2000.

[62] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pp. 436–448, 2017.

[63] A. Jaleel and M. Qureshi, "Personal communication,"