High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)

Aamer Jaleel† Kevin B. Theobald‡ Simon C. Steely Jr.† Joel Emer†

†Intel Corporation, VSSAD
Hudson, MA

‡Intel Corporation, MGD
Hillsboro, OR

{aamer.jaleel, kevin.b.theobald, simon.c.steely.jr, joel.emer}@intel.com

ABSTRACT

Practical cache replacement policies attempt to emulate optimal replacement by predicting the re-reference interval of a cache block. The commonly used LRU replacement policy always predicts a near-immediate re-reference interval on cache hits and misses. Applications that exhibit a distant re-reference interval perform badly under LRU. Such applications usually have a working-set larger than the cache or have frequent bursts of references to non-temporal data (called scans). To improve the performance of such workloads, this paper proposes cache replacement using Re-reference Interval Prediction (RRIP). We propose Static RRIP (SRRIP) that is scan-resistant and Dynamic RRIP (DRRIP) that is both scan-resistant and thrash-resistant. Both RRIP policies require only 2-bits per cache block and easily integrate into existing LRU approximations found in modern processors. Our evaluations using PC games, multimedia, server and SPEC CPU2006 workloads on a single-core processor with a 2MB last-level cache (LLC) show that both SRRIP and DRRIP outperform LRU replacement on the throughput metric by an average of 4% and 10% respectively. Our evaluations with over 1000 multi-programmed workloads on a 4-core CMP with an 8MB shared LLC show that SRRIP and DRRIP outperform LRU replacement on the throughput metric by an average of 7% and 9% respectively. We also show that RRIP outperforms LFU, the state-of the art scan-resistant replacement algorithm to-date. For the cache configurations under study, RRIP requires 2X less hardware than LRU and 2.5X less hardware than LFU.

Categories and Subject Descriptors

B.3.2 [Design Styles]: Cache memories, C.1.4 [Parallel architectures]

General Terms

Design, Performance.

Keywords

Replacement, Scan Resistance, Thrashing, Shared Cache

ISCA'10, June 19–23, 2010, Saint-Malo, France.

1. INTRODUCTION

An optimal replacement policy makes its replacement decisions using perfect knowledge of the re-reference (or reuse) pattern of each cache block and replaces the block that will be re-referenced furthest in the future. Practical cache replacement policies, on the other hand, can be viewed as basing their replacement decisions on a *prediction* of which block will be re-referenced furthest in the future and pick that block for replacement. Typically, on a miss, replacement policies make a prediction on when the missing block will be re-referenced next. These predictions can be updated when further information about the block is available, for example, on a re-reference.

In the commonly used Least Recently Used (LRU) replacement policy, the LRU chain represents the recency of cache blocks referenced with the MRU position representing a cache block that was most recently used while the LRU position representing a cache block that was least recently used. Recent proposals on cache insertion policies [11, 25, 28, 30] and hit promotion policies [30] have altered the description of the LRU chain. Rather than representing recency, the *LRU chain* can instead be thought of as a *Re-Reference Interval Prediction (RRIP) chain* that represents the order in which blocks are predicted to be re-referenced. The block at the head of the RRIP chain is predicted to have a *near-immediate* re-reference interval while the block at the tail of the RRIP chain is predicted to have a *distant* re-reference interval. A *near-immediate* re-reference interval implies that a cache block will be re-referenced sometime soon while a *distant* re-reference interval implies that a cache block will be re-referenced in the distant future. On a cache miss, the block at the tail of the RRIP chain (i.e., the block predicted to be referenced most far into the future) will be replaced[1].

Using the RRIP framework, LRU replacement predicts that a block filled into the cache has a *near-immediate re-reference* interval and thus places it at the head of the RRIP chain. Upon re-reference to a block, LRU updates its prediction and again anticipates that the block has a *near-immediate* re-reference interval. In effect, LRU predicts that cache blocks are re-referenced in the reverse-order of reference, i.e., LRU predicts that a Most Recently Used (MRU) cache block will be re-referenced much sooner than an LRU cache block.

While LRU provides good performance for workloads with high data locality, LRU limits performance when the prediction of a *near-immediate* re-reference interval is incorrect. Applications whose re-references only occur in the distant future perform badly under LRU. Such applications correspond to situations where the application working set is larger than the available cache or when a burst of references to non-temporal data discards the active working set from the cache. In both scenarios, LRU inefficiently utilizes the cache since newly inserted blocks have no temporal locality after insertion.

The Dynamic Insertion Policy (DIP) [25] improves LRU replacement in situations where the re-reference interval is in the

1. We are viewing the RRIP chain like a "snake" where, for LRU, new cache blocks enter the head and leave at the tail.

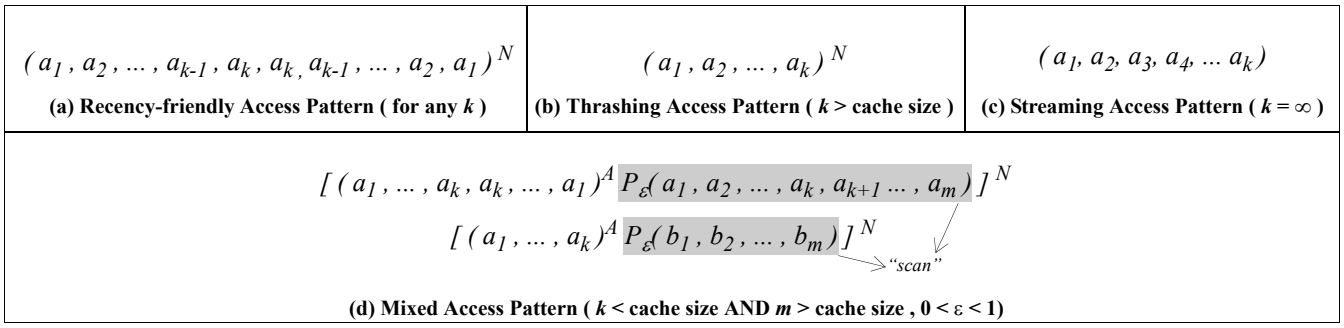

Figure 1: Common Cache Access Patterns.

distant future by dynamically changing the re-reference prediction from a *near-immediate* re-reference interval to a *distant* re-reference interval. At the time of a cache miss, the LRU Insertion Policy (LIP) component of DIP predicts that the cache blocks that already reside in the cache will be re-referenced sooner than the missing cache block. As a result, when the working set is larger than the available cache, LIP preserves part of the working set in the cache by replacing the most recently filled cache block instead of using LRU replacement. DIP dynamically uses LIP for workloads whose working set is larger than the available cache and relies on LRU for all other workloads.

Unfortunately, DIP makes the same predictions for all references of a workload. The LRU component of DIP predicts that *all* re-references to missing cache blocks will be *near-immediate* and inserts them at the head of the RRIP chain. On the other hand, the LIP component of DIP predicts that *all* re-references to missing cache blocks will be in the *distant* future and inserts them at the tail of the RRIP chain. Consequently, when the workload re-reference pattern is mixed, i.e., both *near-immediate* and *distant* re-references occur, neither LRU nor DIP can make accurate predictions. For example, both DIP and LRU limit cache performance when *scans* [5] discard the frequently referenced working-set of an application from the cache. A *scan* is defined as a burst of references to data whose re-reference interval is in the *distant* future. In comparison, accesses that do not belong to the scan have a *near-immediate* re-reference interval. Our studies show that many real world applications suffer from frequent scans in their cache access patterns. Consequently, improving the performance of real world applications require a practical scan-resistant cache replacement policy.

Scans, regardless of their length, do not receive cache hits after their initial reference. This is because the re-reference interval of a scan block is in the *distant* future. The situation may be different for the blocks resident in the cache when the scan first starts. When the data referenced after the scan is different from the data referenced before the scan, replacement decisions during the scan are irrelevant because the references to the new data cause compulsory misses[2]. However, when the data referenced after the scan belongs to the working set prior to the scan, the optimal replacement policy *knows* that a *distant* re-reference interval be applied to cache blocks belonging to the scan and a *near-immediate* re-reference interval be applied to cache blocks belonging to the working set. In doing so, the optimal replacement policy preserves the frequently referenced working set in the cache after the scan completes. Practical replacement policies can potentially accomplish this by using LIP-style replacement during the course of the scan and LRU replacement in the absence of the scan.

In comparison to prior scan-resistant replacement algorithms [5, 13, 17, 22, 27, 29], this paper focuses on designing a high performing scan-resistant replacement policy that requires low hardware overhead, retains the existing cache structure, and most importantly integrates easily into existing hardware approximations of LRU [1, 2]. To that end, we propose a practical replacement policy that uses *Re-reference Interval Prediction (RRIP)*.

RRIP prevents cache blocks with a *distant* re-reference interval (i.e., scan blocks) from evicting blocks that have a *near-immediate* re-reference interval (i.e., non-scan blocks). RRIP accomplishes this by requiring an M-bit register per cache block to store its *Re-reference Prediction Value (RRPV)*. We propose *Static RRIP (SRRIP)* that is scan-resistant and *Dynamic RRIP (DRRIP)* that is both scan-resistant and thrash-resistant. Both SRRIP and DRRIP improve performance over LRU and easily integrate into existing hardware approximations for LRU. In fact, when M=1, SRRIP degenerates to the Not Recently Used (NRU) [2] replacement policy commonly used in modern high performance processors [1, 2].

The rest of this paper is organized as follows, Section 2 motivates the need for a scan-resistant replacement algorithm, Section 3 provides related work, Section 4 introduces RRIP, Section 5 provides the experimental methodology, Section 6 presents results, and finally Section 7 summarizes the paper.

2. MOTIVATION

Efficient last-level cache (LLC) utilization is crucial to avoid long latency cache misses to main memory. Under LRU replacement, many studies have illustrated that the filtering of temporal locality by small caches cause the majority of blocks inserted into the LLC to never be re-referenced [14, 16, 25, 30]. The inefficient cache utilization is because LRU performs poorly for the cache access patterns resulting from the filtered temporal locality.

To better understand when LRU performs poorly, Figure 1 presents several representative cache access patterns commonly found in applications. Let a_i denote the address of a cache line, $(a_1, \ldots, a_k)$ denote a temporal sequence of references to k unique addresses and let $P_\varepsilon(a_1, \ldots, a_k)$ denote a temporal sequence that occurs with some probability ε. A temporal sequence that repeats N times is represented as $(a_1, \ldots, a_k)^N$. The cache access patterns can be classified into the following categories:

- Recency-friendly Access Patterns: Figure 1a presents a typical stack access pattern that repeats N times. In general, recency-friendly access patterns have a *near-immediate* re-reference interval. For any value of k, the access pattern benefits from LRU replacement. Any other replacement policy besides LRU can degrade the performance of these access patterns.
- Thrashing Access Patterns: Figure 1b presents a cyclic access pattern of length k that repeats N times. When k is less than or equal to the number of blocks in the cache, the working set fits into the cache. However, when k is larger than the number of cache blocks, LRU receives zero cache hits due to cache thrashing. For such patterns, LRU provides no cache hits unless

2. Compulsory misses cannot be reduced under any replacement policy.

the cache size is increased to hold all k entries of the access pattern. When the available cache is less than k entries, the optimal replacement policy preserves some of the working set in the cache. Unfortunately, LRU is incapable of doing so.

- Streaming Access Patterns: Figure 1c presents a streaming access pattern. When $k = \infty$, the access pattern has no locality in its references. Streaming access patterns can be characterized as workloads that have *infinite re-reference* interval. Consequently, streaming access patterns receive no cache hits under any replacement policy. As a result, LRU is adequate since replacement decisions are irrelevant in the presence of streaming access patterns.
- Mixed Access Patterns: Mixed access patterns can be characterized as workloads where some references have a *near-immediate* re-reference interval while other references have a *distant* re-reference interval. Figure 1d illustrates this using two example access patterns that have scans (highlighted by the grey box in the figure). Both examples include an access pattern of length k that repeats A times followed by a reference to a sequence of length m with probability ε. Both the scan and access pattern repeat N times. The first reference pattern is representative of an application that performs operations on a linked list of m entries. The initial stack reference pattern illustrates operations that have temporal locality to the beginning of the linked list. The scan illustrates a search or update operation that requires traversing the entire list. The second example is representative of an application that operates on a data structure of k entries and then updates a different data structure of m entries. For both access patterns, when $m + k$ is less than the available cache, the total working set fits into the cache and LRU works well. However, when $m + k$ is greater than the available cache, LRU discards the frequently referenced working set from the cache. Consequently, accesses to the frequently referenced working set always misses after the scan. In the absence of scans, mixed access patterns prefer LRU. However, in the presence of scans, the optimal policy preserves the active working set in the cache after the scan completes. Unfortunately, LRU cannot preserve the active working set.

For the access patterns described above, there is room to improve LRU for thrashing and mixed access patterns. DIP [25] addresses the cache thrashing problem by preserving some of the working set in the cache. Unfortunately, DIP only targets workloads that have a working set larger than the available cache and relies on LRU for all other workloads. As a result, DIP limits performance of workloads where frequent scans discard the active working from the cache. To illustrate this problem, Figure 2 compares the cache performance of thrash-resistant DIP to scan-resistant $HYB_{LRU/LFU}$. $HYB_{LRU/LFU}$ is a hybrid cache replacement policy that uses Set Dueling [25] to dynamically choose between the scan-resistant *Least Frequently Used (LFU)*[3] [17] replacement policy and LRU replacement. The study consists of 14 workloads each running on a 2MB LLC. The y-axis presents the average reduction in cache misses compared to LRU while the x-axis presents the applications and their categories. The application categories under study are *PC games*, *multimedia*, *server*, and *SPEC CPU 2006*.

Figure 2 shows that DIP outperforms LFU for two workloads from the SPEC CPU2006 category. However, LFU significantly outperforms DIP for workloads in the PC games and multimedia categories (and the *hmmer* workload in the SPEC CPU2006 category). For example, PC games and multimedia workloads observe no benefit from thrash-resistant DIP but observe up to 20% reduction in cache misses from scan-resistant LFU. The results in Figure 2 motivate the need for a practical cache replacement policy that is not just thrash-resistant but also scan-resistant. Section 4 discusses the design of such a replacement policy.

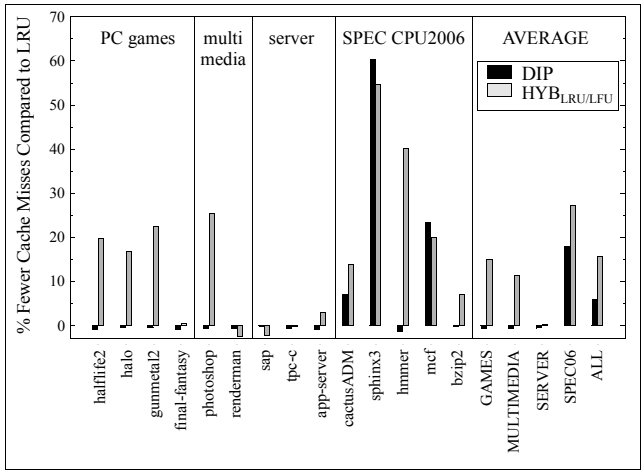

Figure 2: Comparing LRU to Thrash-Resistant DIP and Scan-Resistant LFU.

3. RELATED WORK

Both industry and academia have produced an impressive amount of research work dedicated to improving the performance of replacement policies. While we cannot describe all replacement policies that exist in the literature, we summarize prior art that most closely relates to improving LLC performance by targeting cache blocks that are dead upon cache insertion.

Dead blocks brought in by scans have commonly been addressed by using *access frequency* to predict the re-reference pattern. The proposed *Least Frequently Used (LFU)* [17] replacement policy predicts that blocks that are frequently accessed will be re-referenced in the *near-immediate* future while blocks infrequently accessed will be re-referenced in the *distant* future. LFU accomplishes this by using counters to measure a block's access frequency. While LFU improves the performance of workloads with frequent scans, it significantly degrades the performance of workloads where *recency* is the preferred choice for replacement. Several studies have combined recency and frequency [17, 23, 27] to address the problem but they require several parameters to be tuned on a per-workload basis. Self-tuning adaptive policies [22, 5, 29] exist, however they significantly increase the hardware overhead and complexity. The hardware overhead and complexity can be reduced via hybrid cache replacement [25]. Hybrid cache replacement uses *set dueling* [25] to dynamically choose between multiple replacement policies. While hybrid cache replacement using LRU and LFU can provide scan-resistance, hybrid replacement requires hardware and verification overhead for two *different* cache replacement policies. It would be highly desirable that a single cache replacement policy provide scan resistance and perform well for recency friendly workloads.

Another area of research predicts when the re-reference interval of a cache block becomes *distant*, i.e., a cache block becomes dead [16, 18]. A recent study applied dead block prediction at the LLC [19]. The proposed policy attaches a prediction with each cache block to determine whether or not the block is dead. The proposed policy uses the block's re-reference history to predict *death* after the block moves out of the head of the RRIP chain. The victim selection policy selects dead blocks closer to the tail of the RRIP chain. While dead block prediction improves cache performance, it requires additional hardware overhead for the dead block predictor.

3. Access frequency is measured by using a 4-bit counter per cache block. All LFU counters in the set are halved whenever any counter saturates.

A recent study [25] shows that dead blocks occur when the application working set is larger than the available cache. In such scenarios, the proposed *Dynamic Insertion Policy (DIP)* [25] dynamically changes the insertion policy from always inserting blocks at the head of the RRIP chain to inserting the majority of the blocks at the tail of the RRIP chain. By doing so, DIP preserves some of the working set in the cache. Since DIP makes a single insertion policy decision for *all* references of a workload, DIP only targets workloads whose working set is larger than the available cache. Consequently, in the presence of scans, the LRU component policy of DIP is unable to preserve the active working set in the cache.

Another recent study proposes pseudo-LIFO [8] based replacement policies. The policy proposes cache replacement using a fill stack as opposed to the recency stack. The proposed policy learns the re-reference probabilities of a cache block beyond each fill stack position and finds that evicting blocks from the upper portion of the fill stack improves cache utilization by evicting dead blocks quickly. The proposed policy however requires additional hardware to keep track of a block's fill stack position and also requires a dynamic mechanism to learn the best eviction position on the fill stack.

Various other solutions [6, 13, 20, 26, 31] exist but they either require significant additional hardware or they drastically change the organization of the existing cache. Reuse distance prediction [15] most closely resembles the work presented in this paper. Reuse distance prediction explicitly calculates the reuse distance of a given cache block by using a PC indexed predictor. RRIP does not explicitly calculate reuse distance. Instead, RRIP always predicts that all missing cache block will have the same re-reference interval and updates the prediction when more information is available, for example, on a re-reference. RRIP also differs from prior work in that it proposes a high performing practical scan-resistant cache replacement policy that does not require significant hardware overhead or changes to the existing cache structure.

4. RE-REFERENCE INTERVAL PREDICTION (RRIP)

When workloads have mixed access patterns, LRU replacement and its approximations cannot perfectly distinguish between blocks that have a *distant* re-reference interval from blocks that have a *near-immediate* re-reference interval. Since chain-based LRU replacement is impractical to build in hardware for highly associative caches, we illustrate the problem for mixed access patterns using the *Not Recently Used (NRU)* [1, 2] replacement policy.

4.1. Not Recently Used (NRU) Replacement

The *Not Recently Used (NRU)* replacement policy is an approximation of LRU commonly used in modern high performance processors. NRU uses a single bit per cache block called the *nru*-bit[4]. With only one bit of information, NRU allows two re-reference interval predictions: *near-immediate* re-reference and *distant* re-reference. An nru-bit value of '0' implies that a block was recently used and the block is predicted to be re-referenced in the *near-immediate* future. An nru-bit value of '1' implies that the block was not recently used and the block is predicted to be re-referenced in the *distant* future. On cache fills, NRU always predicts that the missing block will have a *near-immediate* re-reference. Upon re-reference, NRU again anticipates that the block referenced will have a *near-immediate* re-reference. On a cache miss, NRU selects the victim cache block whose predicted re-reference is in the *distant* future, i.e., a block whose nru-bit is '1'. Because multiple blocks may have a *distant* re-reference prediction, a tie-breaker is needed. The NRU victim selection policy always starts the victim search from a fixed location (the left in our studies). In the event that all nru-bits are '0', i.e., all blocks are predicted to be re-referenced in the *near-immediate* future, NRU updates the re-reference predictions of all cache blocks to be in the *distant* future and repeats the victim search. Updating all nru-bits to '1' allows the victim selection policy to make forward progress while also removing stale blocks from the cache.

Figure 3b illustrates the behavior of NRU using a 4-entry cache initialized with invalid blocks 'I'. For reference, Figure 3a also shows the behavior of LRU. The figure also shows the steps taken by the replacement policy on cache hits and cache misses. We use the following bimodal access pattern to illustrate NRU behavior:

$$(a_1, a_2, a_2, a_1)(b_1, b_2, b_3, b_4)(a_1, a_2, \ldots)$$

Figure 3b shows the four blocks of the cache each with the nru-bit shown in the lower right hand corner. The figure shows that after the scan completes, references to a_1 and a_2 both miss the cache when the optimal replacement policy would have preserved them in the cache.

4.2. Static RRIP (SRRIP)

With only one bit of information, NRU can predict either a *near-immediate* re-reference interval or a *distant* re-reference interval for all blocks filled into the cache. Always predicting a *near-immediate* re-reference interval on all cache insertions limits cache performance for mixed access patterns because scan blocks unnecessarily occupy the cache space without receiving any cache hits. On the other hand, always predicting a *distant* re-reference interval significantly degrades cache performance for access patterns that predominantly have a *near-immediate* re-reference interval. Consequently, without any external information on the re-reference interval for every missing cache block, NRU cannot identify and preserve non-scan blocks in a mixed access pattern.

To address the limitations of NRU, we enhance the granularity of the re-reference prediction stored with each cache block. We propose cache replacement based on *Re-reference Interval Prediction (RRIP)*. RRIP uses M-bits per cache block to store one of 2^M possible *Re-reference Prediction Values (RRPV)*. RRIP dynamically *learns* re-reference information for each block in the cache access pattern. Like NRU, an RRPV of zero implies that a cache block is predicted to be re-referenced in the *near-immediate* future while RRPV of saturation (i.e., 2^M–1) implies that a cache block is predicted to be re-referenced in the *distant* future. Quantitatively, RRIP predicts that blocks with small RRPVs are re-referenced sooner than blocks with large RRPVs. When M=1, RRIP is identical to the NRU replacement policy. When M>1, RRIP enables *intermediate* re-reference intervals that are greater than a *near-immediate* re-reference interval but less than a *distant* re-reference interval.

The primary goal of RRIP is to prevent blocks with a *distant* re-reference interval from polluting the cache. In the absence of any external re-reference information, RRIP statically predicts the block's re-reference interval. Since always predicting a *near-immediate* or a *distant* re-reference interval at cache insertion time is not robust across all access patterns, RRIP always inserts new blocks with a *long* re-reference interval. A *long* re-reference interval is defined as an *intermediate* re-reference interval that is skewed towards a *distant* re-reference interval. We use an RRPV of 2^M–2 to represent a *long* re-reference interval. The intuition behind always predicting a *long* re-reference interval on cache insertion is to prevent cache blocks with re-references in the *distant* future from polluting the cache. Additionally, always predicting a *long* re-reference interval instead of a *distant* re-reference interval allows RRIP more time to learn and improve the re-reference prediction. If the newly inserted cache block has a *near-immediate* re-reference interval, RRIP can then update the re-reference prediction to be shorter than the previous prediction. In effect, RRIP *learns* the block's re-reference interval.

4. This paper describes NRU replacement by inverting the polarity of the bit to represent not-recently used. instead of recently used.

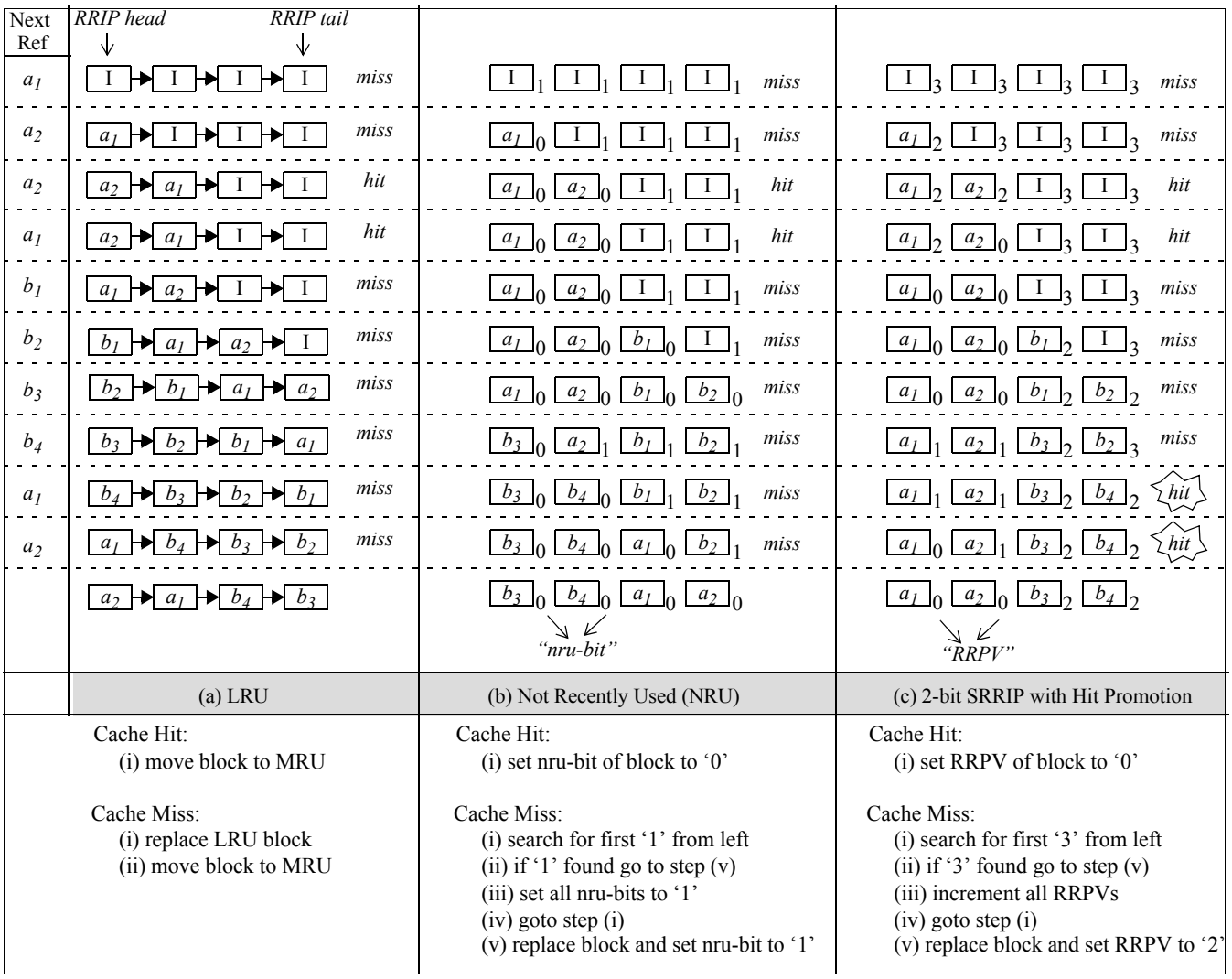

Figure 3: Behavior of LRU, NRU, and SRRIP for a Mixed Access Pattern.

On a cache miss, the RRIP *victim selection policy* selects the victim block by finding the first block that is predicted to be re-referenced in the *distant* future (i.e., the block whose RRPV is 2^M–1). Like NRU, the victim selection policy breaks ties by always starting the victim search from a fixed location (the left in our studies). In the event that RRIP is unable to find a block with a *distant* re-reference interval, RRIP updates the re-reference predictions by incrementing the RRPVs of all blocks in the cache set and repeats the search until a block with a *distant* re-reference interval is found. Updating RRPVs at victim selection time allows RRIP to adapt to changes in the application working set by removing stale blocks from the cache.

A natural opportunity to change the re-reference prediction of a block occurs on a hit to the block. The algorithm for this update of the RRPV register is called the RRIP *hit promotion policy*. The primary purpose of the hit promotion policy is to dynamically improve the accuracy of the predicted re-reference interval of cache blocks. We propose two policies to update the re-reference prediction: *Hit Priority (HP)* and *Frequency Priority (FP)*. The RRIP-HP policy predicts that the block receiving a hit will be re-referenced in the *near-immediate* future and updates the RRPV of the associated block to zero. The goal of the HP policy is to prioritize replacement of blocks that do not receive cache hits over *any* cache block that receives a hit. However, the HP policy can potentially degrade cache performance when a cache block is re-referenced only once after cache insertion. In such situations, the HP policy incorrectly predicts a *near-immediate* re-reference prediction instead of *distant* re-reference prediction for the block and causes the block to occupy valuable cache space without receiving any hits. To address this problem, the RRIP-FP policy uses more information (i.e., cache hits) to update the re-reference prediction. Instead of updating the re-reference prediction to be *near-immediate* on a hit, RRIP-FP updates the predicted re-reference interval to be shorter than the previous re-reference interval each time a block receives a hit. The FP policy accomplishes this by decrementing the RRPV register (unless the RRPV register is already zero) on cache hits. The goal of the FP policy is to prioritize replacement of infrequently re-referenced cache blocks over frequently re-referenced cache blocks.

Since the re-reference predictions made by RRIP are statically determined on cache hits and misses, we refer to this replacement policy as *Static Re-reference Interval Prediction (SRRIP)*. Figure 3c illustrates the behavior of 2-bit SRRIP-HP. The example shows that SRRIP emulates optimal replacement by correctly predicting a *near-immediate* re-reference interval for the actively used cache blocks and a *distant* re-reference interval for the scan blocks.

In general, for associativity A, active working set size w ($w < A$), and scan length S_{len}, M-bit SRRIP is scan-resistant when

$$S_{len} <= (2^M - 1) * (A - w) \quad \text{(Eq. 1)}$$

When the condition in Equation 1 does not hold, SRRIP is unable to preserve the active working set in the cache because the RRPVs of the scan blocks and the non-scan blocks become identical due to the aging mechanism of the victim selection policy. In such scenarios, the active working set can be preserved for a longer time by increasing the width of the RRPV register. While large RRPVs can be resistant to long scans, they can result in inefficient cache utilization when a cache block receives its last hit and the RRPV becomes zero. In such situations the effective cache capacity reduces until the victim selection policy updates the re-reference prediction of the dead block to have a *distant* re-reference. Consequently, scan-resistance using RRIP requires that the width of the RRPV register to be appropriately sized to avoid sources of performance degradation.

4.3. Dynamic RRIP (DRRIP)

SRRIP inefficiently utilizes the cache when the re-reference interval of all blocks is larger than the available cache. In such scenarios, SRRIP causes cache thrashing and results in no cache hits. To avoid cache thrashing, we propose *Bimodal RRIP (BRRIP)* that inserts majority of cache blocks with a *distant* re-reference interval prediction (i.e., RRPV of 2^M–1) and infrequently (with low probability) inserts new cache blocks with a *long* re-reference interval prediction (i.e., RRPV of 2^M–2). BRRIP is analogous to the Bimodal Insertion Policy (BIP) [25] component of DIP which helps preserve some of the working set in the cache.

For non-thrashing access patterns, always using BRRIP can significantly degrade cache performance. In order to be robust across all cache access patterns, we propose to dynamically determine whether an application is best suited to scan-resistant SRRIP or thrash-resistant BRRIP. We propose *Dynamic Re-reference Interval Prediction (DRRIP)* that uses *Set Dueling* [25] to identify which replacement policy is best suited for the application. DRRIP dynamically chooses between scan-resistant SRRIP and thrash-resistant BRRIP by using two *Set Dueling Monitors (SDMs)* [11]. An SDM estimates the misses for any given policy by permanently dedicating a few sets[5] of the cache to follow that policy. Set Dueling uses a single policy selection (PSEL) counter to determine the winning policy. DRRIP uses the winning policy of the two SDMs for the remaining sets of the cache.

4.4. Comparing SRRIP to LRU

A natural way of modifying an LRU managed cache to predict the re-reference interval would be by changing the insertion position of blocks on the LRU chain, creating a RRIP chain. The baseline MRU Insertion Policy (MIP) [25] predicts a *near-immediate* re-reference interval by always inserting new blocks at the head of the RRIP chain. The LRU Insertion Policy (LIP) [25] predicts a *distant* re-reference interval by always inserting new blocks at the tail of the RRIP chain. Insertion positions in the middle of the RRIP chain [28, 30] can be used to predict an *intermediate* re-reference interval. While different insertion positions in the RRIP chain can provide scan-resistance, they require tuning on a per-application basis. A static insertion position for all applications can degrade performance when changes in the working set requires an alternate re-reference prediction, i.e., an alternate insertion position on the RRIP chain.

Unlike SRRIP, modified LRU cannot automatically adapt to changes in the application working set size and thus can degrade performance significantly. Set dueling can be used to design a mechanism that dynamically identifies the best insertion position on the RRIP chain suitable to the application (or application phase). Specifically, SDMs can *monitor* the performance of different insertion positions on the RRIP chain and then *apply* the best position to the remainder of the cache. However, using SDMs to identify the best insertion position on the RRIP chain does not scale well with increasing cache associativity. This is because the number of monitor sets required can exceed the total number of sets in the cache. As a result, set dueling controlled policies to identify the best insertion position on the RRIP chain are not considered as a practical solution to provide scan resistance. Nonetheless, we compare SRRIP and DRRIP to an offline profiling mechanism that *knows* the best single insertion location on the RRIP chain on a per application basis. We refer to this scheme as the *Best Offline Insertion Policy (BOIP)*. We do not consider the potentially better and more complex scheme where the insertion position on the RRIP chain dynamically adapts to different phases of an application.

4.5. RRIP Extensions to Shared Caches

With the growing number of cores on-chip, shared caches are now very common. Since shared caches receive access patterns from concurrently executing workloads, the combined access stream from the different workloads can also be thought of as a mixed access pattern. Thus, SRRIP naturally extends to shared caches and can minimize cache contention between applications with varying memory demands. For example, using the mixed access pattern terminology, the references to the *active working set* can potentially be described as memory references by an application whose working set is small and fits in the shared LLC and the *scan* can be described as memory references by an application with a very large working set. In such situations, SRRIP reduces cache contention by preserving the small working set in the shared LLC.

Extending DRRIP to shared caches is analogous to the extension of DIP to shared caches. We propose *Thread-Aware DRRIP (TA-DRRIP)* which is similar to the *Thread-Aware Dynamic Insertion Policy (TA-DIP)* [9]. TA-DRRIP uses two SDMs per application to dynamically determine whether the application should use SRRIP or BRRIP in the presence of other applications. Like TA-DIP, TA-DRRIP merely requires a policy selection counter (PSEL) for each hardware thread sharing the LLC.

5. EXPERIMENTAL METHODOLOGY

5.1. Simulator

We use CMP$im [10], a Pin [21] based trace-driven x86 simulator for our performance studies. Our baseline processor is 4-way out-of-order with a 128-entry reorder buffer and a three level cache hierarchy. Only the LLC of the hierarchy enforces inclusion. Our cache hierarchy is roughly comparable to the Intel Core i7 [3]. The L1 instruction and data caches are 4-way 32KB each while the L2 cache is unified 8-way 256KB. The L1 and L2 cache sizes are kept constant in our study. We support two L1 read ports and one L1 write port on the data cache. We evaluate both single-core and 4-core configurations. In the 4-core configuration, the L1 and L2 are private and only the LLC is shared by all four cores. The baseline LLC (L3) is 16-way 2MB in the single-core and 8MB in the 4-core system. All caches in the hierarchy use a 64B line size. For replacement decisions, all caches in the hierarchy use the LRU replacement policy. Only demand references to the cache update the LRU state while non-demand references (e.g., write back references) leave the LRU state unchanged. The load-to-use latencies for the L1, L2, and L3 caches are 1, 10, and 24 cycles respectively. We model a 250 cycle penalty to main memory and support a maximum of 32 outstanding misses to memory.

5. Prior work has shown that 32 sets are sufficient to estimate cache performance [11]. Throughout the paper an SDM consists of 32 sets.

Table 1: Benchmarks

Category	Workloads
PC Games	final-fantasy, gunmetal2, halflife2, halo
Multimedia	photoshop, renderman
Server	app-server, sap, tpc-c
SPEC CPU2006	bzip2, cactusADM, hmmer, mcf, sphinx3

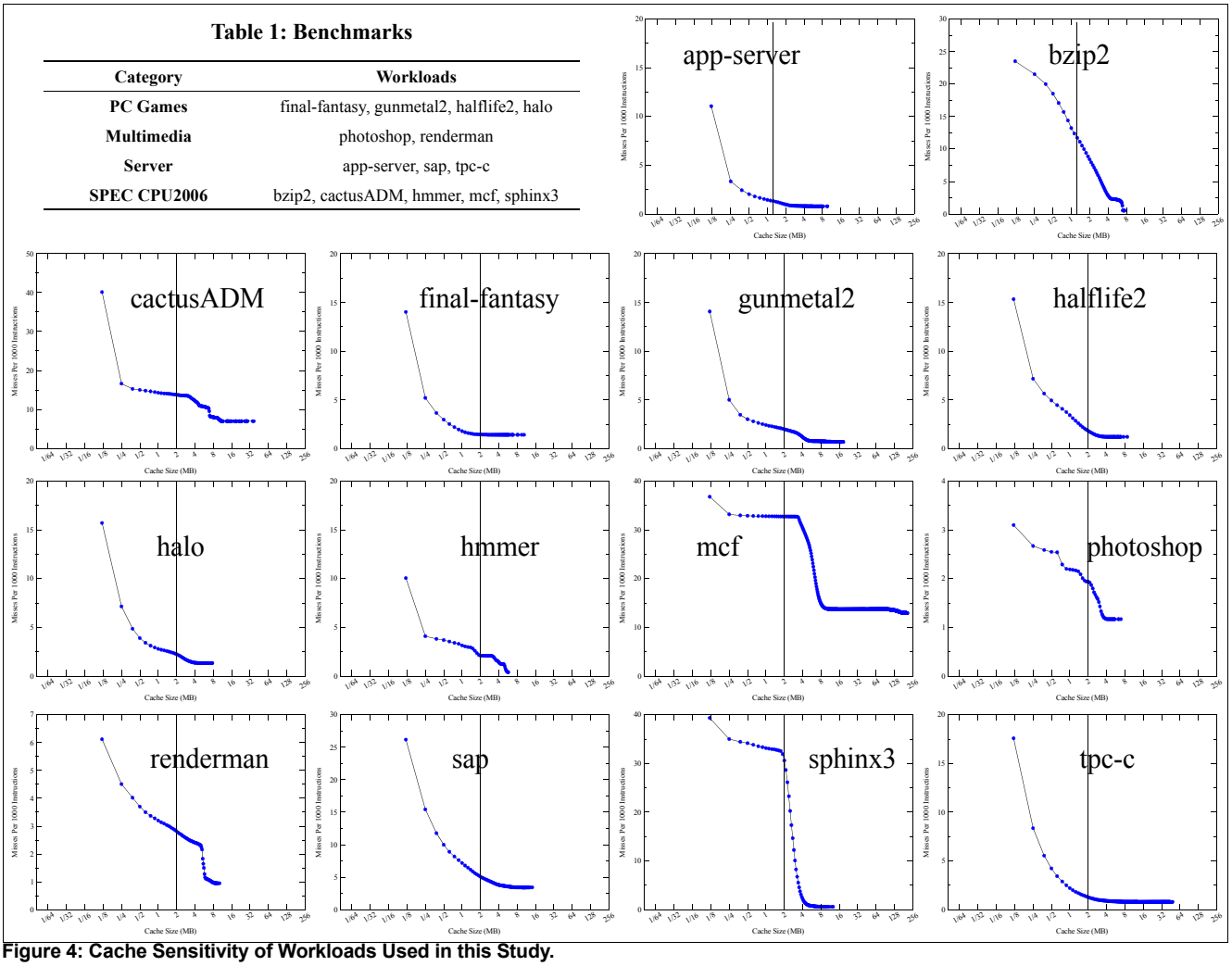

Figure 4: Cache Sensitivity of Workloads Used in this Study.

5.2. Benchmarks

For our single-core studies we use five workloads from the SPEC CPU2006 benchmark suite and nine "real world" workloads from the PC game, multimedia, and server workload segments. These workloads were selected because they are sensitive to memory latency on the baseline processor configuration and there is an opportunity to improve their performance through enhanced replacement decisions. To be thorough, we report results across a broad set of memory intensive and non-memory intensive workloads in the appendix. The SPEC CPU2006 workloads were all collected using PinPoints [24] for the reference input set while the real world workloads were all collected on a hardware tracing platform. The real world workloads include both operating system and user-level activity while the SPEC CPU2006 workloads only include user-level activity. Table 1 lists the workloads used and Figure 4 provides their sensitivity to different cache sizes. The workloads were all run for 250M instructions.

For our multi-core workloads, we created all possible 4-core combinations (14 choose 4 = 1001 workloads). Simulations were run until *all* benchmarks ran 250 million instructions. Statistics for each core were collected only for the first 250 million instructions. If the end of the trace is reached, the model rewinds the trace and restarts from the beginning. The simulation methodology is similar to recent work on shared caches [11, 8, 30, 20].

6. RESULTS AND ANALYSIS

6.1. SRRIP Sensitivity to RRPV on Insertion

Figure 5 shows the sensitivity of SRRIP-HP to the width of the M-bit register and the *Re-reference Prediction Value* on cache insertion when both are changed statically. The y-axis represents the percent reduction in cache misses compared to LRU replacement. For M=1 (NRU), 2, 3, 4, and 5, the x-axis shows all 2^M possible RRPVs for cache insertion. The x-axis labels follow the format "INS=r, M=m" and denotes an m-bit SRRIP configuration where all missing cache blocks are inserted with an RRPV of 'r'. For each SRRIP configuration, the figure also shows the maximum, average, and minimum values for the reduction in cache misses across all workloads. The average is represented by squares while the minimum and maximum values are represented by triangles.

Figure 5 shows that when M>1, always predicting that a missing cache block has a *long* re-reference interval has the best performance. In fact, predicting a *long* re-reference interval consistently outperforms NRU replacement (M=1). This is because RRIP enhances the granularity for predicting the re-reference interval. By always predicting a *long* re-reference interval, cache blocks that do not have temporal locality (i.e., scan blocks) do not pollute the cache for an extended period of time. Always predicting a *distant* re-reference interval has the worst performance because SRRIP does not

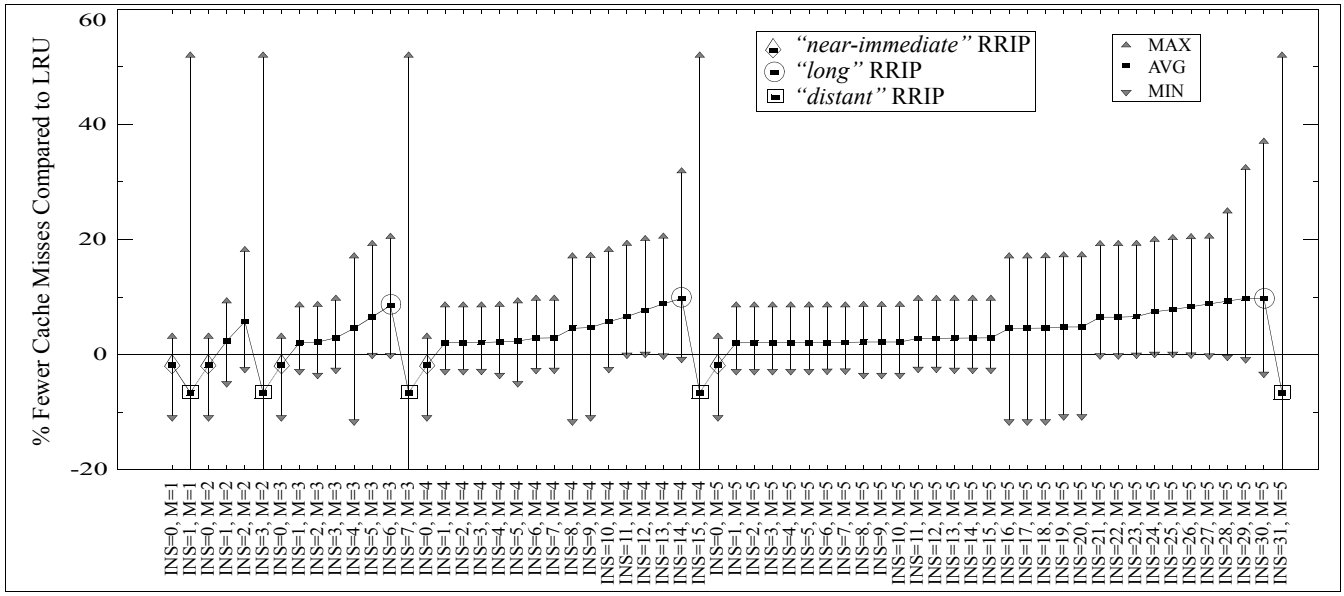

Figure 5: Re-reference Prediction Value (RRPV) Sensitivity Study for SRRIP-HP.

have enough time to improve the blocks re-reference interval. While always predicting *distant* re-reference interval has positive outliers, always predicting a *long* re-reference interval is robust across all workloads and reduces cache misses relative to LRU by 6-10%.

6.2. SRRIP Performance

Figure 6 presents the per-workload behavior for SRRIP-HP and SRRIP-FP for M=1, 2, 3, 4 and 5. Both SRRIP policies always predict a *long* re-reference interval on cache insertion. The x-axis shows the different workloads while the y-axis shows the reduction in cache misses. The x-axis labels GAMES, MULTIMEDIA, SERVER, and SPEC06 represent the average for the workloads in these categories while ALL is the average of all 14 workloads. We use arithmetic mean for cache performance and geometric mean for system performance. Figure 6a shows that SRRIP-FP reduces MPKI by 5-18%. The reductions in MPKI allow SRRIP-FP to outperform

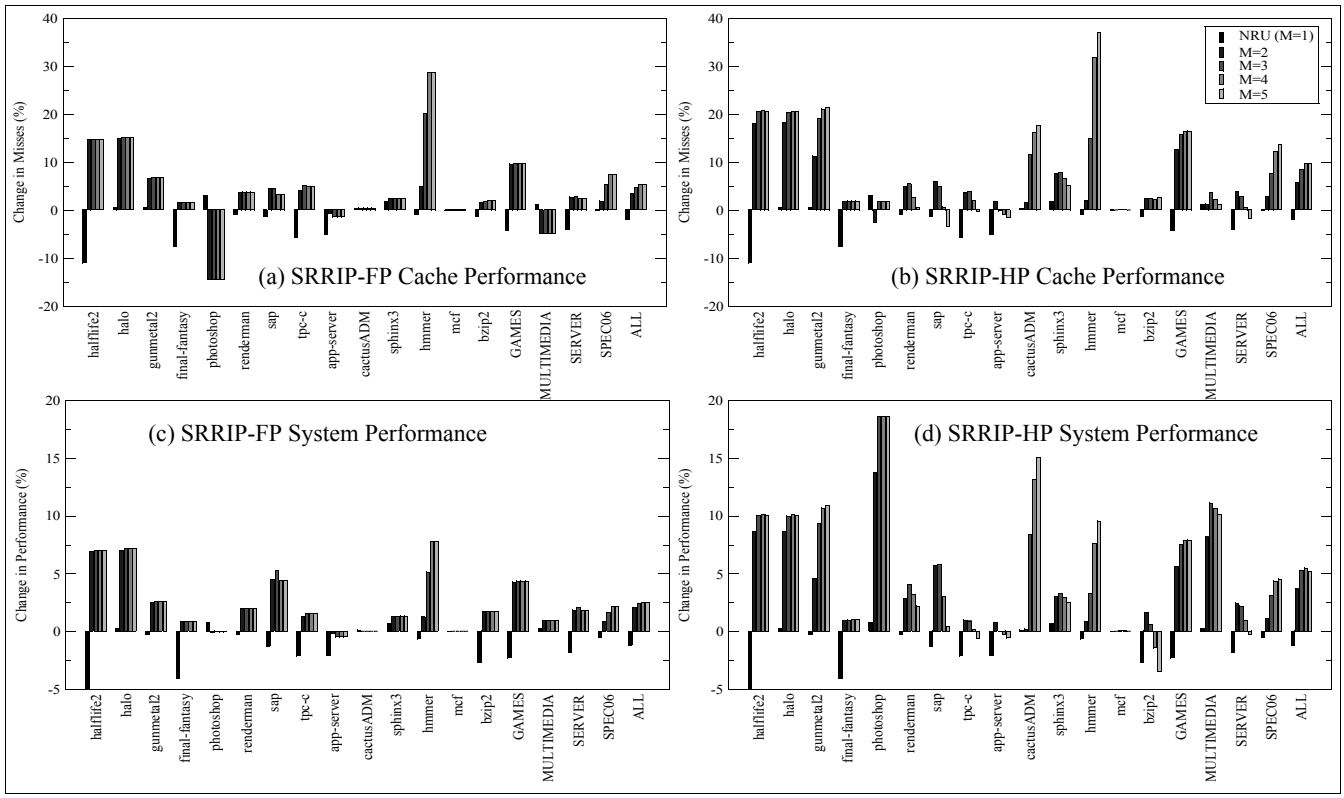

Figure 6: SRRIP Sensitivity to Width of M-bit register.

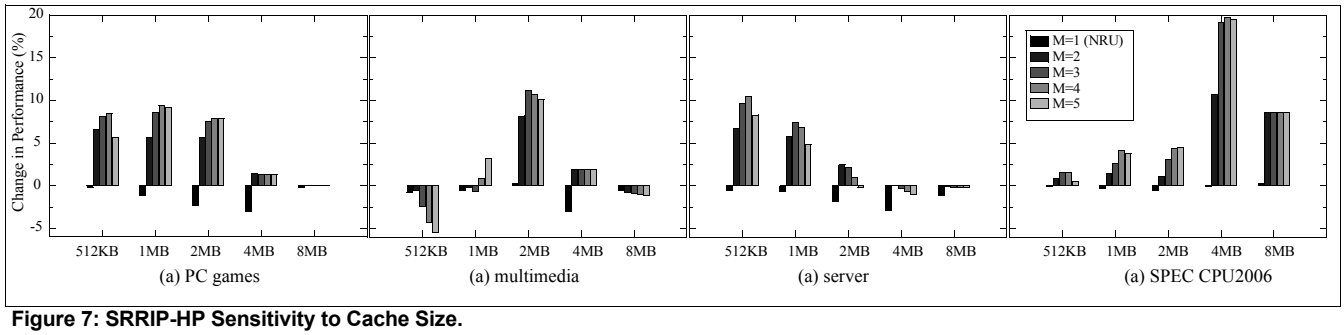

Figure 7: SRRIP-HP Sensitivity to Cache Size.

LRU by an average of 2.5% across all workloads (Figure 6c). PC games receive the most benefit where SRRIP-FP outperforms LRU by 4% on average. On the other hand, SRRIP-HP reduces MPKI by 5-15% for eight of the fourteen workloads (Figure 6b). The reductions in MPKI allow SRRIP-HP to outperform LRU by 5% across all workloads (Figure 6d). PC games and multimedia workloads receive the most benefit over LRU by 8-11%. These results are consistent with Figure 2 where PC games and multimedia workloads benefitted the most from scan-resistance.

On average, SRRIP is insensitive to the width of the RRPV register when M>3. Some workloads experience performance degradation when the width of the RRPV register increases. This is because wider RRPV registers retain stale blocks in the cache for long periods of time (after their last hit) and reduce the effective cache capacity. For the workloads in this study, 2-bit or 3-bit RRPV is sufficient to be scan-resistant.

Finally, both SRRIP-HP and SRRIP-FP outperform LRU. NRU (M=1) almost always performs worse than LRU. Additionally, SRRIP-HP provides twice the performance benefits of SRRIP-FP. This implies that the first order benefit of a scan-resistant replacement algorithm is not from precisely detecting frequently referenced data in the cache but from preserving data that receives cache hits, i.e., the active working set. For the rest of the paper, unless otherwise stated, we only provide results for SRRIP-HP.

6.3. SRRIP Sensitivity to Cache Configuration

Figure 7 presents SRRIP performance for the different workload categories on different LLC sizes: 512KB, 1MB, 2MB, 4MB, and 8MB. All LLCs are 16-way set associative. The y-axis shows the performance relative to LRU replacement of the respective LLC. The figure shows that NRU (M=1) always performs similar to LRU for all cache sizes. However, SRRIP outperforms LRU by 5-20% for various cache sizes. We also conducted a SRRIP sensitivity study by varying the cache associativity from 4-way to 128-way. Our studies yielded results comparable to Figure 6. These results show that SRRIP is scalable to different cache configurations. Since the majority of performance gains is achieved by a 3-bit RRPV register, we focus only on 2-bit and 3-bit SRRIP.

6.4. DRRIP Performance

Figure 8 presents the performance of 2-bit and 3-bit DRRIP[6]. Figure 8a shows that DRRIP significantly improves cache performance for SPEC CPU2006 workloads *sphinx3*, *hmmer*, and *mcf*. These workloads have a *knee* in the working set that is slightly larger than a 2MB cache (see Figure 4). PC games and multimedia workloads also benefit from a reduction in cache misses. Server workloads on the other hand have no *knee* in the working set, hence observe no benefit from DRRIP. Across most workloads, DRRIP has similar or better performance than SRRIP. DRRIP only hurts *photoshop* performance despite the 10% reduction in cache misses. Further analysis showed that *photoshop* is extremely sensitive to a region of memory that is frequently referenced between scans. Since DRRIP optimizes for the cache miss metric and not the throughput metric, DRRIP can degrade performance when the cost of a miss varies in an application. Enhancing DRRIP to optimize for throughput instead of cache misses can address the problem for *photoshop*. Nonetheless, on average, DRRIP improves performance by an additional 5% above SRRIP. Since both 2-bit and 3-bit DRRIP perform similarly, we conclude that 2-bit DRRIP is sufficient for scan-resistance and thrash-resistance. Thus, for the remainder of the paper we only focus on 2-bit RRIP.

6. We use 32-entry SDMs, 10-bit PSEL counter and ε=1/32. [11, 25]

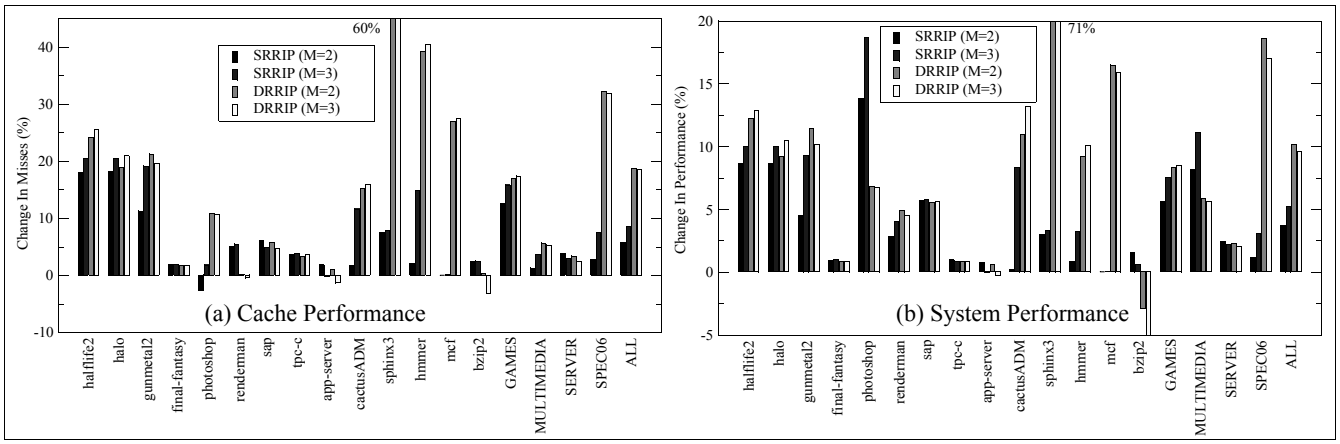

Figure 8: DRRIP Performance.

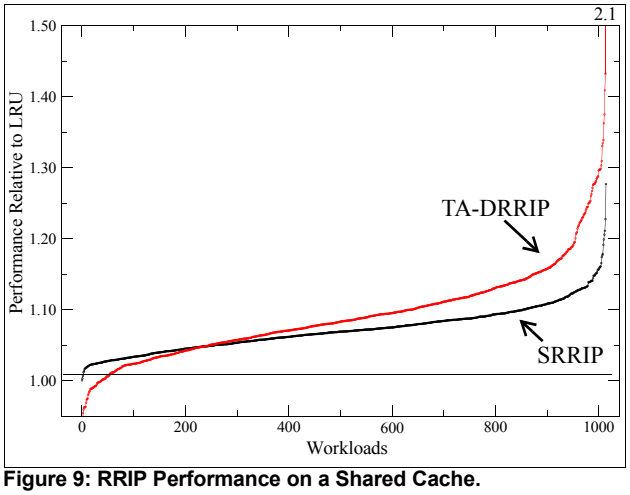

Figure 9: RRIP Performance on a Shared Cache.

6.5. RRIP on Shared Caches

Figure 9 presents the s-curve for the performance of SRRIP and TA-DRRIP compared to LRU for the throughput metric. The x-axis represents all 1001 multi-programmed workloads while the y-axis represents the performance relative to LRU. SRRIP improves performance up to 25% on the 4-core CMP while TA-DRRIP improves performance by as much 2.1X. Across all workloads in the study, SRRIP does not degrade performance for any of the workloads while TA-DRRIP degrades performance by 2-5% for less than 25 workloads due to cost of experimenting with the BRRIP SDMs. On average, across the 1001 multi-programmed workloads, SRRIP improves performance by 7% while TA-DRRIP improves performance by 10%. Thus, these results show that both SRRIP and TA-DRRIP are both robust and high performing.

6.6. RRIP at Different Cache Levels

We compared the performance of SRRIP to LRU when applied at the L1 and L2 caches of our three-level hierarchy. At the L1 cache, SRRIP provides no opportunity to improve performance because the cache size is too small and the temporal locality is too high. At the L2 cache, SRRIP provides no significant performance gains because the L2 cache is small (256KB in our study). SRRIP did not degrade performance of the L1 or L2 caches. To ensure that SRRIP performs well at the LLC, we modified our hierarchy from a 3-level to a 2-level hierarchy by removing the L2 cache. For this 2-level hierarchy, both SRRIP and DRRIP outperform LRU by 4.8% and 10% respectively. Thus, RRIP is most applicable at the LLC where the temporal locality is filtered by smaller levels of the hierarchy[7].

6.7. Hardware Overhead and Design Changes

RRIP requires a 2-bit register per cache block. RRIP integrates into the existing NRU implementation with minor modifications to the victim selection hardware. The NRU victim selection policy searches for the first block with *nru*-bit value of '1'. SRRIP on the other hand searches for the first cache block whose re-reference interval is furthest in the future, i.e., the block whose RRPV is the largest in the set. The search can be implemented by replicating the Find First One (FFO) logic. For 2-bit RRIP, four FFO circuits (with appropriate inputs) are required to find pointers to the first '0', '1', '2', and '3' RRPV registers. A priority MUX chooses the output of the appropriate FFO circuit as the victim. In the event that a block with *distant* RRIP is not found, RRIP also requires additional logic to age the RRPV registers. NRU ages cache blocks by simply inverting all the *nru*-bits in the set. SRRIP requires state machine logic to age all the RRPV registers in the set. DRRIP and TA-DRRIP merely require the per-thread 10-bit policy selection (PSEL) counter and the logic for choosing SDMs. The design changes for SRRIP and DRRIP are not on the critical path and thus do not affect the cache access time.

7. Recent studies [7, 18] have evaluated cache configurations where the linesize of the LLC is larger than the linesize of the L1 and L2 caches. For such configurations, DIP and RRIP require modifications to the hit promotion policy to filter the "false temporal locality" observed by the LLC. For example, re-references to different sectors of the large LLC cache line (due to spatial locality) should not update the LRU state while re-references to the same sector of the line should update the LRU state.

Table 3: Comparison of Replacement Policies on Single Core

Replacement Policy	% Performance Improvement Over LRU	Hardware Overhead[a]
LRU	—	$n \cdot \log_2 n$
NRU	-1.19	n
peLIFO	2.85	$2n \cdot \log_2 n + 2n$
SRRIP	3.69	2n
DIP	5.43	n
$HYB_{NRU/LFU}$	7.57	5n
DRRIP	10.18	2n
BOIP	8.13	N/A

a. Assuming an n-way set associative cache, HW overhead is measured in number of bits required per cache set.

6.8. Comparing RRIP to Other Policies

For the 14 workloads in our study, Table 3 compares the performance of the following replacement policies to LRU replacement: NRU, SRRIP, peLIFO [8], DIP, $HYB_{LRU/LFU}$, DRRIP, and BOIP. Figure 10 presents the performance comparison of these replacement policies on a per application basis. DIP uses set dueling to dynamically select between *near-immediate* and *distant* re-reference interval predictions. Both DIP and peLIFO use NRU replacement as the baseline replacement policy. $HYB_{NRU/LFU}$ also uses set dueling to dynamically choose between NRU and LFU replacement. BOIP uses offline profiling information to determine the best static insertion position on the RRIP chain suitable to the application. peLIFO tracks cache hits on the fill stack to guide cache replacement. SRRIP, a non-adaptive policy, outperforms LRU replacement while DRRIP outperforms the best performing scan-resistant hybrid cache replacement policy ($HYB_{NRU/LFU}$) and also a policy that requires profiling information (BOIP).

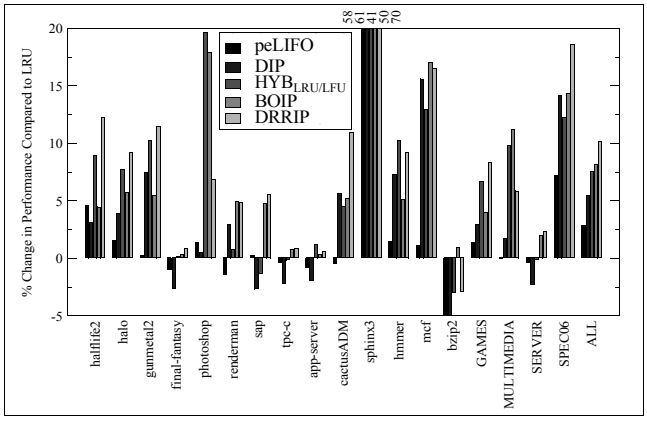

Figure 10: Comparison of Replacement Policies.

We also compared the performance of SRRIP and TA-DRRIP on shared caches to peLIFO and TA-DIP. Both peLIFO and TA-DIP use NRU as the baseline replacement policy. For the 1001 multi-programmed workloads in our study, we found that TA-DIP improved performance relative to LRU by 4% while peLIFO improved performance relative to LRU by 1.5%. SRRIP and TA-DRRIP on the other hand improve performance by 7% and 9% respectively. We believe that our results for peLIFO differ from [8] because we evaluate across a much broader selection of workloads.

RRIP requires less hardware than LRU replacement yet outperforms LRU replacement on average. For an n-way associative cache, LRU replacement requires $n \cdot \log_2 n$ bits per cache set, while RRIP only requires 2n bits per cache set. Compared to $HYB_{NRU/LFU,}$ the LFU component policy requires hardware for the frequency counter and the NRU component requires hardware for tracking recency. Assuming four bits for the LFU frequency counter, $HYB_{NRU/LFU}$ requires 5n bits per cache set. In addition, hybrid replacement also requires verification overhead for designing two *different* replacement policies. Comparatively, SRRIP and DRRIP provide scan-resistance and thrash-resistance in a single replacement policy. RRIP requires 2.5X less hardware than $HYB_{NRU/LFU,\cdot}$

7. SUMMARY

Practical cache replacement policies attempt to emulate optimal replacement by predicting the re-reference interval of a cache block. The commonly used LRU replacement policy always predicts a *near-immediate* re-reference interval on misses and hits. However, the prediction of *near-immediate* re-reference interval inefficiently utilizes the cache when the actual re-reference interval of the missing block is in the *distant* future. When the re-reference interval of all blocks referenced by an application is in the *distant* future, dynamic insertion policies [25] avoid cache thrashing by preserving some of the blocks in the cache. However, when the re-reference interval of blocks accessed by an application consist of mixed access patterns, dynamic insertion policies cannot preserve blocks with *near-immediate* re-reference interval in the cache. This paper shows that many real world game, server, and multimedia applications exhibit such mixed access patterns. Specifically, such applications experience bursts of references to non-temporal data (called *scans*) that discards their active working set from the cache. This paper improves the performance of such real world applications by making the following contributions:

1. We propose cache replacement using *Re-reference Interval Prediction (RRIP)*. RRIP statically predicts the re-reference interval of all missing cache blocks to be an *intermediate* re-reference interval that is between a *near-immediate* re-reference interval and a *distant* re-reference interval. RRIP updates the re-reference prediction to be shorter than the previous prediction upon a re-reference. We call this policy as *Static RRIP (SRRIP)*. We show that SRRIP is scan-resistant and only requires 2-bits per cache block.
2. We propose two SRRIP policies: *SRRIP-Hit Priority (SRRIP-HP)* and *SRRIP-Frequency Priority (SRRIP-FP)*. SRRIP-HP predicts that any cache block that receives a hit will have a *near-immediate* re-reference and thus should be retained in the cache for an extended period of time. SRRIP-FP on the other hand predicts that frequently referenced cache blocks will have a *near-immediate* re-reference and thus they should be retained in the cache for an extended period of time. We show that SRRIP-HP performs significantly better than SRRIP-FP and conclude that scan-resistance is not from precisely detecting frequently referenced blocks but from preventing blocks that receive hits from getting evicted by blocks that do not receive hits (i.e., *scan* blocks).
3. We propose *Dynamic RRIP (DRRIP)* as an enhancement to SRRIP-HP. DRRIP provides both scan-resistance and thrash-resistance by using set dueling to dynamically select between inserting all missing cache blocks with an *intermediate* re-reference interval or with a *distant* re-reference interval. In addition to the hardware overhead of SRRIP, DRRIP does not require any additional hardware overhead besides a single saturating counter.

We show that SRRIP and DRRIP outperform LRU by an average of 4% and 10% on a single-core processor with a 16-way 2MB LLC. We also show that SRRIP and DRRIP outperform LRU by an average of 7% and 9% on a 4-core CMP with a 16-way 8MB shared LLC. We also show that RRIP outperforms LFU, the state-of the art scan-resistant replacement algorithm to-date, by 2.5%. For the cache configurations under study, RRIP requires 2X less hardware than LRU and 2.5X less hardware than LFU.

In this study, we have applied re-reference interval prediction on cache misses and *learn* the re-reference interval of the missing block without any external information. Re-reference interval prediction on cache hits ideally requires knowledge of when a cache block receives its last hit. RIPP can use such information to update the re-reference prediction of the re-referenced cache block to *intermediate*, *long* or *distant* re-reference interval. Automatically learning the last reference on a cache hit is more challenging without any external information. Predicting re-reference interval on cache hits in the absence of external information or in the presence of dead block and last touch predictors [18, 16] is part of our on-going work.

8. ACKNOWLEDGEMENTS

The authors would like to thank Eric Borch, Malini Bhandaru, Paul Racunas, Krishna Rangan, Samantika Subramaniam and the anonymous reviewers for their feedback in improving this paper.

9. REFERENCES

[1] "Inside the Intel Itanium 2 Processor", HP Technical White Paper, July 2002.
[2] "UltraSPARC T2 supplement to the UltraSPARC architecture 2007", Draft D1.4.3. 2007.
[3] Intel. Intel Core i7 Processor. http://www.intel.com/products/processor/corei7/specifications.htm
[4] H. Al-Zoubi, A. Milenkovic, M. Milenkovic. "Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite." In ACMSE, 2004.
[5] S. Bansal and D. S. Modha. "CAR: Clock with Adaptive Replacement", In FAST, 2004.
[6] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, J. Martinez. "Scavenger: A New Last Level Cache Architecture with Global Block Priority". In Micro-40, 2007.
[7] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. In IBM Systems journal, pages 78–101, 1966.
[8] M. Chaudhuri. "Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches". In Micro, 2009.
[9] F. J. Corbat´o, "A paging experiment with the multics system," In Honor of P. M. Morse, pp. 217–228, MIT Press, 1969.
[10] A. Jaleel, R. Cohn, C. K. Luk, B. Jacob. CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In MoBS, 2008.
[11] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, S. C. Steely Jr., J. Emer. "Adaptive Insertion Policies for Managing Shared Caches". In PACT, 2008.
[12] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," In Proc. ACM SIGMETRICS Conf., 2002.
[13] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," In VLDB Conf., 1994.

[14] S. Kaxiras, Z. Hu, M. Martonosi. “Cache decay: exploiting generational behavior to reduce cache leakage power.” In ISCA-28.
[15] G. Keramidas, P. Petoumenos, S. Kaxiras. “Cache replacement based on reuse-distance prediction”. In ICCD, 2007
[16] A. Lai, C. Fide, B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In ISCA-28, 2001
[17] D. Lee, J. Choi, J. Kim, S. H. Noh, S. Lyul Min, Y. Cho, C. Sang Kim. “LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies,” IEEE Trans. Computers, vol. 50, no. 12, pp. 1352–1360, 2001.
[18] W. Lin and S. K. Reinhardt. “Predicting last-touch references under optimal replacement.” Technical Report CSE-TR-447-02, U. of Michigan, 2002.
[19] H. Liu, M. Ferdman, J. Huh, D. Burger. “Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency.” In Micro-41, 2008.
[20] G. Loh. “Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy”. In Micro, 2009.
[21] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In PLDI, pages 190–200, 2005.
[22] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” in FAST, 2003.
[23] E. J. O’Neil, P. E. O’Neil, G. Weikum. “The LRU-K page replacement algorithm for database disk buffering,” in Proc. ACM SIGMOD Conf., pp. 297–306, 1993.
[24] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, A. Karunanidhi. “Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation”. In MICRO-37, 2004.
[25] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, J. Emer. “Adaptive Insertion Policies for High Performance Caching”. In ISCA-34, 2007.
[26] K. Rajan and G. Ramaswamy. “Emulating Optimal Replacement with a Shepherd Cache”. In Micro-40, 2007.
[27] J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement,” in SIGMETRICS Conf, 1990.
[28] S. Srinath, O. Mutlu, H. Kim, Y. Patt. “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetcher”. In HPCA-13, 2007.
[29] R. Subramanian, Y. Smaragdakis, G. Loh. “Adaptive caches: Effective shaping of cache behavior to workloads.” In MICRO-39, 2006.
[30] Y. Xie and G. Loh. “PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches.” In ISCA-36, 2009
[31] Y. Zhou and J. F. Philbin, “The multi-queue replacement algorithm for second level buffer caches,” in USENIX Annual Tech. Conf, 2001.

10. Appendix

Cache replacement is not a problem for workloads that have a working set that fits in the available cache or for workloads that have a working set that is much larger than the available cache. Nonetheless we conducted a thorough study of SRRIP and DRRIP using a broader set of memory intensive and non-memory intensive workloads for the baseline single-core configuration. The study covers 28 SPEC CPU2006 workloads and 47 workloads from PC games, multimedia, server, and other categories. Table 4 compares the performance and hardware overhead of several replacement policies compared to LRU. Note that the average performance improvement compared to LRU across all workload categories is small (< 2%) because the study also includes workloads that do not benefit from cache replacement. We also compare against two additional hardware LRU approximations: PLRU [4] and CLOCK [5]. DIP provides competitive performance compared to SRRIP for SPEC workloads while SRRIP consistently outperforms LRU, DIP, PLRU, CLOCK, and NRU for PC games, multimedia, and server workloads. This shows the potential pitfalls of using SPEC workloads as a proxy for real world workloads. SRRIP outperforms LRU while requiring 2X less hardware. DRRIP outperforms scan-resistant replacement $HYB_{NRU/LFU}$ with 2.5X less hardware.

Table 4: Hardware Overhead and Performance of Replacement Policies

Replacement Policy	LRU	DIP_{LRU} [25]	PLRU [4]	CLOCK [5]	NRU [1]	DIP_{NRU} [25]	SRRIP	DRRIP	$HYB_{NRU/LFU}$
HW Overhead[a]	$n \cdot \log_2 n$	$n \cdot \log_2 n$	$n - 1$	$n + \log_2 n$	n	n	2n	2n	5n
HW for 16-way cache[a]	64	64	15	20	16	16	32	32	80
ALL Workloads[b]	1.0000	1.0081	0.9973	1.0024	0.9952	1.0061	1.0075	1.0172	1.0097
PC Games[b]	1.0000	0.9975	0.9982	1.0101	0.9869	1.0071	1.0294	1.0366	1.0343
Multimedia[b]	1.0000	0.9975	0.9966	1.0037	0.9950	0.9963	1.0287	1.0173	1.0243
Server[b]	1.0000	0.9985	0.9882	1.0069	0.9818	0.9896	1.0153	1.0138	0.9883
CPU2006 - FP[b]	1.0000	1.0325	1.0005	0.9989	0.9989	1.0278	0.9965	1.0320	1.0182
CPU2006 - INT[b]	1.0000	1.0140	0.9939	1.0022	0.9960	1.0141	1.0031	1.0232	1.0127
Other[b]	1.0000	0.9990	0.9990	1.0004	0.9986	0.9949	1.0013	0.9988	0.9953
Max Relative Perf	N/A	1.6589	1.0207	1.0360	1.0079	1.6158	1.1381	1.7112	1.4128
Min Relative Perf	N/A	0.9687	0.9706	0.9806	0.9507	0.9135	0.9853	0.9656	0.9394

a. Assuming an n-way set associative cache, hardware overhead is measured in number of bits required per cache set.
b. Performance is relative to LRU and is reported as geomean across 75 memory intensive and non-memory intensive workloads from server, multimedia, PC games, and the entire SPEC CPU2006 benchmark suite.