Effective Mimicry of Belady's MIN Policy

Ishan Shah
The University of Texas at Austin, USA
ishan@cs.utexas.edu

Akanksha Jain
Google, USA[*]
akanksha@cs.utexas.edu

Calvin Lin
The University of Texas at Austin, USA
lin@cs.utexas.edu

Abstract—**The past decade has seen the rise of highly successful cache replacement policies that are based on binary prediction. For example, the Hawkeye policy learns whether lines loaded by a given PC are Cache Friendly (likely to remain in the cache if Belady's MIN policy had been used) or Cache Averse (likely to be evicted by Belady's MIN policy). In this paper, we instead present a cache replacement policy that is based on multiclass prediction, which allows it to directly mimic Belady's MIN policy in a surprisingly simple and effective way. Our policy uses a PC-based predictor to learn each cache line's reuse distance; it then evicts lines based on their predicted time of reuse. We show that our use of multiclass prediction is more effective than binary prediction because it allows for a finer-grained ordering of cache lines during eviction and because it is more robust to prediction errors.**

Our empirical results show that our new policy, which we refer to as Mockingjay, outperforms the previous state-of-the-art on both single-core and multi-core platforms and both with and without a prefetcher. For example, with no prefetcher, on a mix of 100 multi-core workloads from the SPEC 2006, SPEC 2017, and GAP benchmark suites, Mockingjay sees an average improvement over LRU of 15.2%, compared to 7.6% for SHiP and 12.9% for Hawkeye. On a single-core platform, Mockingjay's improvement over LRU is 5.7%, which approaches the 6.0% improvement of Belady MIN's unrealizable policy. On a single-core platform (with a prefetcher) running the high-MPKI CVP workloads, Mockingjay's improvement over LRU is 20.1%, compared to 13.4% for Hawkeye.

I. Introduction

Cache replacement is an important and well-studied problem that has grown in sophistication over the years. Early solutions used variants of simple heuristics, such as LRU (Least Recently Used) and MRU (Most Recently Used). In 2007, Qureshi et al. [30] ushered in the era of adaptive solutions by introducing a policy that used efficient sampling to choose from among two different heuristics. The past decade has seen a movement to prediction-based policies that phrase the cache replacement problem as a binary prediction problem. For example, SDBP [20] predicts whether lines loaded by a given PC will be dead or alive, SHiP [41] predicts whether lines loaded by a PC will have a long or intermediate reuse distance, and Hawkeye [13] predicts whether a line loaded by a PC would tend to be cached or not cached if Belady's MIN policy had been used. These prediction-based solutions work well because they learn from historical behavior, which allows them to proactively evict lines that are unlikely to receive cache hits.

However, the coarse granularity of binary classification leads to two shortfalls: First, a small prediction error will flip a prediction from one class to the other. Second, there can be many ties among lines in the same class, which are typically broken by falling back on the same LRU heuristic that these polices attempt to improve upon.

An alternative to binary classification is multiclass prediction, where the different classes represent different reuse distances [3], [7], [10], [38]. One particularly elegant idea, first proposed by Keramidas et al. [19] and later improved upon by Petoumenos et al. [29], attempts to mimic Belady's MIN policy [6] more directly: The *stated goal* is to have a predictor learn each cache line's reuse distance and to always evict the line that is predicted to be reused furthest in the future. Conceptually, each reuse distance is translated into the line's predicted time of reuse, which is known as its ETA (Estimated Time of Arrival), and lines are ordered by ETA [19], [29].

Unfortunately, there are two reasons why these previous ETA-based solutions do not in fact closely mimic Belady's MIN. First, their eviction policies deviate from their stated goal by using a combination of a line's predicted ETA and the line's age in the cache. Section II explains this flaw conceptually; empirically, we find that 93% of the decisions in Keramidas et al.'s solution and 36.2% of the decisions in Petoumenos et al.'s solution use an age-based LRU ordering. Second, even when these solutions use ETA-based ordering, their predicted ETA values are often incorrect due to the low accuracy of their reuse distance predictions, as we explain in Section II. As a result, neither solution performs well, with Petoumenos et al.'s solution improving IPC over LRU by just 2.6% on a set of SPEC and GAP benchmarks. Thus, the research community has apparently moved away from the idea of ETA-based replacement, as none of the published entries in the 2^{nd} Cache Replacement Championship used ETA-based solutions.

In this paper, we present Mockingjay,[2] an ETA-based policy that faithfully adheres to the stated goal of ETA-based replacement. We show that Mockingjay outperforms

[*]This work was done before the author joined Google.

[1]Belady's MIN is not well defined for multicore systems.

[2]A mockingjay is a fictional bird from the *Hunger Games* trilogy that can remarkably memorize and mimic human melodies and songs.

	Improvement Over LRU	Year
SHiP	3.4%	2011
Hawkeye	4.5%	2016
Mockingjay	5.7%	2021
Belady's MIN	6.0%	1966

Table I: IPC improvement over LRU for our benchmarks (on a single core[1] with no prefetcher).

the previous state-of-the-art policies and approaches the performance of Belady's impractical MIN policy (see Table I). Moreover, Mockingjay represents an interesting milestone, as it is the first replacement policy that can obtain better performance than an LRU cache that is twice as large. In particular, in a single-core setting with no prefetcher, a 1MB cache using Mockingjay outperforms a 2MB cache using the LRU policy, similarly for 2MB/4MB and 4MB/8MB.

Our solution is effective because (1) it produces accurate reuse distance predictions by using a long history of past accesses and by using per-set reuse distances, and (2) it considers age information only in the few cases where reuse distance information is unavailable. As a result, Mockingjay is able to use ETA information—and thus mimic MIN—for 92% of its evictions.

Compared to the recent state-of-the-art solutions that use binary classification, there are three benefits of our solution. First, multiclass prediction is more resilient to prediction errors than binary classification because a single misprediction has a smaller impact on the predicted ordering of cache lines. Second, as we demonstrate empirically in Section V, errors in the prediction of ETAs translate to smaller errors in the prediction of the *relative order* of ETAs. Third, while most recent policies [13], [20], [34], [41] predict a line's eviction priority at the time of insertion, our solution computes ETAs at insertion time but defers its interpretation as an eviction priority—i.e., its comparison against other ETAs—until eviction time, when more information is available.

This paper makes three main contributions.

- We demonstrate that it is possible to effectively emulate Belady's MIN policy where knowledge of the future is replaced with reuse distance prediction.
- We provide insights (see Section V) that explain why Mockingjay's multiclass prediction approach is superior to recent policies that rely on binary classification.
- We demonstrate that Mockingjay comes extremely close to the performance of Belady's MIN policy (see Table I) and outperforms the previous state-of-the-art policies. For memory-intensive programs from the CVP workloads running on a single core with a prefetcher, Mockingjay improves performance by 20.1% (vs. 13.4% for Harmony[3]). For a mix of SPEC and GAP benchmarks running on a four-core system, the improvements are 13.3% for Mockingjay and 11.1% for Harmony. The Mockingjay source code can be found at https://github.com/ishanashah/Mockingjay.

[3]Harmony is an extension of Hawkeye that is superior in the presence of prefetching. In the absence of prefetching, they behave identically.

This paper also makes the following secondary contributions:

- We show that on a single core with prefetching, Mockingjay reduces uncore (LLC+DRAM) energy consumption by 9.1% compared to Harmony.
- We present an ablation study that shows the relative importance of different components of ETA-based policies [19], [29] (see Figure 6), and we find that both accurate ETA prediction and accurate ETA-based eviction are essential for mimicking MIN.
- We show the performance impact of using four different prefetchers: We find that Mockingjay performs best regardless of the prefetcher, and we show evidence that Mockingjay performs better as the prediction accuracy of the prefetcher improves.

The remainder of this paper is organized as follows. We describe Related Work in Section II, and we describe our solution in Section III. We then present our experimental evaluation in Section IV, followed in Section V by a discussion of the sources of Mockingjay's performance advantage. We conclude in Section VI.

II. Related Work

In 1966, Belady proposed the clairvoyant MIN cache replacement policy [6], which is optimal but impractical. Since then, numerous cache replacement policies have been devised that to varying degrees aim to emulate Belady's policy without looking into the future. We now discuss prior work by relating them to Belady's MIN, dividing previous work into two categories: (1) memoryless policies that do not use historical information to distinguish among lines when they are inserted in the cache, and (2) prediction-based policies that learn from historical behavior to predict future caching behavior of incoming lines.

A. Memoryless Policies

Early solutions modulate replacement priority by observing the reuse behavior of cache-resident lines. These solutions typically emulate Belady's MIN under strong assumptions. For example, if we assume that lines have temporal locality, then the line that is reused furthest in the future will be the oldest line in the cache, yielding the LRU (Least Recently Used) policy. However the LRU policy is susceptible to thrashing when the working set size exceeds the cache capacity. Therefore, other solutions [9], [17], [23], [27], [35], [40] preferentially evict newer lines, because for a thrashing access pattern, the newest line is likely to be reused furthest in the future [30], [32].

Jaleel et al. recognize that a spectrum of policies exist between LRU and MRU, and they accommodate scanning accesses[4] using their RRIP policy [15]. To adapt to changes in access patterns over time, adaptive cache replacement solutions [30], [36] dynamically change the replacement policy over time. All of these policies have a key limitation: They are customized to a few specific access patterns and do not mimic MIN for more complex cache access patterns.

Shepherd Cache is the memoryless policy that most closely emulates Belady's MIN policy [31]. Shepherd Cache repurposes some ways from the cache to extend the main cache's window into the future. Therefore, the fundamental tradeoff is that that for a fixed-sized cache, the longer the lookahead, the smaller the main cache. Previous studies [13] have shown that Belady's MIN requires a long window into the future that is about 8× the size of the cache, which suggests that for the Shepherd Cache to approach MIN's behavior, it would need to shrink the main cache to $\frac{1}{8}$ its size.

B. Prediction-Based Policies

Mockingjay is the latest in a recent trend that takes a predictive approach to the cache replacement problem, where past behavior is observed at a fine granularity (typically at the granularity of load instructions) to guide future caching decisions. These policies use past behavior to predict either binary caching priorities or multiclass reuse distances.

Binary-Classification-Based Policies: Dead block predictors [18], [20], [22] and many recent state-of-the-art replacement policies [13], [16], [39], [41] phrase cache replacement as a binary prediction problem, where the goal is to predict whether an incoming line should be cached or not cached. For example, SDBP [20] and SHiP [41] learn whether loads by some instructions are more likely to result in cache hits than others, and both policies preferentially evict lines inserted by load instructions that are predicted to be *cache-averse* (SDBP correlates reuse with the load instruction that first loaded the line, whereas SHiP correlates reuse with the load instruction that caused the reuse). The perceptron predictor [39] and MPPPB [16] use richer features and more sophisticated prediction mechanisms to improve prediction accuracy. Instead of learning from the hit and eviction behaviors of the LRU policy, the Hawkeye cache replacement policy [13], [14] learns the caching behavior of the optimal solution for past accesses. The optimal solution is produced by applying a variant of Belady's MIN algorithm on a long history of past cache accesses (8× the size of the cache), and the optimal solution is learned using a PC-based predictor. Glider [34] improves upon Hawkeye in two ways: (1) it uses better features that were identified using deep learning and (2) it uses a more sophisticated predictor to perform its binary classification. Section V explains the drawbacks of these binary classification-based policies.

[4]Scans refer to accesses that are never reused.

Reuse-Distance Prediction Policies: Many cache replacement policies [3], [7], [8], [10], [19], [21], [25], [38] perform reuse distance prediction but differ fundamentally from Mockingjay in three respects. First, instead of mimicking MIN, these solutions [3], [7], [10], [38] protect lines until their age exceeds their predicted reuse distance, even if those reuse distances are long. Thus, these policies *protect* lines that are reused furthest in the future, instead of *evicting* lines that are reused furthest in the future. Second, the effectiveness of such protection-based policies is exposed to even small errors in reuse distance predictions. By contrast, Mockingjay is exposed to errors in the relative ordering of ETAs, which we show in Section V occur less frequently than errors in reuse distance prediction. Leeway [8] introduces novel prediction techniques to adapt to variability in reuse distances, but Leeway continues to rely on line protection, which is wasteful when the line will be eventually evicted anyway. Third, none of these policies learn long reuse distances (8× the size of the cache), which limits their prediction accuracy.

The EVA policy [5] differs from other reuse distance prediction based policies as it computes a line's priority based on the statistical distribution of age values for which cache hits are observed. Mockingjay enjoys two benefits over EVA. First, Mockingjay estimates reuse distances at a fine granularity by estimating a different reuse distance for each PC, whereas EVA estimates reuse distances for just two classes of lines, those that have been reused at least once and those that have not. Second, Mockingjay uses a simpler hardware solution, whereas EVA uses a software routine to update eviction priorities.

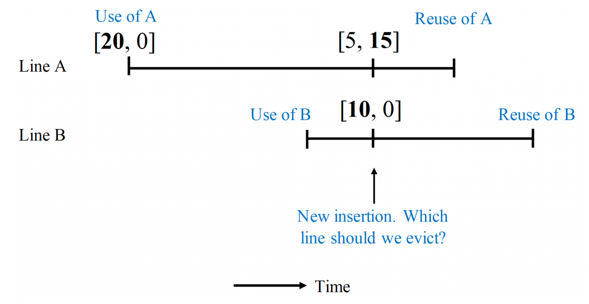


Figure 1: KPK and IbRDP use max(ETR, age) to make eviction decisions, so they erroneously evict *A*, while MIN would evict *B*, whose ETA is further in the future. Here, [20,0] represents an ETR of 20 and an age of 0, and the bold lettering highlights the max of the two values.

Two prior solutions, KPK [19] and IbRDP [29], share Mockingjay's goal: Evict lines based on their ETA. However, both solutions suffer from the fundamental flaw that instead of evicting lines based on their ETA, they take the larger of the line's ETR (Estimate Time Remaining) and its age in the cache, where ETR is a counter, initialized to a line's reuse distance, that counts down as it ages. **But this scheme**

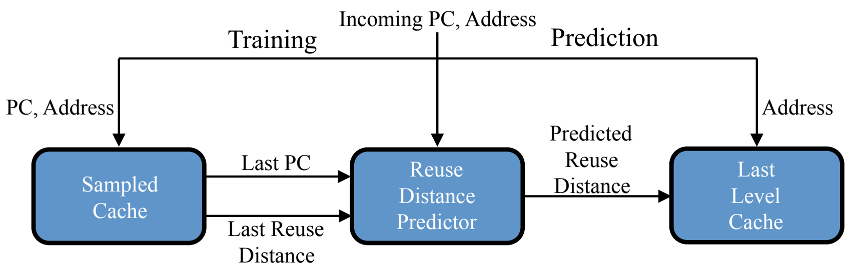


Figure 2: Overview of the Mockingjay cache replacement policy.

does not order lines based on ETA. Figure 1 shows an example where A will be reused earlier than line B, so at the point where a new line is inserted, line B should be evicted. However, both KPK and IbRDP would instead evict A because max(ETR,age) for A is max(15,5) = 15, which is greater than max(ETR,age) for B, which is max(10,0) = 10. **In general, once a line's age reaches half of its reuse distance, KPK and IbRDP ignore the line's ETA and instead use the line's age to make eviction decisions, which means that a line is most likely to be evicted just as it reaches its ETA.** By contrast, under Belady's MIN policy, lines that approach their ETA are the *last* to be evicted.

Even when KPK and IbRDP use ETA-based eviction, they are limited because of their low reuse distance prediction accuracy, as we now explain.

KPK observes low reuse distance prediction accuracy because it does not use a long history of the past and instead predicts reuse distances based only on the contents of the cache—prior results show that Belady's MIN performs worse than LRU if given a window into the future that is equal to the size of the cache [13].

IbRDP improves upon KPK by using a long history, but it measures reuse distances and ETRs in terms of global cache accesses, and we find empirically that global reuse distances have extremely high variability. As a result, IbRDP has a prediction accuracy of just 43%. Mockingjay is much more accurate than both KPK and IbRDP, achieving a prediction accuracy of 85%. Mockingjay's superior accuracy can be attributed to its use of long history and its use of per-set reuse distances.

Finally, Liu et al. present a machine learning model to directly imitate Belady's MIN policy for the past to predict future eviction decisions [24], but their solution is too expensive to deploy in hardware.

III. Our Solution

Our Mockingjay solution, which we apply to the last-level cache (LLC) is designed to evict the line that is predicted to be reused furthest in the future. Conceptually, for each cache insertion, the line's estimated time of arrival (ETA) is the sum of the current timestamp and the line's predicted reuse distance; at each insertion, the line with the largest ETA is evicted. To predict reuse distances, a PC-based predictor is used to estimate each cache line's reuse distance.

A. Key Components

We now describe the three components of our solution, which are the Sampled Cache, the Reuse Distance Predictor (RDP), and the ETA Counters that reside in the LLC (see Figure 2). We defer the discussion of their hardware implementation to Section III-B.

Sampled Cache: The goal of the Sampled Cache is to track past reuse distances to train the RDP, which predicts future reuse distances. To track reuse distances, the Sampled Cache maintains a long history of past cache accesses for a few sampled cache sets. In keeping with previous claims that Belady's MIN policy requires a long view of the future [13], Mockingjay maintains a history length that is 8× the size of each sampled set. Thus, the Sampled Cache enables Mockingjay to learn both short and long reuse distances.

The Sampled Cache is organized as a set-associative cache and is indexed using the cache line address. Each entry in the Sampled Cache maps block addresses to their last access timestamp and their last PC signature. Since the Sampled Cache only stores unique lines in the 8× history, the required space is much smaller than 8× the total capacity of the sampled sets.

Reuse Distance Predictor (RDP): The RDP is a PC-based predictor that learns reuse distances for loads initiated by a given program counter (PC). The RDP is organized as a direct-mapped cache and is indexed by a PC signature. Mockingjay uses separate predictor entries—and therefore learns separate reuse distances—for loads by a PC that hit in the cache and for loads by that same PC that miss in the cache. In Section III-B, we describe how we combine the PC and the hit/miss information into a hashed *PC signature* for indexing the RDP.

Each entry in the RDP is initialized to 0, and as lines are reused in the Sampled Cache, the RDP entry corresponding to the PC that last accessed the line is updated with the observed reuse distance. Section III-B explains how we use coarse-grained timestamps in the Sampled Cache to efficiently produce reuse distance observations.

Since reuse distances for a PC can oscillate, we use *temporal difference learning* [37] to train the RDP gracefully in the presence of outliers. Temporal difference learning updates predicted values as a linear function of the difference between the predicted and observed values. Thus, to limit the effect of outliers, the counter update is biased to maintain the old value but is still influenced by the new reuse distance.

More concretely, RDP entries are trained as follows: If the new reuse distance is larger than the previous RDP entry, then the entry value is incremented by w, where w is defined to be $min(1, \frac{diff}{16})$ and $diff$ is the absolute difference between the previous entry and the new reuse distance. If however the new reuse distance is smaller than the previous RDP entry, then the entry value is decremented by w. If the signature does not exist in the RDP, its entry is set to be the sampled reuse distance.

ETA Counters: Finally, the cache itself maintains the ETA for each line. Upon insertion into the cache, a line's predicted reuse distance is converted to an ETA, and these ETA values are used to make eviction decisions for future cache misses. Cache insertions that are predicted to have ETA values larger than any existing line in the set are bypassed, which means that they are not inserted in the cache. Promotions are treated the same as insertions, so Mockingjay produces an ETA prediction on both cache hits and misses.

To reduce the expense of maintaining precise ETA timestamps for each cache line, we use a smaller but logically equivalent value, known as the *Estimated Time Remaining (ETR).* The ETR value is initialized to the line's predicted reuse distance and is decremented each time some other line in the set is accessed. As the difference between the present time and the predicted ETA of a cache line decreases, the ETR of that line also decreases. Thus, the relative ordering of ETRs is exactly the same as the relative ordering of ETAs.

At times, our solution will underestimate a line's reuse distance, and its ETR counter value will reach 0 without seeing a reuse. If our policy were to let the counter value saturate at 0, this line would always retain the highest caching priority, which is undesirable. If on the other hand, such lines were given an infinite ETR value, then lines whose ETA were underestimated by just a small amount would immediately be evicted, which is also undesirable.

To handle these imprecise predictions, our solution continues to decrement ETR counters after they reach a value of 0. The negative ETR value indicates the time that has elapsed since the line's expected ETA. For example, an ETR counter value of -4 indicates that the line has exceeded its ETA by 4 set accesses. Upon eviction, our policy evicts the line with the largest absolute ETR value, which is the line furthest from its ETA. Thus, the evicted line is either predicted to be reused furthest in the future or it was predicted to be reused furthest in the past. Ties are broken by evicting lines with a negative ETR in favor of lines with a positive ETR.

B. *Implementation Details*

We now provide additional details for Mockingjay's three components.

Sampled Cache: The Sampled Cache maintains a long history of cache accesses for 32 sampled sets (the Sampled Cache maintains only tags, not data). For a 16-way cache, this history includes the past 128 cache accesses for each sampled set. We implement the Sampled Cache as a 5-way set-associative cache with 512 sets. Conceptually, we can view the 512 sets in the Sampled Cache as a collection of 32 smaller sub-caches, where each sub-cache has 16 sets and maintains the history of cache accesses for one of the 32 sampled sets. Thus, the Sampled Cache is indexed using a concatenation of the 5 set id bits that identify the 32 sampled sets and the bits [3:0] of the block address tag. Sampled Cache lines are tagged with bits [13:4] of the block address tag.

The Sampled Cache is managed using an LRU replacement policy. Each Sampled Cache entry maps block addresses to their last access timestamp and their last PC signature. In particular, each entry includes a valid bit, a 10-bit block address hash, an 11-bit PC signature, and an 8-bit timestamp indicating the time of last access.

For every access to a sampled LLC set, the Sampled Cache is searched. On a Sampled Cache hit, the last timestamp of the block is used to train the RDP with the observed reuse distance. On a Sampled Cache miss, the least recently used line is evicted from the Sample Cache, and the PC signature corresponding to the evicted line is trained to learn that it was not reused and was thus a scan. We associate scanning accesses with an infinite reuse distance and represent them by the maximum possible reuse distance value INF_RD, which in our case is 127. In either case, the Sampled Cache is updated with the current timestamp and the PC signature of the new access. The current timestamp is maintained as an 8-bit running counter for each sampled LLC set and is incremented on every set access. Timestamps wrap around on overflow, but since the current timestamp must occur later than the last access timestamp, we can detect overflow when the current timestamp has a smaller value than the last access timestamp. In this case, we add 1 << TIMESTAMP_BITS to the current timestamp before computing the difference. It is possible that the current timestamp's wrapped around value can exceed the last access timestamp, but since we evict cache lines that are observed to be more than 128 set accesses old, this case is rare.

Reuse Distance Predictor: The RDP is a direct-mapped array that is indexed by the PC signature, and it stores the predicted reuse distance for the blocks corresponding to this signature. The PC signature is a hash of the 11 least-significant bits of the program counter with a bit indicating whether the cache access was a hit or a miss. Each entry in the RDP is a 7-bit saturating counter representing the number of set accesses before a cache line is predicted to be reused.

ETR Counters: On insertion and promotion, a line's ETR is initialized with its predicted reuse distance obtained from the RDP. As explained earlier, scans are associated with an infinite reuse distance, INF_RD, which in our case is 127. To ensure that scanning lines retain the highest eviction priority, lines with a predicted reuse distance of INF_RD are never aged.

Since reuse distance predictions are not perfect, there is uncertainty about lines with a large reuse distance: Should they be considered to be a part of a scan, or should they be given the opportunity to be cached? We find that the former is preferred, because the latter ties up valuable cache space, so we define a threshold, MAX_RD, whose value is close to INF_RD, and any line whose ETR value is greater than this threshold is treated as a scan. In our evaluation, we use MAX_RD = 104, but we find that it can be set to any value that is slightly smaller than INF_RD.

For space efficiency, Mockingjay tracks coarse-grained reuse distances (and corresponding ETRs), which are obtained by dividing the precise value by a constant factor f, where f is set to 8 in our evaluation. To age a line's ETR value, its counter is decremented by an average of 1 on every set access. Thus, the ETR counters for all non-scanning lines in a set are aged every f set accesses. To support this aging scheme, we use a 3-bit clock for every set; the clock is initialized to zero and is incremented on every set access. Every eight set accesses, the set's clock is reset to 0, and every line in the set is aged. As mentioned earlier, a line that has exceeded its ETA—i.e., its ETR counter has reached 0—is still aged by decrementing its ETR. The absolute ETR value in this case indicates the extent to which the line has exceeded its predicted ETA. These aging operations are similar to those of the RRIP policy [15] and are off the critical path.

The insertion and promotion operations also lie off the critical path because they do not interfere with the cache controller's ability to determine whether the line hits or misses. The complexity of Mockingjay's insertion and promotion operations is similar to those of prior prediction-based solutions [13], [41].

Finally, on a cache miss, the line with the largest absolute ETR value is evicted. The use of the max function makes eviction in Mockingjay more expensive than in prior solutions, but these operations can be performed while the cache miss is being serviced from main memory, so they do not affect Mockingjay's hit or miss latencies.

C. Multicore Implementation

Mockingjay's multicore implementation does not significantly differ from its single-core implementation, but a few parameters are scaled to accommodate the increased pressure from multiple cores. First, we increase the signature size to 10 + log(number_of_cores), and the load address is hashed with the core identifier to produce the RDP signature. Second, the RDP is scaled with the number of cores. Finally, the number of sampled sets is scaled with each core, so a 4-core application uses 128 sampled sets.

Table II: Baseline configuration.

Out-of-order Core	352-entry ROB, 128-entry LQ, 72-entry SQ, FetchWidth=6, ExecWidth=4, RetireWidth=4
L1 I-Cache	32 KB, 8-way 4-cycle latency, 8 MSHRs
L1 D-Cache	32 KB, 8-way 4-cycle latency, 16 MSHRs
L2 Cache	256 KB, 8-way 8-cycle latency, 32 MSHRs
LLC per core	2 MB, 16-way 20-cycle latency, 64 MSHRs
DRAM	tRP,tRCD,tCAS=12.5ns 800 MHz, 25.6 GB/s

D. Prefetching Implementation

Since MIN is not optimal in the presence of a prefetcher [14], Mockingjay emulates Flex-MIN [14] in the presence of a prefetcher. The key idea behind Flex-MIN is to preferentially evict lines that will be prefetched in the future.

To identify such lines, Jain and Lin describe the concept of a cache *usage interval,* which refers to the time interval between consecutive accesses to the same line. Since the endpoints of usage intervals can be caused by either a demand access or a prefetch, there are four kinds of usage intervals, namely, D-D, D-P, P-D, and P-P intervals, where a D-D interval represents a demand reuse, while a P-D interval represents a useful prefetch, and a *-P interval represents a line that will be prefetched.

Except for possibly improved timeliness, *-P intervals do not need to be cached because the subsequent prefetch will bring the data back into the cache. However, the eviction of *-P intervals can result in extra prefetcher traffic, since every prefetch request will miss in the cache and will be sent to memory. Thus, Flex-MIN gives *-P intervals low priority only when they are long enough to allow other cache lines to use the freed up cache space.

Mockingjay emulates Flex-MIN by increasing the reuse distance prediction of all *-P lines by a constant factor, thereby discouraging the Mockingjay policy from caching *-P lines. In single core applications, Mockingjay penalizes *-P lines by a factor of 2. In particular, for *-P intervals, the RDP receives a sample reuse distance that is equal to the observed reuse distance times 2. In case the inflated reuse distance is larger than INF_RD, it is saturated at INF_RD. In multicore applications, traffic to main memory is generally higher, so our solution inflates reuse distances by a factor of 1.5. In Section IV, we show Mockingjay's sensitivity to this penalty parameter.

Finally, to distinguish demand requests and prefetch requests, the RDP signature is hashed with the prefetch bit, where the prefetch bit corresponds to the program counter of the load instruction that triggered the prefetch.

E. Hardware Budget

Our solution requires a hardware budget of 31.91 kilobytes on one core and 127.62 kilobytes on 4 cores. The introduction of a prefetcher does not affect the hardware budget.

Sampled Cache History: The total hardware budget for the Sampled Cache is 9.41KB in a single-core setting and 37.63KB in a 4-core setting. In the 4-core version, we scale the Sampled Cache size by a factor of 4 as we sample 4× more sets. We use 8-bit block address hashes and 13-bit signature hashes per entry.

Reuse Distance Predictor: The reuse distance predictor (RDP) maps an 11-bit signature to a 7-bit reuse distance prediction. The total hardware cost for the RDP is 1.75KB. The total 4-core hardware cost for the RDP is 7KB, since signatures are instead 13 bits.

ETR Counters: We use a 5-bit ETR counter, such that counter values can range from -15 to 15 for each line (INF_ETR is set to 15). Each cache set also requires a 3-bit clock to age the cache lines correctly. Thus, the total hardware cost to maintain the ETRs is 20.75KB on a 2MB single-core cache and 83KB on a 8MB 4-core cache.

IV. Evaluation

This section presents our empirical evaluation of Mockingjay.

A. Methodology

Simulator: We evaluate our solution using the most recent version of the ChampSim simulator [2], which was originally released by the 2nd JILP Cache Replacement Championship. Table II shows the parameters for our simulated core and memory hierarchy.

Benchmarks: To evaluate Mockingjay, we use the 33 memory-sensitive applications of the *SPEC CPU2006*, *SPEC CPU2017*, and *GAP* [4] benchmark suites, where we define memory-sensitive applications to be those that have an LLC miss per kilo instructions (MPKI) greater than 1. For the *SPEC CPU2006* and *SPEC CPU2017* benchmark suites, we run the benchmarks using the reference input set, and for *GAP* benchmarks, we use graphs of size 2^{17} nodes. For each benchmark, we use the SimPoint tool [33] to generate a single sample of 1 billion instructions. We warm the cache for 200 million instructions and measure the behavior of the next 1 billion instructions.

We also evaluate Mockingjay on benchmarks provided by Qualcomm for the CVP1 Championship [1]. Since this suite includes over 1300 benchmarks, we choose the 25 benchmarks that have the highest LLC MPKI with the LRU policy.

Multi-Core Workloads: Our multi-core simulation methodology is similar to that of the 2nd Cache Replacement Championship [2]. We simulate four benchmarks running on 4 cores, choosing a random set of 100 mixes from all possible workload mixes that could be created by combining our 33 *SPEC CPU2006*, *SPEC CPU2017*, and *GAP* benchmarks. For each mix, we simulate the simultaneous execution of SimPoint samples of the constituent benchmarks until each benchmark has executed at least 1 billion instructions. If a benchmark finishes early, it is rewound until every other application in the mix has finished running 1 billion instructions; each application is warmed up for 200M instructions. Thus, all the benchmarks in the mix run simultaneously throughout the sampled execution.

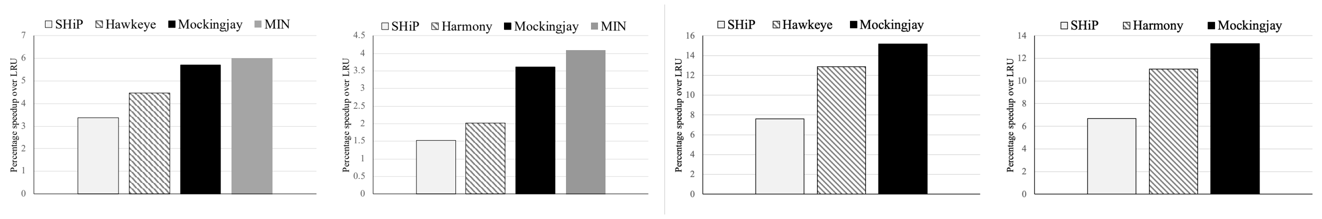
To evaluate performance, we report the weighted speedup normalized to LRU for each benchmark mix. This metric is commonly used to evaluate shared caches [2], [13], [16] because it measures the overall performance of the mix and avoids domination by benchmarks that have high IPC. The metric is computed 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 mix as the sum of $IPC_{shared}/IPC_{single}$ for all benchmarks in the mix, and we normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

Baseline Replacement Policies: We compare Mockingjay with two state-of-the-art replacement policies, namely, SHiP [41], and Hawkeye [13]. For configurations that include a prefetcher, we replace Hawkeye with Harmony [14] since Harmony is an extension of Hawkeye that performs better in the presence of prefetching, and in the absence of prefetching, Harmony defaults to Hawkeye. For a fair comparison, all policies are given a 32 KB hardware budget. The baseline policies all use binary classification, while Mockingjay, of course, predicts reuse times.

We also compare against KPK [19] and IbRDP [29],[5] which are closest to Mockingjay in terms of ETA-based eviction, as both KPK and IbRDP predict reuse distances and evict lines based on a combination of their ETAs and age.

Metrics: To evaluate performance in a single-core setting, we report IPC speedup over LRU; for multi-core settings, we report the weighted speedup over LRU. The weighted speedup metric is commonly used to evaluate shared caches [2], [13], [16] because it measures the overall performance of the mix and avoids domination by benchmarks of 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

[5]We present results for the best version of IbRDP, which is referred to as IbRDP+SC in the original paper.



(a) Single-core (no prefetching). (b) Single-core with prefetching. (c) Multi-core (no prefetching). (d) Multi-core with prefetching.

Figure 3: Summary of results on all evaluated configurations.

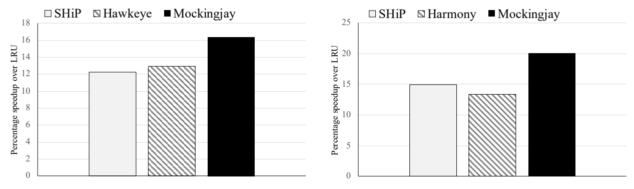
weighted IPC of the mix as the sum of $IPC_{shared}/IPC_{single}$ for all benchmarks in the mix, and we normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

We also report Mockingjay's impact on uncore energy consumption. To estimate energy consumption, we assume 1 unit of energy for each LLC access and an average of 25 units of energy for each DRAM access [11], [42].

B. Summary of Results

Figure 3 shows that Mockingjay outperforms the baselines on all evaluated configurations. On a single-core system without prefetching, Mockingjay improves performance over LRU by 5.7%, while SHiP and Hawkeye improve performance by 3.4% and 4.4%, respectively. Mockingjay's improvement over the baselines improves in the presence of a prefetcher, as Mockingjay improves performance by 3.6%, whereas Harmony improves performance by 2.0%.

Mockingjay also works well in multi-core configurations. In the absence of prefetching, Mockingjay improves performance on a 4-core system by 15.2%, while SHiP and Hawkeye improve performance by 7.6% and 12.9%, respectively. In the presence of prefetching, Mockingjay improves performance on a 4-core system by 13.3%, while SHiP and Harmony improve performance by 6.7% and 11.1%, respectively.



(a) Single-core (no prefetching) (b) Single-core with prefetching

Figure 4: Mockingjay sees larger benefits for the CVP workloads.

Figure 4 shows that for the CVP workloads, which have much higher MPKI than SPEC, Mockingjay's benefits over the baselines improve. With a prefetcher, Mockingjay improves performance by 20.1%, compared to 13.4% for Harmony. When there is no prefetcher, Mockingjay improves performance by 16.4%, compared to 12.9% for Hawkeye.

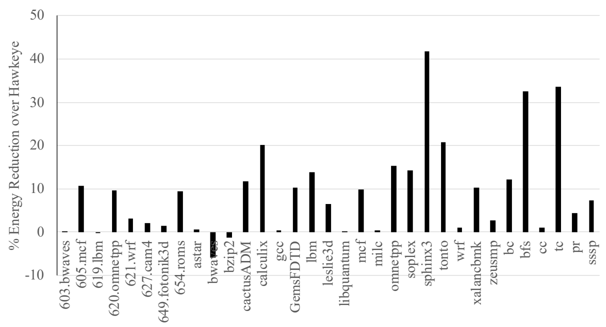


Figure 5: Mockingjay reduces uncore energy consumption.

Figure 5 shows that Mockingjay reduces uncore energy consumption by 9.1% compared to Hawkeye. Mockingjay's energy reduction can be attributed to its 9.8% lower DRAM traffic. Since uncore energy is a large proportion of total system energy, we expect Mockingjay to have a visible impact on overall system energy consumption.

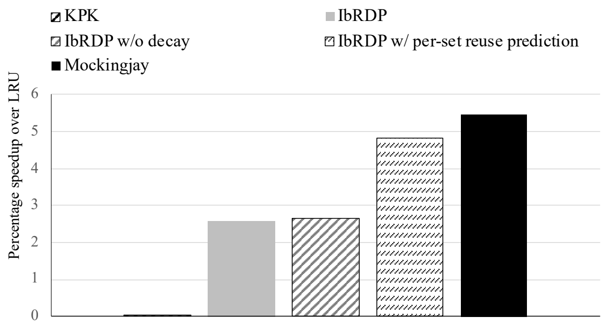


Figure 6: Understanding the difference between KPK, IbRDP, and Mockingjay.

Figure 6 shows that Mockingjay significantly outperforms both KPK and IbRDP. To better understand Mockingjay's improvement over IbRDP, we also show results for two modified versions of IbRDP. The first version uses Mockingjay's ETA-based eviction policy instead of combining ETA and age-based decay, and we see that despite mimicking MIN more faithfully, this version provides a marginal performance improvement because its ETA predictions are only 43% accurate. The second version of IbRDP uses per-set reuse distances, and we find empirically that this change improves reuse distance prediction accuracy from 43% to 85%. However, since this version continues to use IbRDP's original eviction policy that combines ETA with age-based decay, it underperforms Mockingjay (4.8% for IbRDP with per-set reuse distance prediction vs. 5.7% for Mockingjay). These ablation studies show that Mockingjay's benefit comes both from using an accurate per-set reuse distance predictor and from using an ETA-based eviction scheme that mimics MIN more faithfully.

Figure 7 confirms that long histories are essential for Mockingjay. In particular, we see that in the absence of prefetching, Mockingjay's performance suffers when it tracks reuse distances that are smaller than 4× the size of the cache. Increasing the history length to 8× the size of the cache provides a marginal benefit in the presence of prefetching and incurs no additional storage cost, so Mockingjay uses a history length of 8× the size of the cache.

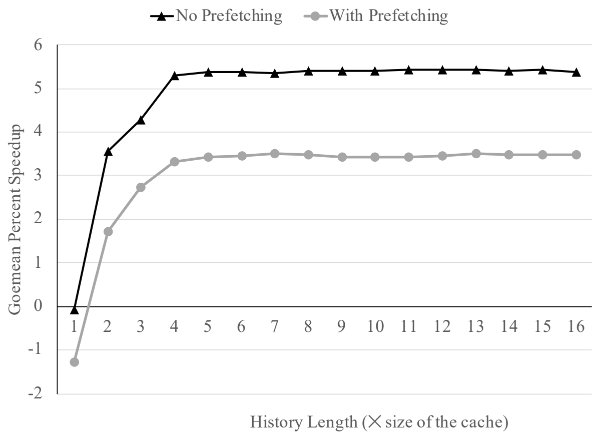


Figure 7: Mockingjay benefits from a long history.

C. Results: Without Prefetching

Figure 8 presents detailed results on the single-core configuration without prefetching. We see that Mockingjay outperforms SHiP and Hawkeye on all benchmarks except libquantum and milc. Mockingjay is not the best solution for such streaming workloads because streaming workloads benefit from a less aggressive bypassing policy. If Mockingjay were to use a less aggressive bypassing policy, its performance on libquantum would improve from 2.5% to 8.3%, but its average performance improvement would decrease from 5.7% to 5.2%.

While Figure 8 shows results for just the memory-intensive benchmarks, Mockingjay works well for the entire SPEC 2006, SPEC 2017, and GAP benchmark suites. In particular, for the entire suites, Mockingjay improves performance by 3.9% in the absence of prefetching (vs. 3.2% for Hawkeye).

A key difference between Hawkeye and Mockingjay is the way that they handle mispredictions. Hawkeye's mispredictions result in the eviction of the least-recently used cache-friendly line, whereas Mockingjay's mispredictions result in the eviction of a line whose ETA has elapsed. We refer to both of these cases as an LRU eviction for the respective policies. Figure 9 shows that Mockingjay defaults to LRU for only 7.8% of the total evictions, whereas Hawkeye defaults to LRU for 13.8% of its evictions.

Figure 11 shows that Mockingjay's reuse distance predictions—the initial ETR values—are quite diverse and that Mockingjay can learn both short and long reuse distance. The x-axis shows the initial ETR values that are quantized from 0 to 15, and the y-axis shows the percentage of insertions for the given ETR value. We see that across all of our evaluated benchmarks, 20% of reuse distances are infinite, 40% are short (within a 2× history), and 40% are long (2×-8× history). These results highlight Mockingjay's ability to cache lines with diverse reuse distances.

D. Results: With Prefetching

Figures 10 and 12 show results in the presence of a prefetcher for the single-core and multi-core configurations, respectively.

As we discussed in Section III, Mockingjay penalizes *-P intervals to improve demand hit rate. Figure 13 shows the sensitivity to this penalty; we see that Mockingjay's performance improvement levels out once the *-P penalty is greater than or equal to two.

Figure 14 shows that Mockingjay works well for a variety of prefetchers, including regular prefetchers, such as, IP-Stride, Best Offset Prefetcher [26], and IPCP [28], and for irregular prefetchers, such as, ISB [12]. In particular, we observe that Mockingjay's gap over the baselines improves for accurate prefetchers, such as IPCP and ISB.

E. Results: Neural Cache Replacement

Our results so far have compared replacement policies that use the same 32KB hardware budget, but a recently proposed solution called Glider [34] uses neural learning to improve upon Hawkeye's predictor accuracy. The resulting solution, which essentially uses perceptrons, uses a 64KB budget. Figures 15 and 16 compare two versions of Mockingjay, one with a 32KB hardware budget, another with a 43KB budget, against Glider. We see that even at half of Glider's hardware budget, Mockingjay still outperforms Glider on all four configurations. We also see that Mockingjay sees marginal improvements beyond a 32KB hardware budget.

The larger picture is that Mockingjay and Glider represent two different ways to improve upon Hawkeye. Mockingjay asks the question, "What should we predict?" By predicting

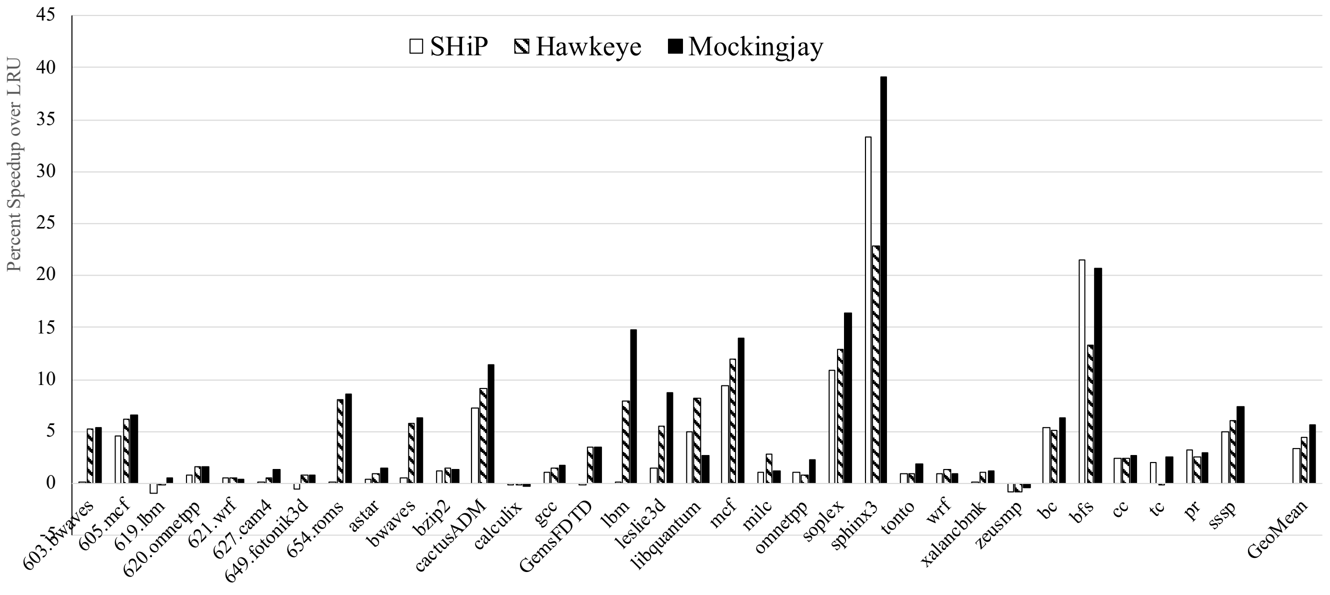


Figure 8: Comparison on single-core system (no prefetching).

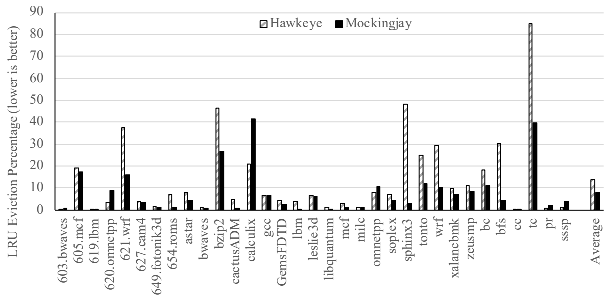


Figure 9: Mockingjay results in fewer LRU evictions than Hawkeye.

ETAs, Mockingjay gets an accuracy benefit and can make decisions at eviction time, when more information is available (see Section V). Glider instead retains Hawkeye's prediction problem and asks the question, "How can we improve predictor accuracy?" These graphs show that the change in prediction problem yields a greater benefit at lower cost—in terms of hardware budget—than the attempt to improve predictor accuracy through the use of neural learning.

V. Discussion

Hawkeye and Mockingjay would both behave identically to Belady's MIN if their predictions were completely accurate. In particular, with a perfect predictor, Hawkeye would classify as Cache Friendly any line that would be retained until its next access by the MIN policy, and it would classify as Cache Averse any line that MIN would evict before its next reuse. Thus, with a perfect predictor, Hawkeye would make the same eviction decisions as MIN. Of course, if Mockingjay had a perfect predictor, then its ETA ordering would be equivalent to MIN's ordering.

Of course, we don't have perfect predictors, so this section provides insights that explain why Mockingjay performs better than policies that use binary classification, such as Hawkeye and its variants [13], [14], [34], which predict whether lines will be Cache Friendly or Cache Averse. We will use Hawkeye as a point of comparison, but our discussion applies broadly to any policy that uses binary classification. We first describe our three insights, followed by empirical evidence that supports the first two insights.

A. *Resilience to Prediction Inaccuracy*

First, Mockingjay's reuse distance predictions are used to order lines in terms of predicted reuse, so errors in reuse distance prediction only have an impact if they are large enough to change the ordering of the lines. For example, if lines *A* and *B* have predicted reuse times (ETAs) that differ by 100, then any reuse prediction error for line *A* that is smaller than 100 will not affect the order of the two lines.

By contrast, any inaccuracy in a binary prediction will incorrectly place the mispredicted line either at the top of the priority queue or at the bottom of the priority queue.

B. *Local Impact of Classification Errors*

Second, when a prediction error does change the relative order of Mockingjay's predicted ETAs, these errors are more localized, so they are less costly than Hawkeye's errors.

To understand this point, consider a false positive prediction for Hawkeye, in which a Cache Averse line is incorrectly predicted to be Cache Friendly. As shown in Figure 17, this

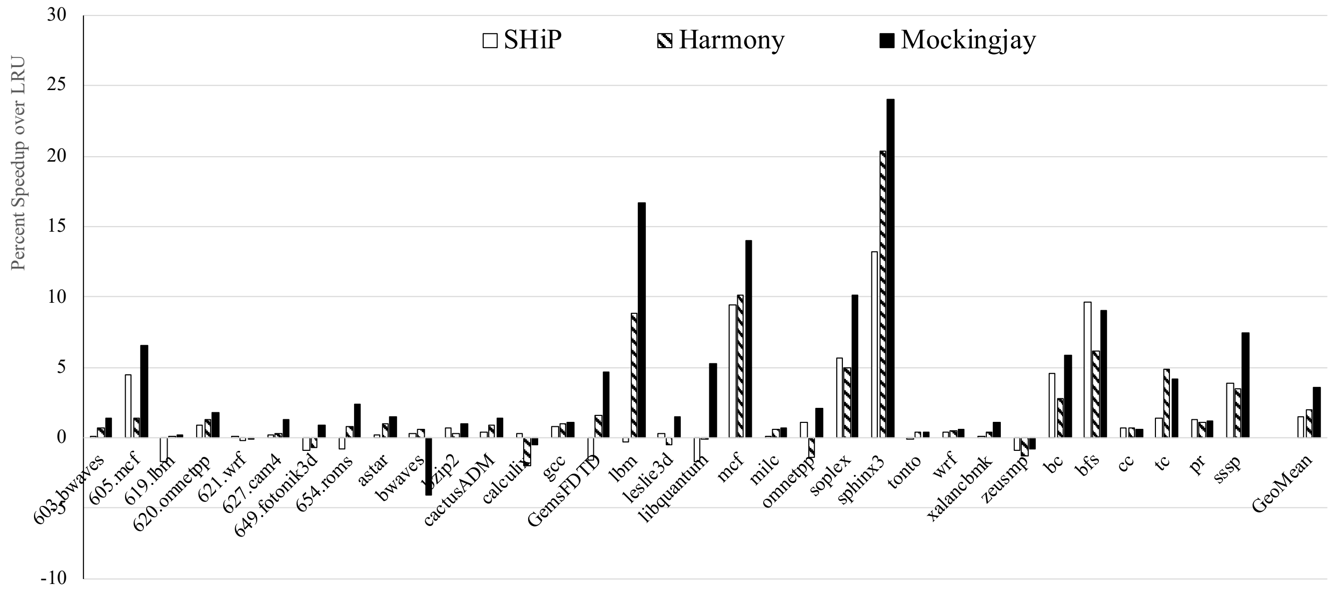


Figure 10: Comparison on single-core systems with prefetching.

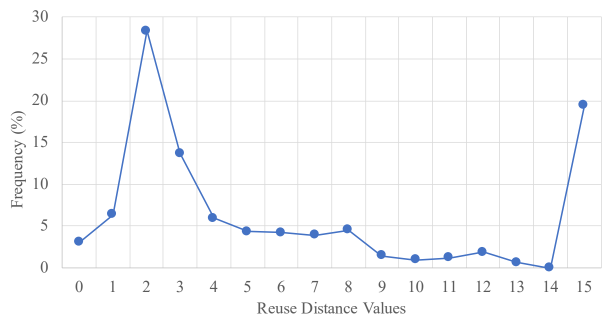


Figure 11: Predicted ETR values are distributed among short, long, and infinite reuse.

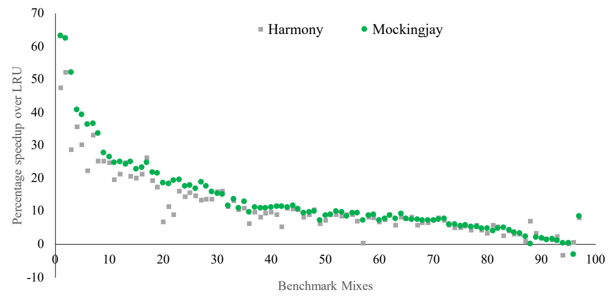


Figure 12: Multi-core comparison with prefetching.

mispredicition causes the line to be inserted with the highest priority, since it is prioritized over all Cache Averse lines and since ties among Cache Friendly lines are broken in LRU order. Thus, all existing cache lines in the set—including

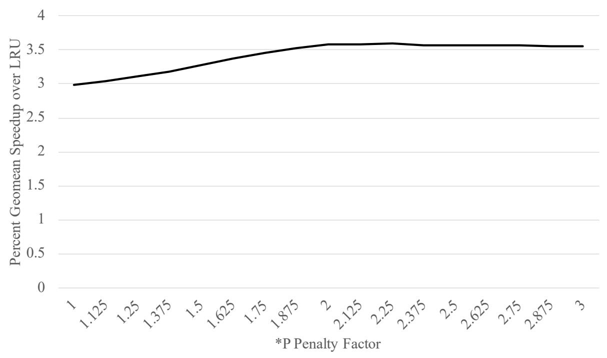


Figure 13: Mockingjay benefits from not caching lines that will be prefetched. In particular, penalizing *-P intervals by 2× is the most performant configuration.

other correctly predicted Cache Friendly lines—need to be evicted before this mispredicted line can be evicted. Moreover, the mispredicted line will occupy the cache for a long period of time only to be evicted without a cache hit. As a result, false positives are expensive for Hawkeye because their impact is not isolated; in the worst case, they can result in lost cache hits even for lines that are correctly predicted.

Mockingjay does not suffer from this same pathology for false positive predictions. Instead, prediction errors have only local impact. For example, as shown in Figure 18, if line A's reuse distance is incorrectly predicted to be shorter by 50, the relative order of A and B will change, but this error will not affect the caching priorities of C or D; C will continue to have the highest priority while D will continue to have

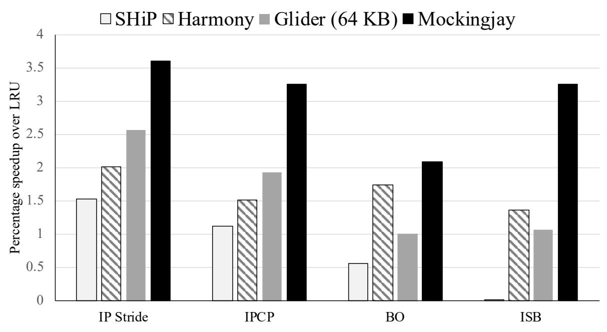


Figure 14: Mockingjay works well with different prefetchers, and its benefit increases for more accurate prefetchers.

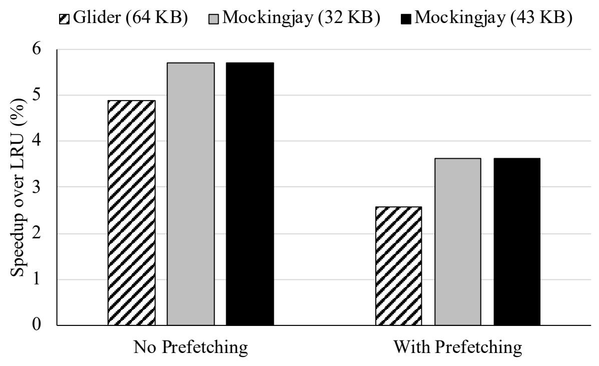


Figure 15: On a single-core single-core system, Mockingjay with a lower hardware budget outperforms Glider.

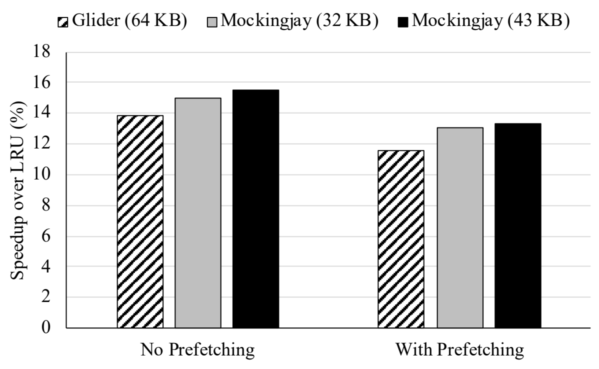
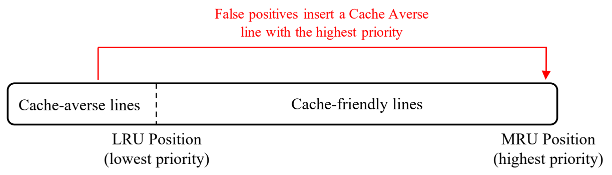


Figure 16: On a multi-core system, Mockingjay with a lower hardware budget outperforms Glider.



False positives insert a Cache Averse line with the highest priority
Cache-averse lines
Cache-friendly lines
LRU Position (lowest priority)
MRU Position (highest priority)

Figure 17: A costly binary classification error in Hawkeye.

the lowest priority in the cache. Mockingjay's reuse distance prediction error would have to be significant ly larger for it to give line *A* the highest priority.

Prediction errors in the other direction—false negatives for Hawkeye (in which a Cache Friendly line is predicted to be Cache Averse) and pessimistic predictions for Mockingjay (in which reuse distances are over-estimated), are less expensive because their impact is isolated to the mispredicted line. For example, for Hawkeye, the line with the false negative prediction will itself not receive a cache hit, but this error does not consume cache resources and does not prevent other correctly predicted Cache Friendly lines to receive cache hits.

C. Late Interpretation of Priorities

The final benefit of Mockingjay is that it interprets priorities at the time of eviction, when reuse information about other lines is available, whereas previous solutions based on binary prediction assign priorities at the time of insertion. In particular, the ETA that Mockingjay predicts at the time of insertion does not represent the line's priority; the line's priority is determined at the time of eviction when its ETA is compared to the ETAs of other candidates.

This difference is significant because it allows subsequent lines to impact a line's priority. For example, a line *A* that is inserted with a predicted reuse distance of 100 could have the highest priority if subsequent insertions had predicted reuse distances that were greater than 100, since *A* would then have the earliest ETA (Figure 19). At some other point in time, line *A* with the same predicted reuse distance of 100 could have the lowest priority if subsequent lines were inserted with sufficiently short predicted reuse distances that *A* would have the latest ETA (Figure 20). Thus, Mockingjay is more resilient to variability in how *other* lines use the cache.

D. Empirical Confirmation

Figure 21 quantitatively confirms these first two benefits for SPEC benchmarks: We see that large errors in ETA (x-axis), correspond to much smaller ordering errors (y-axis); we define the ordering error to be the error between the victim's ideal predicted position—the victim's position as chosen by MIN—and its actual position in the order. Each dot in the figure represents one benchmark; the value on x-axis represents Mockingjay's error in predicting reuse distances; and the value on y-axis represents Mockingjay's error in predicting the ordering of eviction candidates. Across all benchmarks, Mockingjay's ordering error is just 3.2, whereas LRU and Hawkeye see ordering errors of 8.2 and 4.1, respectively.

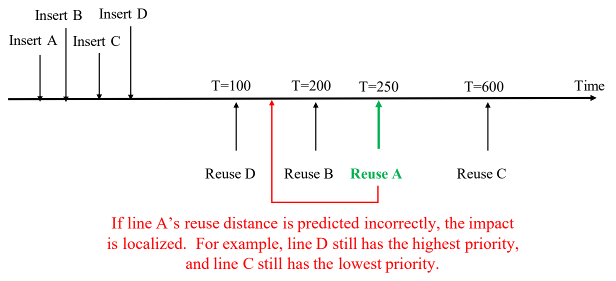

Insert A
Insert B
Insert C
Insert D
T=100
T=200
T=250
T=600
Time
Reuse D
Reuse B
Reuse A
Reuse C
If line A's reuse distance is predicted incorrectly, the impact is localized. For example, line D still has the highest priority, and line C still has the lowest priority.

Figure 18: Mockingjay is resilient to error in reuse distance predictions.

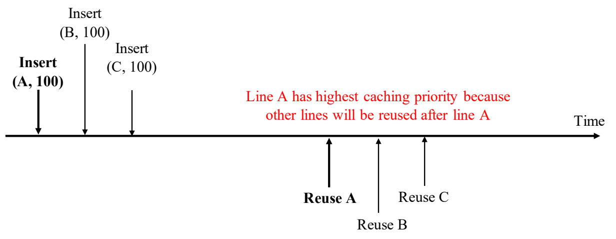

Insert (A, 100)
Insert (B, 100)
Insert (C, 100)
Line A has highest caching priority because other lines will be reused after line A
Time
Reuse A
Reuse B
Reuse C

Figure 19: Line A has the highest caching priority because subsequent insertions have larger reuse distances.

VI. Conclusions

In this paper, we have introduced the Mockingjay cache replacement policy, which mimics Belady's 1966 MIN policy but replaces MIN's knowledge of future reuse times with predicted reuse times. There are several keys to Mockingjay's success:

- It uses a *long history* of the past, which enables Mockingjay to perform accurate reuse distance prediction.
- It performs *multiclass prediction,* which is more resilient to prediction errors than binary prediction.
- It bases priorities on the *relative order* of predicted reuse times, rather than on predicted reuse times, which inoculates Mockingjay against prediction errors that are too small to change the relative order of reuse times.
- It infers priorities at the *time of eviction*—when additional information is available—rather than at the time of insertion.

We have evaluated Mockingjay's performance in systems that use various state-of-the-art data prefetchers, and we find that Mockingjay retains its superiority for all cases. We also observe that Mockingjay's performance gap over the baseline solutions appears to increase with the accuracy of the prefetcher.

Finally, we have shown that Mockingjay compares favorably to the recent Glider cache replacement policy [34], which uses a perceptron to combine an unordered history of past PCs. At a bit more than half the hardware budget, the 32KB Mockingjay outperforms the 64KB Glider.

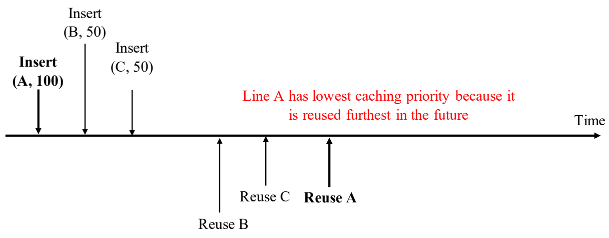

Insert (A, 100)
Insert (B, 50)
Insert (C, 50)
Line A has lowest caching priority because it is reused furthest in the future
Time
Reuse B
Reuse C
Reuse A

Figure 20: Line A has the lowest caching priority because subsequent insertions have shorter reuse distances.

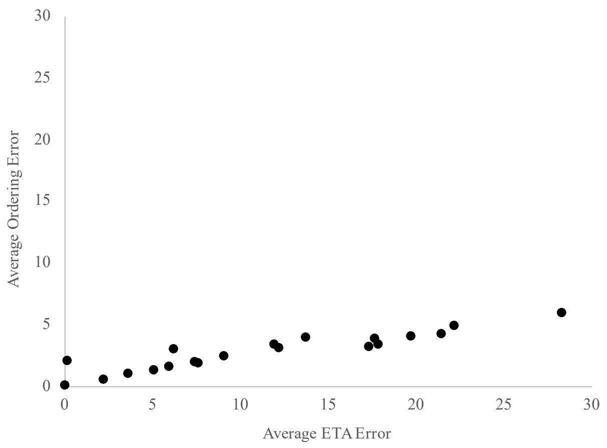


Figure 21: Mockingjay's error in predicting relative ordering of eviction candidates is low despite high errors in precise reuse distance prediction.

Acknowledgments

We thank the anonymous referees, Matthew Pabst, Chirag Sakhuja, and Molly O'Neil for their helpful comments on previous versions of this paper. This work was funded in part by a gift from Arm Research, NSF Grant CCF-1823546, and a gift from Intel Corporation through the NSF/Intel Partnership on Foundational Microarchitecture Research.

References

[1] "Championship value prediction." [Online]. Available: https://www.microarch.org/cvp1/

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

[3] J. Abella, A. González, X. Vera, and M. F. O'Boyle, "Iatac: a smart predictor to turn-off l2 cache lines," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 1, pp. 55–77, 2005.

[4] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[5] N. Beckmann and D. Sanchez, "Maximizing cache performance under uncertainty," in *23rd Annual IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 109–120.

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

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

[8] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in *26th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 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] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behavior," in *29th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2002, pp. 209–220.

[11] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann, 2010.

[12] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 247–259.

[13] ——, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in *43rd Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016, pp. 78–89.

[14] ——, "Rethinking Belady's algorithm to accommodate prefetching," in *45th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018, pp. 110–123.

[15] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *45th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2010, pp. 60–71.

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

[17] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, no. 3, pp. 38–46, 1994.

[18] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *18th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2001, pp. 240–251.

[19] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *25^{th} International Conference on Computer Design (ICCD)*, 2007, pp. 245–250.

[20] 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.

[21] M. Kharbutli and Y. Solihin, "Counter-based cache replacement algorithms," in *23rd International Conference on Computer Design (ICCD)*, 2005, pp. 61–68.

[22] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *28th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2001, pp. 144–154.

[23] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) policies," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1, 1999, pp. 134–143.

[24] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *International Conference on Machine Learning*, 2020, pp. 6237–6247.

[25] 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)*, 2008, pp. 222–233.

[26] P. Michaud, "Best-offset hardware prefetching," in *22nd Annual IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.

[27] 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.

[28] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier based hardware prefetching," in *47th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.

[29] P. Petoumenos, G. Keramidas, and S. Kaxiras, "Instruction-based reuse distance prediction replacement policy," in *JWAC 2010-1st JILP Workshop on Computer Architecture Competitions: cache replacement Championship*, 2010.

[30] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *34th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2007, pp. 381–391.

[31] 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.

[32] 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 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 355–366.

[33] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. 5, pp. 45–57, 2002.

[34] Z. Shi, X. Huang, A. Jain, and C. Lin, “Applying deep learning to the cache replacement problem,” in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 413–425.

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

[36] 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.

[37] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[38] M. Takagi and K. Hiraki, “Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches,” in *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, 2004, pp. 20–30.

[39] E. Teran, Z. Wang, and D. A. Jiménez, “Perceptron learning for reuse prediction,” in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[40] W. A. Wong and J.-L. Baer, “Modified LRU policies for improving second-level cache behavior,” in *6th International Symposium on High-Performance Computer Architecture (HPCA)*, 2000, pp. 49–60.

[41] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 430–441.

[42] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, “Temporal prefetching without the off-chip metadata,” in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 996–1008.