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 framework, which we use to define a new cache replacement policy called Harmony. Our empirical evaluation shows that for a mix of SPEC 2006 benchmarks running on a 4-core system with a stride prefetcher, Harmony improves IPC by 7.7% over an LRU baseline, compared to 3.7% for the previous state-of-the-art. On an 8-core system, Harmony improves IPC by 9.4% compared to 4.4% for the previous state-of-the-art.

***Keywords*-Caches; Prefetching; Cache Replacement**

I. Introduction

Although caches and data prefetchers have been around for decades [44], [10], [21], 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 [51] also attempts to retain all hard-to-prefetch lines, but we argue that there exists a much richer space of possible solutions to explore.

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 [3] 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.

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 falls 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, freeing up cache space for other demand loads. Demand-MIN's 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. However, this improvement in demand misses often comes with increased traffic. For example, in our example above, Demand-MIN turns the prefetch at time t=1 from a cache hit (under the MIN policy) into a DRAM access.

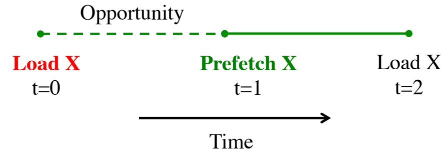


Figure 1. Opportunity to improve upon MIN.

What then is the best 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 remaining benchmarks (bwaves and cactus) present less obvious tradeoffs.

Unfortunately, there are two difficulties in identifying the

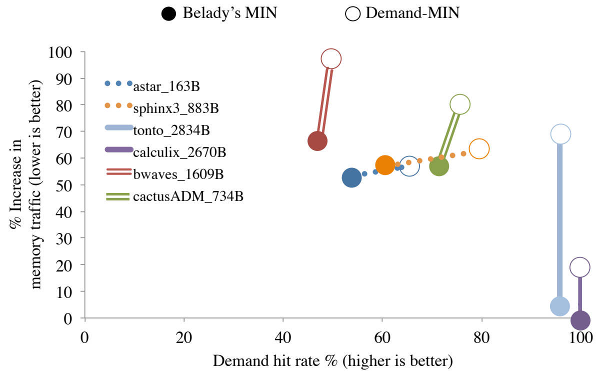


Figure 2. With prefetching, replacement policies face a tradeoff between demand hit rate and prefetcher traffic.

best policy for a given workload. First, it is difficult to know what the workload's 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 knew the point on the curve that offered the best performance, it would be difficult to design a replacement policy that would evict lines to reach that point.

To navigate the space between MIN and Demand-MIN, this paper defines a simple new metric, Lines Evicted per Demand-Hit (LED), 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 [15] 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 LED 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 Harmony.

This paper makes the following contributions:

- We demonstrate 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 solutions.
- Because different workloads prefer different points in this design space, we introduce the Flex-MIN policy, which uses the notion of Lines Evicted per Demand-Hit (LED), described in Section IV-B, to select an appropriate point in the space for a given workload.
- We encapsulate these ideas in a practical replacement policy, called Harmony, that in the presence of prefetching significantly improves upon the state-of-the-art. Using the ChampSIM simulation infrastructure [24], [1] with a Stride prefetcher and multi-programmed SPEC 2006 benchmarks, we show that on 4 cores, Harmony improves IPC over LRU by 7.7%, compared with 3.7% for an optimized version of the PACMan policy [51]. On 8 cores, Harmony sees 9.4% improvement over LRU, while optimized PACMan sees 4.4% improvement.
- We show that Demand-MIN, Flex-MIN, and Harmony provide benefits for a variety of prefetchers. We also provide insights and empirical results that characterize the varying impact that prefetchers have on the tradeoff between MIN and Demand-MIN.

This paper is organized as follows. The important new ideas are concentrated in Section III, where we present the Demand-MIN policy, and in Section V-G, which provides insights into the impact of prefetchers on Demand-MIN and Flex-MIN. Section IV explains the largely straightforward implementation of these ideas in the Hawkeye architecture, and Section V evaluates our solution. Related Work resides in its customary position in Section II.

II. 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 of practical replacement policies.

A. *Variants of Belady's MIN*

Belady's 1966 paper [3] presents the MIN cache replacement algorithm for minimizing cache misses. Mattson et al., propose a different replacement algorithm [29], along with the first proof of optimality for cache replacement, but Belady and Palermo's 1974 paper proves that Belady's MIN and Mattson's OPT are identical [4]. More recently, Michaud presents a new way to reason about the optimal solution [32] and proves interesting mathematical facts about the optimal policy. None of this work considers prefetches.

Variants of MIN in the presence of prefetching typically focus on defining an optimal prefetching schedule based on future knowledge. For example, Cao et al., propose two strategies for approaching an optimal caching and prefetching strategy [5], [25] for file systems. Temam et al. [49] 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.

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

B. 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 prefetch-agnostic cache replacement solutions and their implications for prefetch-aware cache replacement. Finally, we discuss solutions that modify the prefetcher to improve cache efficiency.

Prefetch-Aware Cache Replacement: Most previous work in prefetch-aware cache replacement focuses on minimizing the cache pollution caused by inaccurate prefetchers. Several solutions [47], [43] use prefetcher accuracy to determine the replacement priority of prefetched blocks. Ishii et al., instead use the internal state of the AMPM prefetcher [12] to inform the insertion priority of prefetched blocks [13]. PACMan [51] uses set dueling [38], [37] to determine whether prefetches should be inserted with high or low priority. KPC [24] is a co-operative prefetching and caching scheme that uses feedback on prefetcher accuracy to determine whether incoming prefetches are likely to be useful, and it uses feedback on prefetcher timeliness to determine the level of the cache hierarchy at which to insert the prefetch.

Like these solutions, Harmony also deals with inaccurate prefetches: Flex-MIN, like all variants of MIN, discards inaccurate prefetches, because inaccurate prefetches are always reused furthest in the future. Since Harmony learns from Flex-MIN, Harmony learns to discard inaccurate prefetches at the PC granularity.

However, inaccurate prefetches represent just a small part of the Harmony strategy. The main goal of Harmony is to evict lines that will be prefetched in the future. PACMan [51] is the first work to state this goal, but PACMan uses a much simpler scheme: It refrains from increasing the insertion priority when a line receives a hit due to a prefetch. Thus, there are two fundamental differences between PACMan and Harmony. First, PACMan does not consider the tradeoff between hit rate and traffic but instead uniformly deprioritizes all prefetch-friendly lines, resulting in large traffic overheads (see Section V-C). Second, PACMan deals with *prefetched lines,* while Harmony deals with *lines that are likely to be prefetched in the future.* Thus, PACMan is triggered only on prefetches that hit in the cache, allowing PACMan to handle just one of the three classes of references that we define in Section III. By contrast, Harmony can assign low priority to both demand loads and prefetches that are likely to be prefetched again. Section V-C provides a more detailed quantitative analysis of these differences, but we find that on 4 cores, PACMan improves SHiP's performance by only 0.3%.

Seshadri et al. [43], claim that prefetches are often dead after their first demand hit, so their policy demotes prefetched blocks after their first hit. While this strategy is effective for streaming workloads, it does not generalize to complex workloads and sophisticated prefetchers. We find that for SHiP+PACMan, this optimization provides only a 0.2% performance improvement on four cores.

Finally, unlike KPC's replacement policy, which is explicitly co-designed to work only with its own prefetcher, Harmony is designed to work with any prefetcher.

Advances in Cache Replacement: Much of the research in cache replacement policies has been prefetcher-agnostic [52], [40], [9], [26], [17], [37], [18], [23], [50], [8], [2], [38], [48].

Many replacement policies observe the reuse behavior for cache-resident lines to modulate their replacement priority [18], [34], [27], [17], 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.

Recent solutions [50], [23], [15], [8], [18], [20] use load instructions to learn the past caching behavior of demand accesses. Such solutions can reduce cache pollution if they distinguish between demand loads 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 Harmony solution builds on Hawkeye [15], which uses a PC-based predictor to provide replacement priorities for prefetches. Our paper is the first to use a PC-based predictor to determine whether a demand load or a prefetch is likely to be prefetched again and to use this information to insert such lines with a low priority.

Prefetcher-Centric Solutions: Finally, there are solutions that reduce cache pollution by improving prefetcher accuracy [14], [22], [46] and by dynamically controlling the prefetcher's aggressiveness [47], [33], [53], [7], [35]. Such solutions are likely to benefit from replacement policies that intelligently balance prefetch-friendly and hard-to-prefetch lines. In particular, such solutions increase available bandwidth, providing more opportunities for policies such as Flex-MIN and Harmony to trade off traffic for improved demand hit rates.

III. 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.

A. Limitations of Belady's MIN algorithm

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

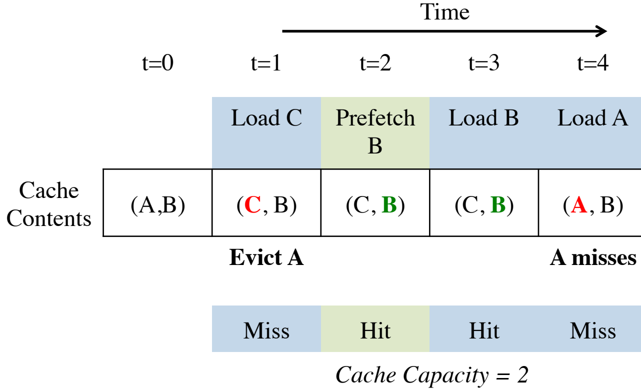


Figure 3. Belady's MIN results in 2 demand misses.

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*.

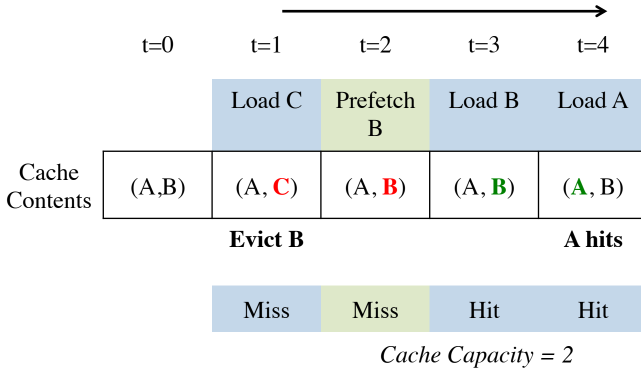


Figure 4. Demand-MIN results in 1 demand miss.

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 Demand-MIN 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).

B. The Demand-MIN Algorithm

This section describes the Demand-MIN algorithm, along with a conceptual framework for reasoning about the tradeoff between MIN and Demand-MIN.

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.

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 will be prefetched in the future. Figure 5 shows the three classes of accesses that Demand-MIN evicts to create room for demand loads:

- *Shadowed Lines*: We define a *shadowed line* to be a line that is prefetched into the cache and whose next access is a prefetch, as shown in Figure 5(a). The second prefetch can be either accurate or inaccurate.
- *Prefetch-Friendly Lines*: We define a *prefetch-friendly line* to be a line that is brought into the cache by a demand load and whose next access is an accurate prefetch. (See Figure 5(b)).
  These lines need not be cached to receive the subsequent demand hit because the prefetcher will fetch this line anyway. The importance of evicting these lines is illustrated by 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 Lines*: We define a *dead line* (Figure 5(c)) to be a line that is brought into the cache by a demand load and whose next access is an inaccurate prefetch. Dead lines represent spurious demand on the cache.

To reason about cache replacement in the presence of prefetches, we now extend Hawkeye's notion of a usage interval [15]—defined to be the time between two consecutive references to the same cache line[3]—to identify the endpoints as being either a demand access (D) or a prefetch (P). Thus

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

[3] For example, in Figure 5(a), the dashed line represents one usage interval.

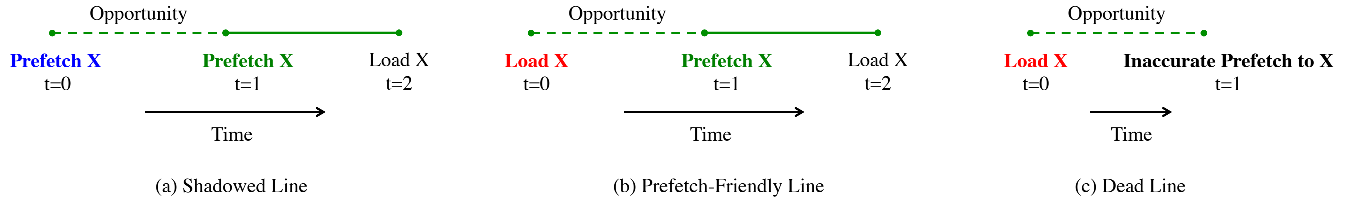


Figure 5. Demand-MIN increases demand hit by evicting 3 classes of cache accesses.

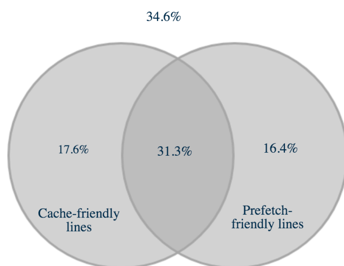


Figure 6. Demand-MIN increases demand hit rate by using space allocated to prefetch-friendly lines (dark Gray) to instead cache hard-to-prefetch lines (white space).

Usage Interval	Definition	Cacheable by MIN	Cacheable by Demand-MIN
$D-D$	Demand Reuse	✓	✓
$P-D$	Accurate Prefetch	✓	✓
$D-open$	Scan	×	×
$P-open$	Inaccurate Prefetch	×	×
$P-P_{accurate}$ $P-P_{inaccurate}$	Shadowed Line	✓	×
$D-P_{accurate}$	Prefetch-Friendly Line	✓	×
$D-P_{inaccurate}$	Dead Line	✓	×

Table I
UNLIKE MIN, DEMAND-MIN EVICTS ALL *-P INTERVALS.

there are four types of usage intervals, namely, P-P, D-P, D-D, and P-D. If we include open intervals, representing lines that are never reused, there are two more types: P-open and D-open.

Table I uses these extended usage intervals to illustrate the uniqueness of Demand-MIN. The first four rows show cases that are handled identically by MIN and Demand-MIN. The first two types of intervals (D-D and P-D) are cached by both MIN and Demand-MIN if there is room in the cache, and the last two types (D-open and P-open) are preferentially evicted by both MIN and Demand-MIN because they are never reused. Since these intervals can be handled correctly without distinguishing between demand accesses and prefetches, state-of-the-art replacement policies, such as Hawkeye and SHiP, can handle these types of intervals correctly.

Demand-MIN differs from MIN in its treatment of the usage intervals shown in the last three rows. These D-P and P-P intervals correspond to the three classes of accesses shown in Figure 5. MIN considers these intervals for caching even though they do not yield demand hits, but Demand-MIN evicts these intervals to make room for other intervals that can yield demand hits.

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. It is noteworthy that Demand MIN's benefit over MIN does not come from evicting inaccurate prefetches, which appear as open P-open intervals.

1) On the Optimality of Demand-MIN: The proofs of MIN's optimality are rather lengthy [29], [30], [6], [41], [28], 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 that is prefetched furthest in the future, because that line has the largest opportunity cost.

C. The Flex-MIN Algorithm

To realize a better tradeoff between demand hit rate and traffic, we introduce the notion of a *protected line*, which is a cache line that would be evicted by Demand-MIN but

not by Flex-MIN, because it would generate traffic without providing a significant improvement in 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 lines are protected, and it is equivalent to Demand-MIN if no lines are protected.

1) Protected Lines: *Protected lines* are lines that lie at the beginning of *-P intervals and that if evicted would be likely to increase traffic with little payoff in demand hit rate. Thus, Flex-MIN protects these lines from being evicted.

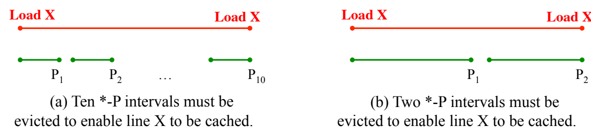


Figure 7. Protected lines can be identified by comparing the lengths of *-P intervals and D-D intervals.

For example, the left side of Figure 7 shows a scenario where ten *-P usage intervals (shown in green) would need to be evicted to enable a single D-D interval (shown in red) to be cached, clearly a poor tradeoff. By contrast, the right side of Figure 7 shows a scenario where only two *-P intervals need to be evicted to enable the D-D interval to be cached, a much better tradeoff that yields one additional demand hit at the cost of just two extra prefetches.

The examples in Figure 7 reveal two important insights. 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 their interval length. Second, the tradeoff between hit rate and traffic can be estimated by comparing the lengths of D-D intervals that can be potentially cached and the lengths of the *-P intervals that can be potentially evicted.

Thus, we define a **protected line** to be a line that lies at the beginning of a *-P usage interval whose length is below some threshold. As we will explain in Section IV, the threshold can be computed using two counters that track the average length of D-D intervals and *-P intervals.

IV. The Harmony Replacement Policy

This section explains how we employ Flex-MIN as part of a practical cache replacement policy called Harmony. Flex-MIN, like MIN, is impractical because it requires knowledge of the future, but the recently proposed Hawkeye replacement policy [15] shows how MIN can be used as part of a practical replacement policy, so Harmony takes the same approach, replacing MIN with Flex-MIN, with the key difference being the need to navigate the complex design space introduced by prefetches.

For those unfamiliar with Hawkeye, we first review its central ideas.

A. 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 [15], 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.

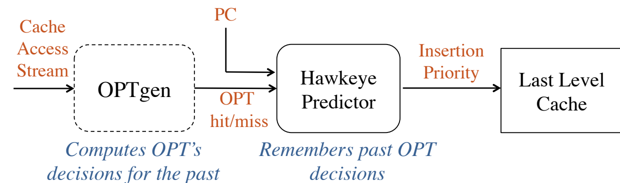


Figure 8. Overview of the Hawkeye Cache.

Figure 8 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 whether lines would have been cached if the optimal 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 therefore 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, we 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 9, which shows X's usage interval. Here, assume that the cache capacity is 2 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 [15].

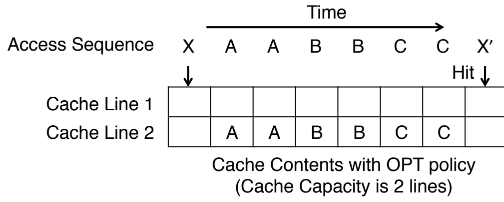


Figure 9. Intuition behind OPTgen.

The Hawkeye Predictor: The Hawkeye Predictor learns the behavior of the OPT policy as applied to a long history of previous 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 predicts 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 [17] 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.

B. Learning from Flex-MIN

Harmony 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. Harmony's insertion and promotion policies are identical to Hawkeye.

FlexMINgen: FlexMINgen determines the caching decisions of the Flex-MIN policy. To simulate Flex-MIN, we modify OPTgen to distinguish between demand loads and prefetches, specifically, between *-P intervals and *-D intervals.

FlexMINgen allows *-P intervals to be cached only if they are shorter than a given *threshold*, because as explained in Section III-C, the caching of short *-P intervals can avoid significant prefetcher traffic. For *-D intervals, FlexMINgen follows the same policy as OPTgen.

The threshold for caching *-P intervals is modulated dynamically by computing the ratio of the average length of D-D intervals that miss in the cache and the average length of *-P intervals that are cache-friendly. We call this ratio Lines Evicted per Demand-Hit (LED). Intuitively, the ratio is a proxy for the slope of the line shown in Figure 2 and approximates the average number of *-P intervals that need to be evicted to enable a D-D interval to be cached. We empirically find that the threshold should be set to 2.5 times the LED value, which means that as the slope increases, Flex-MIN will evict *-P intervals more conservatively.

To compute the LED value, Harmony uses 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, then 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 LED value 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 linearly with the core count.

Harmony Predictor: The Harmony predictor learns FlexMINgen's solution for past accesses. It differs from the Hawkeye predictor in two ways. First, it uses separate predictors to learn Flex-MIN's behavior for demand accesses 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, for a *-P interval that is longer than the threshold, the demand or prefetch predictor for the PC that loaded the left endpoint of the interval is negatively trained, and for a *-P interval that is shorter than the threshold, the demand or prefetch predictor is trained based on FlexMINgen's decision. We find that there is some benefit in tuning the aggressiveness with which the predictors are trained negatively for long *-P intervals.

Hardware Overhead: Harmony adds 32 bytes of hardware to Hawkeye (Hawkeye has a budget of 28KB [15]). In particular, it needs four counters to compute the LED value. The counters are updated using addition operations, and the division operation to compute the LED value is approximated using a shift operator.

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 40.5ns for row misses 800MHz, 3.2 GB/s for single-core, and 12.8 GB/s for multi-core
Single-core Four-core Eight-core	2MB shared LLC 8MB shared LLC 16MB shared LLC

Table II
BASELINE CONFIGURATION.

V. EVALUATION

This section describes our empirical evaluation of Demand-MIN, Flex-MIN and Harmony, as applied to the last-level cache (LLC).

A. Methodology

Simulator: We evaluate our new policy using ChampSIM [24], [1], 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 [19].

The parameters for our simulated memory hierarchy are shown in Table II. 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 prefetcher achieves 49% accuracy, while the L2 prefetcher achieves 63% accuracy. Together the prefetchers achieve 53% coverage. To study the impact of prefetchers, we also evaluate with two other state-of-the-art prefetchers, namely, the Best Offset Prefetcher (BO) [31] and the Access Map Pattern Matching (AMPM) Prefetcher [12].

The main memory is modeled in detail as it simulates data bus contention, bank contention, row buffer locality, and bus turnaround delays. Bus contention increases memory latency. 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.

Workloads: To stress the LLC, we use multi-programmed SPEC2006 benchmarks with 1, 4 and 8 cores. For 4-core results, we simulate 4 benchmarks chosen uniformly randomly from among the 20 most replacement-sensitive benchmarks, and for 8-core results, we choose 8 benchmarks chosen uniformly randomly. For each individual benchmark, we use the reference input and trace the highest weighted SimPoint [36], [11]. Overall, we simulate 100 4-core mixes and 50 8-core mixes.

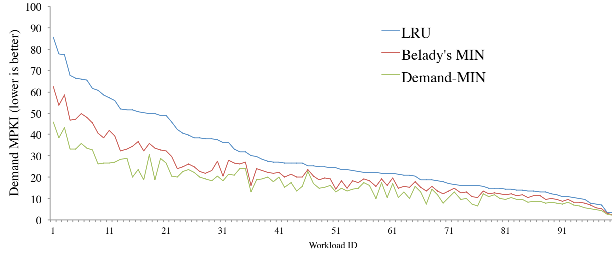
For each mix, we simulate the simultaneous execution of SimPoints of the constituent benchmarks until each benchmark has executed at least 1 billion instructions. To ensure that slow-running applications always observe contention, we restart benchmarks that finish early so that all benchmarks in the mix run simultaneously throughout the execution. 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 [23], [16], [45], [52], [20] 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 Harmony against PACMan [51], a state-of-the-art prefetch-aware cache replacement policy. We also use as a baseline PACMan + SHiP, an optimized version of PACMan that uses SHiP [50] instead of PACMan's original DRRIP [17] policy. With respect to prefetches, PACMan + SHiP includes two optimizations. (1) It uses a separate predictor to predict the insertion priority for prefetches on a miss. (2) To evict prefetch-friendly lines more quickly, it uses PACMan's policy of not updating the RRIP value on prefetch hits.



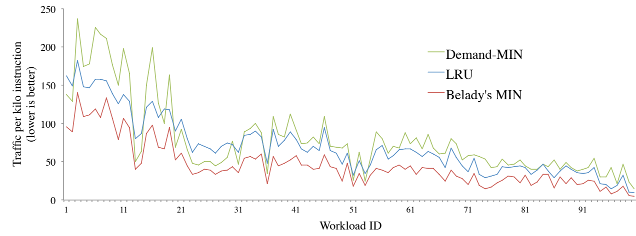


Figure 10. Belady's MIN minimizes traffic, not Demand-MPKI (results for multi-programmed SPEC 2006)

B. 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.

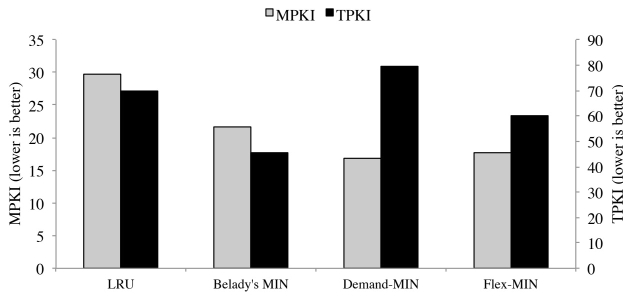


Figure 11. Flex-MIN achieves a good tradeoff between MIN and Demand-MIN.

Demand-MIN: The benefits of Demand-MIN are shown in the left graph of Figure 10, 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 (21.7 vs. 29.8), Demand-MIN achieves even lower MPKI of 16.9.

The cost of Demand-MIN is shown in the right graph of Figure 10, which compares prefetcher traffic for these same policies. We see that MIN achieves the lowest TPKI of 45.4, while the average TPKI for Demand-MIN is as high as 79.4. In fact, we see that the traffic-overheads of MIN-Demand typically exceed that of LRU.

Flex-MIN: Figure 11 plots both average MPKI (on the left axis) and average TPKI (on the right axis), showing that Flex-MIN achieves an excellent tradeoff, as it approaches the miss rates of Demand-MIN and the traffic overhead of MIN. In particular, Flex-MIN's MPKI of 17.7 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 60.1 is closer to MIN's 45.5 than to Demand-MIN's 79.4 (and is significantly better than LRU's 69.8).

Figure 12 shows that the tradeoff between hit rate and prefetcher traffic varies across mixes. For example, Figures 12(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, selecting a point midway between MIN and Demand-MIN in Figure 12(a) and selecting a point close to MIN in Figure 12(b), where the tradeoff is much more skewed.

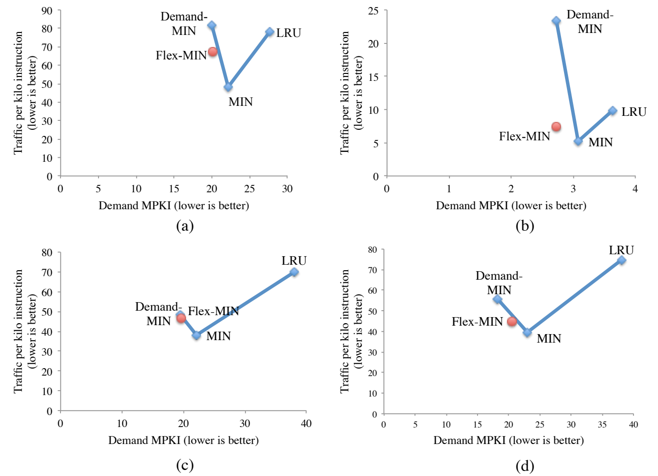


Figure 12. Tradeoff between Belady's MIN and Demand-MIN is workload dependent.

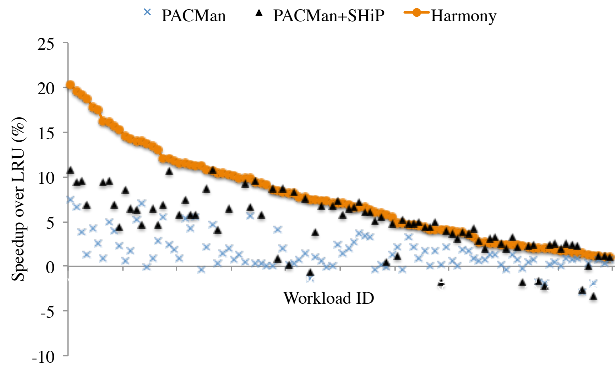
By contrast, Figures 12(c) and (d) show mixes where Demand-MIN achieves considerably better hit rates, and for these mixes, a solution closer to Demand-MIN is likely to give the best performance. Flex-MIN appropriately picks a point close to Demand-MIN in Figure 12(c) and a point midway between MIN and Demand-MIN in Figure 12(d).

From these results, we see that Flex-MIN is a good foundation upon which to build a practical cache replacement solution, which we now evaluate in the next section.

C. Harmony

On one core, Harmony improves performance by 3.3% over LRU, PACMan+SHiP by 2.4% and PACMan by 1.6%.

On 4 cores, Harmony outperforms both PACMan and PACMan+SHiP (see Figure 13(left)), improving perfor-



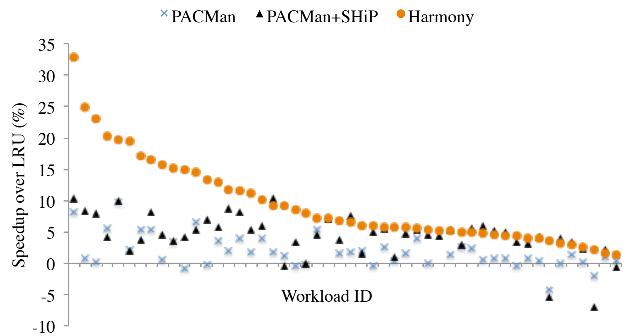


Figure 13. Harmony outperforms PACMan+SHiP and MPPPB on both 4 cores (left) and 8 cores (right).

mance by 7.7% over LRU, while PACMan and PACMan+SHiP improve performance by 1.5% and 3.7%, respectively. The average MPKI for PACMan, PACMan+SHiP and Harmony are 27.8, 22.9, and 21.9, respectively, and the average TPKI for the three policies are 68.0, 63.6, and 57.5.

On 8 cores (Figure 13(right)) Harmony sees a much larger performance improvement of 9.4% over LRU, while PACMan+SHiP improves performance by 4.4%. Harmony performs better because it reduces both average MPKI (41.7 vs. 43.3 for PACMan+SHiP) and average TPKI (57.4 vs. 63.6 for PACMan+SHiP).

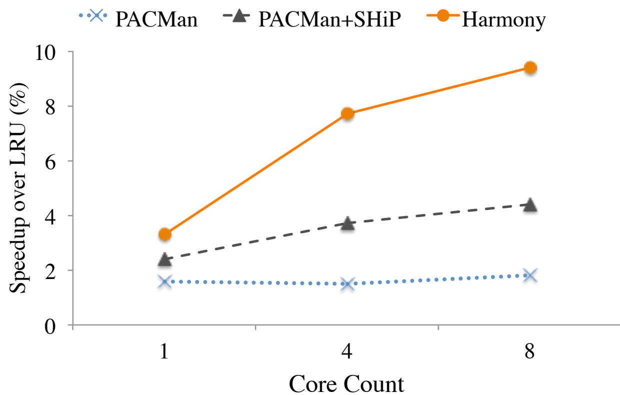


Figure 14. Harmony's advantage increases with more cores.

Figure 14 shows that the performance gap between Harmony and SHiP+PACMan grows with higher core count. There are two sources of Harmony's advantage. First, in a multi-core system, cache management decisions taken by one core can significantly impact the cache performance of other cores. Harmony 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, Flex-MIN's ability to reason about the tradeoff between hit rate and traffic becomes more important.

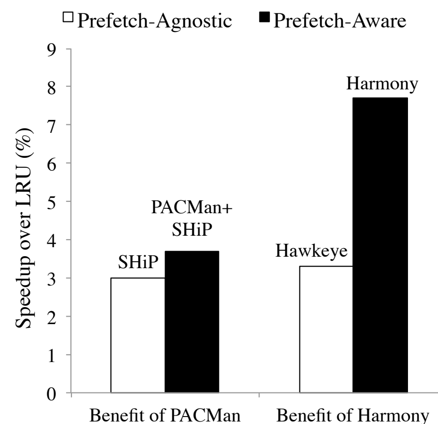
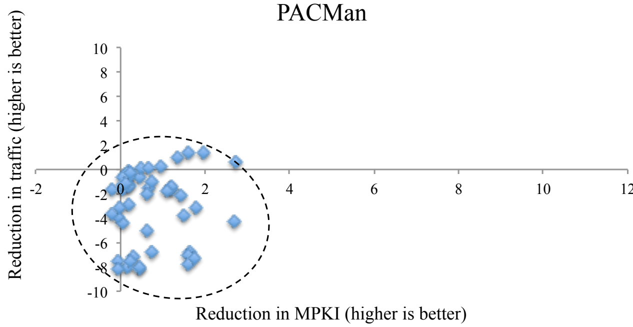


Figure 15. Harmony is more effective at retaining hard-to-prefetch lines than PACMan.

D. Understanding Harmony's Benefits

To isolate Harmony's and PACMan's ability to deal with prefetches, Figure 15 compares PACMan+SHiP against vanilla SHiP and compares Harmony against Hawkeye. These results show the performance gain that comes from (1) using a PC-based predictor for prefetch requests and (2) evicting prefetch-friendly lines. We see that PACMan only increases SHiP's performance from 3.0% to 3.7%, while Harmony improves Hawkeye's performance from 3.3% to 7.7%. We further observe that for Hawkeye, PC-based prediction alone achieves a 6.3% performance improvement, and the prefetch-aware aspects of Harmony further improve 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%.)

Figure 16 sheds insight into Harmony's advantage. The left graph plots PACMan+SHiP's MPKI reduction over vanilla SHiP on the x-axis and its traffic reduction over SHiP



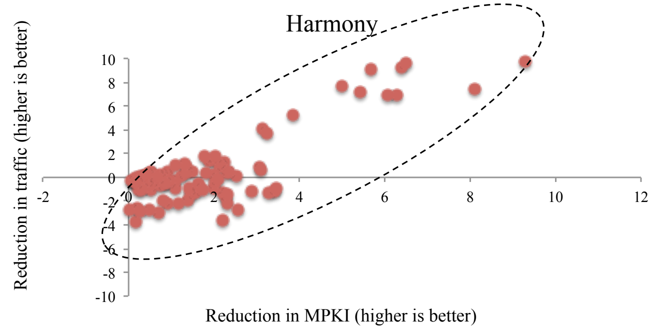


Figure 16. Our scheme (right) explores a larger and more attractive part of the design space than PACMan (left).

on the y-axis (each dot represents a workload mix). Similarly, the right graph plots Harmony's MPKI reduction and traffic reduction over Hawkeye. We make two observations. First, many blue dots in the left graph increase hit rate at the expense of significant traffic (bottom right quadrant). By contrast, fewer red dots in the right graph reside in that quadrant, and in general, the red dots in the bottom right quadrant provide a larger reduction in MPKI per unit traffic. Second, if we focus on the positive cases in each graph, Harmony explores a much larger part of the design space than PACMan+SHiP, as it can reduce absolute MPKI by up to 9 points and reduce traffic by up to 10 points, while PACMan + SHiP sees a maximum MPKI reduction of 2.7 and a maximum traffic reduction of 0.9.

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 16 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 prefetcher 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.

E. 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 Harmony as a tool to measure their impact indirectly by creating two new versions of Harmony, one that learns from MIN and another that learns from Demand-MIN. We call these versions Harmony-MIN and Harmony-Demand-MIN. Figure 17 shows that on 4 cores, Harmony outperforms both Harmony-MIN and Harmony-Demand-MIN, achieving a performance improvement of 7.7% over LRU, compared with Harmony-MIN's 6.3% and Harmony-Demand-MIN's

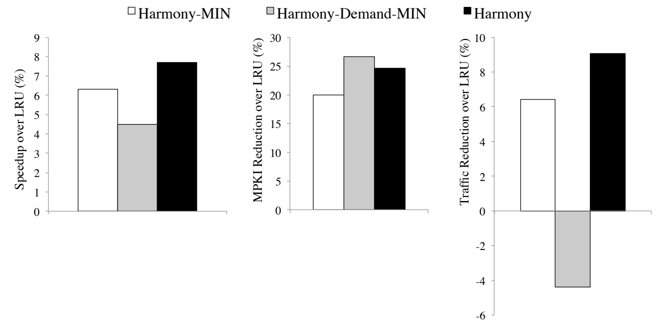


Figure 17. Three Versions of Harmony.

4.5%. Not surprisingly, Harmony achieves the best tradeoff between hit rate and traffic, reducing MPKI by 26.6% and reducing traffic by 9%. By contrast, Harmony-Demand-MIN achieves the highest MPKI reduction of 26.6%, but it increases traffic by 4.4%, while Harmony-MIN reduces traffic by 6.4% but reduces MPKI by only 19.9%.

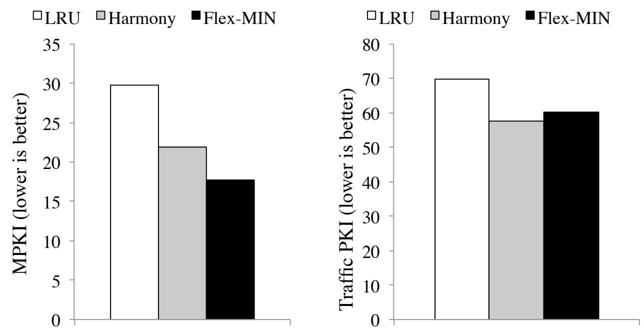


Figure 18. A better predictor will shrink the gap between Harmony and Flex-MIN.

Harmony Predictor Accuracy: While Harmony outperforms SHiP+PACMan (see Figure 18), it does not match the behavior of Flex-MIN: Flex-MIN's average MPKI is 17.7, while Harmony's is 21.9. And Flex-MIN's traffic overhead in TPKI is 60.1, while Harmony's is 57.5.

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

F. Comparison against CRC Entrants

We now compare Harmony against top submissions from the 2nd Cache Replacement Championship (CRC), using the two configurations that included a prefetcher [1]. An older version of Harmony won the championship, and the solution presented in this paper improves upon that version by exploring the design space between MIN and Demand-MIN more completely and systematically.

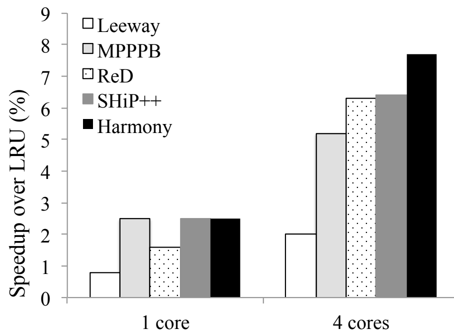


Figure 19. Harmony outperforms top submissions to the Cache Replacement Championship 2017. An older version of Harmony won the championship.

Figure 19 compares Harmony against four CRC submissions: SHiP++ (2nd place), ReD (3rd place), as well as MPPPB and Leeway, two solutions that have been previously compared against Hawkeye [20], [8]. For the single-core configuration, Harmony's performance is comparable to that of other submissions, but with 4 cores its benefit over other policies grows significantly.

To understand these results, observe that Harmony is the only policy that treats prefetches as a first-class concern; the others either treat demand accesses and prefetches identically or use existing heuristics [51], [42] to handle inaccurate prefetches. Such heuristics can be tuned to perform well in single-core scenarios, but they do not scale well to multi-core systems, where different cores observe different prefetcher accuracies, and where bandwidth contention adds complexity that is not addressed by these heuristics.

G. Impact of the Prefetcher

To explore the impact of the prefetcher on Demand-MIN, Flex-MIN, and Harmony, we now replace the baseline Stride prefetcher with the winners of the last two Data Prefetching Championship winners, namely, (1) the Access Map Pattern Matching prefetcher (AMPM) [12] and (2) the Best Offset Prefetcher (BO) [32]. These prefetchers provide a range of increasing coverage—63% for Stride, 71% for AMPM, and 73% for BO—and decreasing accuracy—83% for Stride, 75% for AMPM, and 68% for BO.

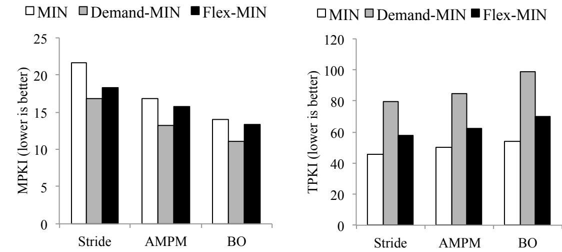


Figure 20. Demand-MIN reduces MPKI for both BO and AMPM.

Impact on Demand-MIN and Flex-MIN: Figure 20 shows that Demand-MIN improves MPKI over MIN for all three prefetchers. Flex-MIN continues to provide a good tradeoff between MIN and Demand-MIN; Flex-MIN's TPKI is closer to MIN's superior TPKI for all three prefetchers.

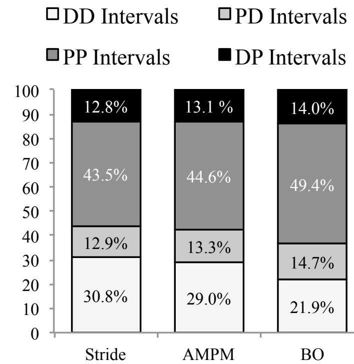


Figure 21. Distribution of D-D, P-D, D-P and P-P intervals for Stride prefetcher, BO and ISB for 2 workload mixes.

These results show that Demand-MIN is beneficial for prefetchers with different coverage and accuracy tradeoffs, but interestingly, Figure 21 shows that Demand-MIN makes different tradeoffs for each prefetcher, as the distribution of *-P and *-D intervals depends on the prefetcher's coverage and accuracy. In particular, BO, the most aggressive of the three prefetchers, has the highest proportion of *-P intervals, which allows Demand-MIN to evict a larger number of *-P intervals to accommodate hard-to-cache demand intervals. Thus, Demand-MIN also shows the highest increase in TPKI (83% over MIN) in Figure 20. By contrast, the Stride prefetcher, the most conservative of the three prefetchers, has the highest proportion of D-D intervals, providing the greatest opportunity to cache demand intervals in lieu of *-P

intervals. Thus, Demand-MIN's MPKI benefit (22.1% over MIN) is the largest for the Stride prefetcher. Finally, AMPM strikes a good balance between coverage and accuracy and sees a good balance between D-D intervals and *-P intervals.

In general, Demand-MIN generates more traffic for more aggressive prefetchers, such as BO, and it has greater opportunity to reduce Demand MPKI for more conservative prefetchers, such as the Stride prefetcher. Along these lines, Demand-MIN, and by extension Harmony, should have greater MPKI benefits in the presence of prefetcher throttling [47], [33], [53], [7], [35]. Flex-MIN's ability to navigate the space becomes more important for aggressive prefetchers where it becomes necessary to consider the cost of Demand-MIN's added traffic.

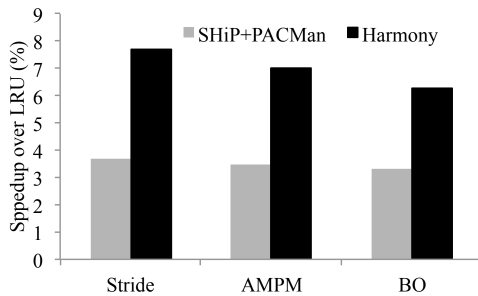


Figure 22. Harmony outperforms top submissions to the Cache Replacement Championship 2017. An older version of Harmony won the championship.

Impact on Harmony: Figure 22 shows that Harmony outperforms PACMan+SHiP for all three prefetchers, achieving a 6.3% speedup for BO (vs. 3.3% for PACMan+SHiP) and a 7.0% speedup for AMPM (vs. 3.5% for PACMan+SHiP).

VI. 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 prefetcher traffic—to reduce the number of demand misses in the cache.

More specifically, 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. We have then introduced the Flex-MIN policy, which uses the notion of LED values 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 the Harmony replacement policy. Our results show that Harmony 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, Harmony achieves an average speedup over LRU of 9.4%, compared to 4.4% for PACMan+SHiP.

Acknowledgments

We thank Daniel Jiménez, Tres Brenan, Don Fussell, and the anonymous referees for their valuable feedback on early drafts of this paper. This work was funded in part by NSF grant CNS-1543014 and by a gift from Oracle Labs.

References

[1] *2nd Cache Replacement Championship*, 2017. [Online]. Available: http://crc2.ece.tamu.edu/

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

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

[4] 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.

[5] 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*, 1995, pp. 188–197.

[6] A. Cohen and W. Burkhard, "A proof of optimality of the MIN paging algorithm using linear programming duality," *Operations Research Letters*, vol. 18, no. 1, pp. 7–13, August 1995.

[7] 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*, 2009, pp. 316–326.

[8] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, 2017, pp. 180–193.

[9] 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.

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

[11] 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.

[12] 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, 2011, pp. 1–24.

[13] 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*, 2012, pp. 267–278.

[14] 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.

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

[16] 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 (PACT)*, 2008, pp. 208–219.

[17] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010, pp. 60–71.

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

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

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

[21] 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)*, 1990, pp. 364–373.

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

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

[24] 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 Int'Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 737–749.

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

[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)*, 2000, pp. 139–148.

[27] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and 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.

[28] 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, no. 2, pp. 168–170, February 2016.

[29] 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.

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

[31] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.

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

[33] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: an adaptive data cache prefetcher," in *13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004, pp. 135–145.

[34] 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*, 1993, pp. 297–306.

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

[36] 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)*, 2003, pp. 318–319.

[37] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007, pp. 381–391.

[38] 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)*, 2006, pp. 167–178.

[39] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The V-Way cache: demand-based associativity via global replacement," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005, pp. 544–555.

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

[41] B. V. Roy, "A short proof for the optimality of the MIN paging algorithm," *Information Processing Letters*, vol. 102, no. 2-3, pp. 72–73, April 2007.

[42] 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 (PACT)*, 2012, pp. 355–366.

[43] 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.

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

[45] 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 (ASPLOS)*, 2000, pp. 234–244.

[46] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009, pp. 69–80.

[47] 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 (HPCA)*, 2007, pp. 63–74.

[48] 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)*, 2006, pp. 385–396.

[49] 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.

[50] 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)*, 2011, pp. 430–441.

[51] 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)*, 2011, pp. 442–453.

[52] Y. Xie and G. H. Loh, "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009, pp. 174–183.

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