SHiP: Signature-based Hit Predictor for High Performance Caching

Carole-Jean Wu[*§] Aamer Jaleel[†] Will Hasenplaugh[†‡] Margaret Martonosi[*] Simon C. Steely Jr.[†] Joel Emer[†‡]

Princeton University[*]
Princeton, NJ
{carolewu,mrm}@princeton.edu

Intel Corporation, VSSAD[†]
Hudson, MA
{aamer.jaleel,william.c.hasenplaugh,
simon.c.steely.jr,joel.emer}@intel.com

Massachusetts Institute of Technology[‡]
Cambridge, MA

ABSTRACT

The shared last-level caches in CMPs play an important role in improving application performance and reducing off-chip memory bandwidth requirements. In order to use LLCs more efficiently, recent research has shown that changing the re-reference prediction on cache insertions and cache hits can significantly improve cache performance. A fundamental challenge, however, is how to best predict the re-reference pattern of an incoming cache line.

This paper shows that cache performance can be improved by correlating the re-reference behavior of a cache line with a unique signature. We investigate the use of memory region, program counter, and instruction sequence history based signatures. We also propose a novel Signature-based Hit Predictor (SHiP) to learn the re-reference behavior of cache lines belonging to each signature. Overall, we find that SHiP offers substantial improvements over the baseline LRU replacement and state-of-the-art replacement policy proposals. On average, SHiP improves sequential and multiprogrammed application performance by roughly 10% and 12% over LRU replacement, respectively. Compared to recent replacement policy proposals such as Seg-LRU and SDBP, SHiP nearly doubles the performance gains while requiring less hardware overhead.

Categories and Subject Descriptors

B.8.3 [**Hardware**]: Memory Structures

General Terms

Design, Performance

Keywords

Replacement, Reuse Distance Prediction, Shared Cache

1. Introduction

The widespread use of chip multiprocessors with shared last-level caches (LLCs) and the widening gap between processor and memory speeds increase the importance of a high performing LLC. Recent studies, however, have shown that the commonly-used LRU replacement policy still leaves significant room for performance improvement. As a result, a large body of research work has focused on improving LLC replacement [5, 10, 15, 16, 17, 20, 27, 29, 31, 32]. This paper focuses on improving cache performance by addressing the limitations of prior replacement policy proposals.

In the recently-proposed Re-Reference Interval Prediction (RRIP) framework [10], cache replacement policies base their replacement decisions on a *prediction* of the re-reference (or reuse) pattern of each cache line. Since the exact re-reference pattern is not known, the predicted re-reference behavior is categorized into different buckets known as *re-reference intervals*. For example, if a cache line is predicted to be re-referenced soon, it is said to have a *near-immediate* re-reference interval. On the other hand, if a cache line is predicted to be re-referenced far in the future, it is termed to have a *distant* re-reference interval. To maximize cache performance, cache replacement policies continually update the re-reference interval of a cache line. The natural opportunity to predict (and update) the re-reference interval is on cache insertions and cache hits.

The commonly-used LRU replacement policy (and its approximations) predict that *all* cache lines inserted into the cache will have a *near-immediate* re-reference interval. Recent studies [10, 27] have shown that always predicting a *near-immediate* re-reference interval on cache insertions performs poorly when application references have a *distant* re-reference interval. This situation occurs when the application working set is larger than the available cache or when the application has a mixed access pattern where some memory references have a *near-immediate* re-reference interval while others have a *distant* re-reference interval. For both of these access patterns, LRU replacement causes inefficient cache utilization. Furthermore, since LLCs only observe references filtered through the smaller caches in the hierarchy, the view of re-reference locality at the LLCs can be skewed by this filtering of upper-level caches.

In efforts to improve cache performance, many studies have proposed novel ideas to improve cache replacement [5, 15, 16, 17, 20, 31]. While these proposals address the limitations of LRU replacement, they either require additional hardware overhead or require significant changes to the cache structure. Alternatively, studies have also shown that simply changing the re-reference prediction on cache insertions [10, 27, 29, 32] can significantly improve cache performance at very low hardware overhead. However, a fundamental challenge today is how to best design a practical mechanism that can accurately predict the re-reference interval of a cache line on cache insertions.

Recent proposals [10, 27, 32] use simple mechanisms to predict the re-reference interval of the incoming cache line. Specifically, thes mechanisms predict the *same* re-reference interval for the ma-

[§]A large part of this work was performed while Carole-Jean Wu was an intern at Intel/VSSAD.

MICRO '11 December 3-7, 2011, Porto Alegre, Brazil

Table 1: Common Cache Access Patterns.

Recency-Friendly	$(a_1, ..., a_{k-1}, a_k, a_k, a_{k-1}, ..., a_1)^N$
Thrashing ($k > C$)	$(a_1, ..., a_{k-1}, a_k)^N$
Streaming ($k = \infty$)	$(a_1, ..., a_{k-1}, a_k)$
Mixed ($k < C, m > C$)	$[(a_1, ..., a_k)^A P_\epsilon(b_1, ..., b_m)]^N$

C represents the cache set associativity.
a_i represents a cache line access.
$(a_1, a_2, ..., a_{k-1}, a_k)$ is a temporal sequence of k unique addresses to a cache set.
$(a_1, a_2, ..., a_{k-1}, a_k)^N$ represents a temporal sequence that repeats N times.
$P_\epsilon(a_1, a_2, ..., a_{k-1}, a_k)$ is a temporal sequence that occurs with some probability P_ϵ.

jority of cache insertions and, as a result, make replacement decisions at a coarse granularity. While such proposals are simple and improve cache performance significantly, we show that there is opportunity to improve these predictions. Specifically, we show that re-reference predictions can be made at a finer granularity by categorizing references into different groups by associating a *signature* with each cache reference. The goal is that cache references that have the same signature will have a similar re-reference interval.

This paper investigates simple, high performing, and low overhead mechanisms to associate cache references with a unique signature *and* to predict the re-reference interval for that signature. We propose a Signature-based Hit Predictor (SHiP) to predict whether the incoming cache line will receive a future hit. We use three unique signatures to predict the re-reference interval pattern: memory region signatures (SHiP-Mem), program counter signatures (SHiP-PC), and instruction sequence history signatures (SHiP-ISeq). For a given signature, SHiP uses a Signature History Counter Table (SHCT) of saturating counters to *learn* the re-reference interval for that signature. SHiP updates the SHCT on cache hits and cache evictions. On a cache miss, SHiP indexes the SHCT with the corresponding signature to predict the re-reference interval of the incoming cache line.

SHiP is not limited to a specific replacement policy, but rather can be used in conjunction with any ordered replacement policy. SHiP is a more sophisticated cache insertion mechanism that makes more accurate re-reference predictions than recent cache insertion policy proposals [10, 27, 32]. Our detailed performance studies show that SHiP significantly improves cache performance over prior state-of-the art replacement policies. Of the three signatures, SHiP-PC and SHiP-ISeq perform the best across a diverse set of sequential applications including multimedia, games, server, and the SPEC CPU2006 applications. On average, with a 1 MB LLC, SHiP outperforms LRU replacement by 9.7% while existing state-of-the-art policies like DRRIP [10], Seg-LRU [5] and SDBP [16] improve performance by 5.5%, 5.6%, and 6.9% respectively. Our evaluations on a 4-core CMP with a 4 MB shared LLC also show that SHiP outperforms LRU replacement by 11.2% on average, compared to DRRIP (6.5%), Seg-LRU (4.1%), and SDBP (5.6%).

2. Background and Motivation

Increasing cache sizes and the use of shared LLCs in CMPs has spurred research work on improving cache replacement. Innovations in replacement policies include improvements in cache insertion and promotion policies [10, 27, 32], dead block prediction [15, 16, 18, 20], reuse distance prediction [10, 14, 17], frequency-based replacement [19] and many more [1, 2, 6, 11, 24, 28, 30, 31].

To better understand the need for more intelligent replacement policies, a recent study [10] summarized frequently occurring access patterns (shown in Table 1). LRU replacement (and its approximations) behaves well for both recency-friendly and streaming access patterns. However, LRU performs poorly for thrashing and mixed access patterns. Consequently, improving the performance for these access patterns has been the focus of many replacement policy proposals.

Thrashing occurs when the application working set is larger than the available cache. In such cases, preserving part of the working set in the cache can significantly improve performance. Recent proposals show that simply changing the re-reference predictions on cache insertions can achieve this desired effect [10, 27].

Mixed access patterns on the other hand occur when a frequently referenced working set is continuously discarded from the cache due to a burst of non-temporal data references (called *scans*). Since these access patterns are commonly found in important multimedia, games, and server applications [10], improving the cache performance of such workloads is of utmost importance.

To address mixed access patterns, a recent study proposed the Dynamic Re-Reference Interval Prediction (DRRIP) replacement policy [10]. DRRIP consists of two component policies: Bimodal RRIP (BRRIP) and Static RRIP (SRRIP). DRRIP uses Set Dueling [27] to select between the two policies. BRRIP is specifically targeted to improve the performance of thrashing access patterns. SRRIP is specifically targeted to improve the performance of mixed access patterns. DRRIP performance is shaped by how well it performs on the two component policies.

The performance of the SRRIP component policy is dependent on two important factors. First, SRRIP relies on the active working set to be re-referenced at least once. Second, SRRIP performance is constrained by the scan length. Table 2 illustrates scan access patterns using the scan notation from Table 1, and their performance under SRRIP. For short scan lengths, SRRIP performs well. However, when the scan length exceeds the SRRIP threshold or when the active working set has not been re-referenced before the scan, SRRIP behaves similarly to LRU replacement. We focus on designing a low overhead replacement policy that can improve the performance of mixed access patterns.

3. Signature-based Hit Predictor

Most replacement policies attempt to learn the re-reference interval of cache lines by always making the *same* re-reference prediction for all cache insertions. Instead of making the *same* re-reference predictions for all cache insertions, we associate each cache reference with a distinct *signature*. We show that cache replacement policies can be significantly improved by dynamically learning the re-reference interval of each signature and applying the information learned at cache insertion time.

3.1 Signature-based Replacement

The goal of signature-based cache replacement is to predict whether the insertions by a given signature will receive future cache hits. The intuition is that if cache insertions by a given signature

Table 2: n-bit SRRIP Behavior with Different Mixed Access Patterns

Mixed Access Patterns	$[(a_1, a_2, ..., a_{k-1}, a_k)^A P_\epsilon(b_1, b_2, ..., b_m)]^N$	Example
Short Scan	$m <= (C - K) * (2^n - 1)$ and $A > 1$	$(a_1, a_2), (a_1, a_2), b_1, b_2, b_3, (a_1, a_2), ...$
Long Scan	$m > (C - K) * (2^n - 1)$ and $A > 1$	$(a_1, a_2), (a_1, a_2), b_1, b_2, b_3, b_4, b_5, b_6, (a_1, a_2), ...$
Exactly One Reuse	$A = 1$ regardless of m	$(a_1, a_2), b_1, b_2, b_3, b_4, (a_1, a_2), ...$

m: the length of a scan; C: the cache set associativity; K: the length of the active working set.

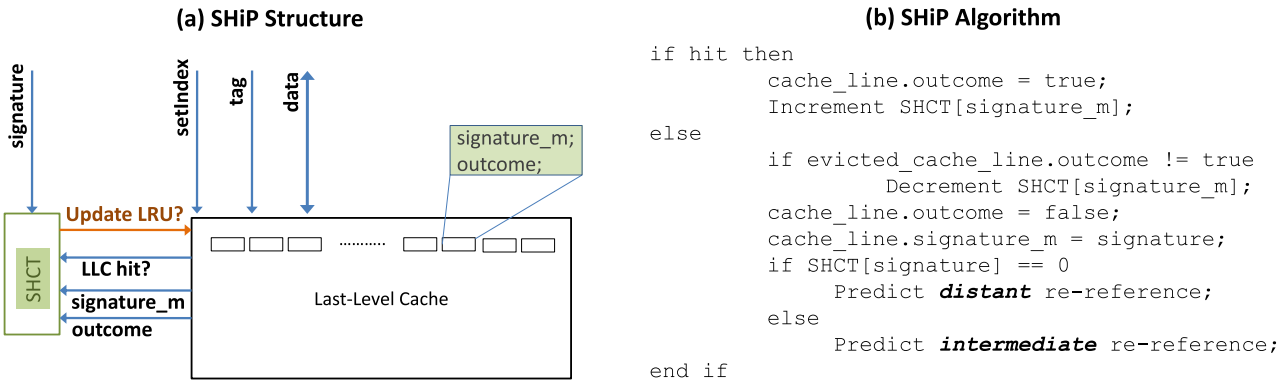

Figure 1: (a) SHiP Structure and (b) SHiP Algorithm.

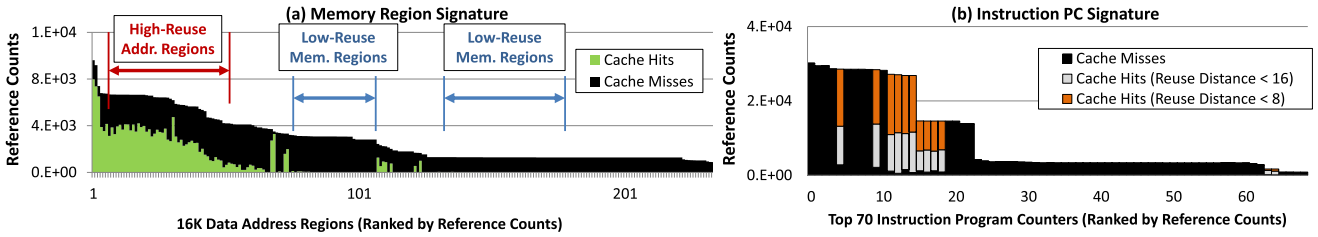

Figure 2: (a) Memory region signature: 393 unique 16KB address regions are referenced in the entire `hmmer` program run. The most-referenced regions show high data reuse whereas the other regions include mostly cache misses. (b) Instruction PC signature: reference counts per PC for `zeusmp`

are re-referenced, then future cache insertions by the same signature will again be re-referenced. Conversely, if cache insertions by a given signature do not receive subsequent hits, then future insertions by the same signature will again not receive any subsequent hits. To explicitly correlate the re-reference behavior of a signature, we propose a *Signature-based Hit Predictor (SHiP)*.

To learn the re-reference pattern of a signature, SHiP requires two additional fields to be stored with each cache line: the signature itself and a single bit to track the *outcome* of the cache insertion. The *outcome* bit (initially set to zero) is set to one only if the cache line is re-referenced. Like global history indexed branch predictors [33], we propose a *Signature History Counter Table (SHCT)* of saturating counters to learn the re-reference behavior of a signature. When a cache line receives a hit, SHiP increments the SHCT entry indexed by the signature stored with the cache line. When a line is evicted from the cache but has not been re-referenced since insertion, SHiP decrements the SHCT entry indexed by the signature stored with the evicted cache line.

The SHCT value indicates the re-reference behavior of a signature. A zero SHCT value provides a strong indication that future lines brought into the LLC by that signature will not receive any cache hits. In other words, references associated with this signature always have a *distant* re-reference interval. On the other hand, a positive SHCT counter implies that the corresponding signature receives cache hits. The exact re-reference interval is unknown because the SHCT only tracks whether or not a given signature is re-referenced, but not its timing.

Figure 1 illustrates the SHiP structure and SHiP pseudo-code. The hardware overhead of SHiP includes the SHCT and the two additional fields in each cache line: signature and outcome. SHiP requires no changes to the cache promotion or victim selection policies. To avoid the per-line overhead, Section 7.1 illustrates the use of set sampling [27] to limit the hardware overhead to only a few cache lines for SHCT training.

SHiP can be used in conjunction with any ordered replacement policy. The primary goal of SHiP is to predict the re-reference interval of an incoming cache line. For example, on a cache miss, the signature of the missing cache line is used to consult the SHCT. If the corresponding SHCT entry is zero, SHiP predicts that the incoming line will have a *distant* re-reference interval, otherwise SHiP predicts that the incoming line will have an *intermediate* re-reference interval. Given the re-reference prediction, the replacement policy can decide how to apply it. For example, LRU replacement can apply the prediction of *distant* re-reference interval by inserting the incoming line at the end of the LRU chain (instead of the beginning). Note that SHiP makes re-reference predictions only on cache insertions. Extensions of SHiP to update re-reference predictions on cache hits are left for future work.

For our studies, we evaluate SHiP using the SRRIP replacement policy. We use SRRIP because it requires less hardware than LRU and in fact outperforms LRU [10]. In the absence of any external information, SRRIP conservatively predicts that all cache insertions have an *intermediate* re-reference interval. If the newly-inserted cache line is referenced quickly, SRRIP updates the re-reference prediction to *near-immediate*; otherwise, SRRIP downgrades the re-reference prediction to *distant*. In doing so, SRRIP learns the re-reference interval for all inserted cache lines upon re-reference. SHiP on the other hand explicitly and dynamically predicts the re-reference interval based on the SHCT. SHiP makes no changes to the SRRIP victim selection and hit update policies. On a cache miss, SHiP consults the SHCT with the signature to predict the re-reference interval of the incoming cache line. Table 3 summarizes the cache insertion and hit promotion policies for the 2-bit SRRIP and 2-bit SHiP schemes.

Table 3: Policies for the 2-bit SRRIP and SHiP.

	SRRIP	SHiP
Insertion	always 2	if (SHCT[Signature] == 0) 3; else 2;
Promotion	always 0	always 0

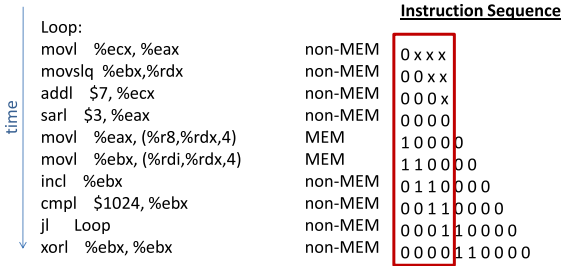

Figure 3: Example of Encoding Instruction Sequence History

3.2 Signatures for Improving Replacement

While there are many ways of choosing a signature for each cache reference, we evalaute SHiP using the following three signatures.

- **Memory Region Signature**: Cache references can have signatures based on the memory region being referenced. Specifically, the most significant bits of the data address can be hashed to form a signature. Figure 2(a) illustrates the reuse characteristics for `hmmer`. The x-axis shows a total of 393 unique 16KB memory regions referenced in the entire program (ranked by reference counts) while the y-axis shows the number of references in each region. Data references to certain address regions have "low-reuse" and always result in cache misses. On the other hand, references to other address regions are reused more often. A memory-region-based signature can generate accurate re-reference predictions if all references to a given memory region have a typical access pattern (e.g. scans).
- **Program Counter (PC) Signature**: Cache references can be grouped based on the instructions which reference memory. Specifically, bits from the instruction Program Counter (PC) can be hashed to form a signature. Figure 2(b) illustrates the reference counts per instruction PC for a SPEC CPU2006 application, `zeusmp`. The x-axis shows the 70 instructions that most frequently access memory (covering 98% of all LLC accesses), while the y-axis shows the reference counts. The bars show, under LRU replacement, whether these memory references receive cache hits or misses. Intuitively, SHiP can identify the frequently-missing instructions (i.e. instructions 1-4) and predict that all memory references by these instructions have a *distant* re-reference interval. A PC-based signature can generate accurate re-reference predictions if most references from a given PC have similar reuse behavior.
- **Instruction Sequence History Signature**: Cache references can also be grouped using an instruction sequence history for that memory reference. We define instruction sequence history as a binary string that corresponds to the sequence of instructions decoded before the memory instruction. If a decoded instruction is a load/store instruction, a '1' is inserted into the sequence history, else a '0' is inserted into the sequence history. Figure 3 illustrates this using an example. Instruction sequences can capture correlations between references and may be more compact than PC-based signatures.

4. Experimental Methodology

4.1 Simulation Infrastructure

We evaluate SHiP using the the simulation framework released by the First JILP Workshop on Computer Architecture Competitions [12]. This Pin-based [22] CMP$im [8] simulation framework models a 4-way out-of-order processor with a 128-entry reorder buffer and a three-level cache hierarchy. The three-level cache hierarchy is based on an Intel Core i7 system [7]. The L1 and L2 caches use LRU replacement and our replacement policy studies are limited to the LLC. Table 4 summarizes the memory hierarchy.

For the SHiP scheme, we use a default 16K-entry SHCT with 3-bit saturating counters[1] SHiP-PC uses the 14-bit hashed instruction PC as the signature indexing to the SHCT. SHiP-ISeq uses the 14-bit hashed memory instruction sequence as the signature. The instruction sequence is constructed at the decode stage of the pipeline. Like all prior PC-based schemes, the signature is stored in the load-store queue and accompanies the memory reference throughout all levels of the cache hierarchy. SHiP-Mem uses the upper 14-bit of data addresses as the signature. Table 3 summarizes re-reference predictions made by SHiP upon cache insertion.

4.2 Workload Construction

Our evaluations use both sequential and multiprogrammed workloads. We use 24 memory-sensitive applications from multimedia and PC games (Mm.), enterprise server (Srvr.), and the SPEC

[1]Each SHCT entry is a measure of confidence. When the entry is zero, it gives high confidence that the corresponding references will not be re-referenced.

Table 4: Architectural parameters of the simulated system.

L1 Inst. Caches	32KB, 4-way, Private, 1 cycle	MSHR	32 entries allowing up to 32 outstanding misses
L1 Data Caches	32KB, 8-way, Private, 1 cycle	LLC	1MB per-core, 16-way, Shared, 30 cycles
L2 Caches	256KB, 8-way, Private, 10 cycles	Main Memory	32 outstanding requests, 200 cycles

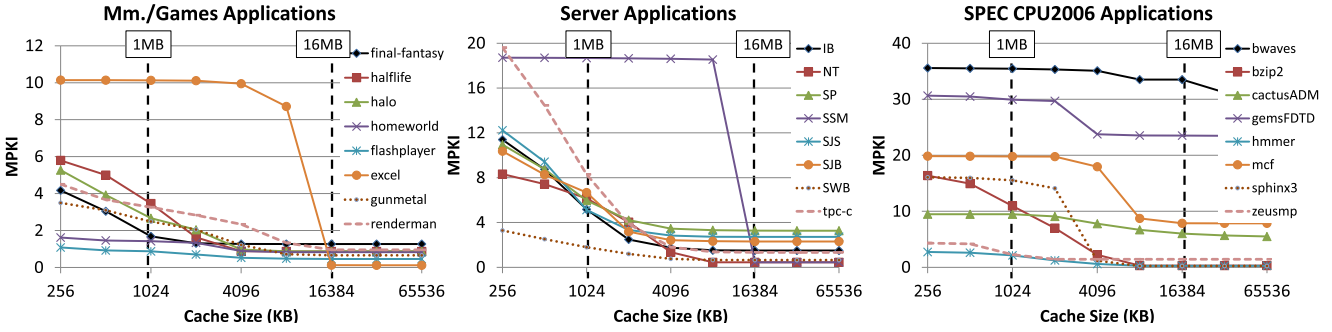

Figure 4: Cache Sensitivity of the Selected Applications.

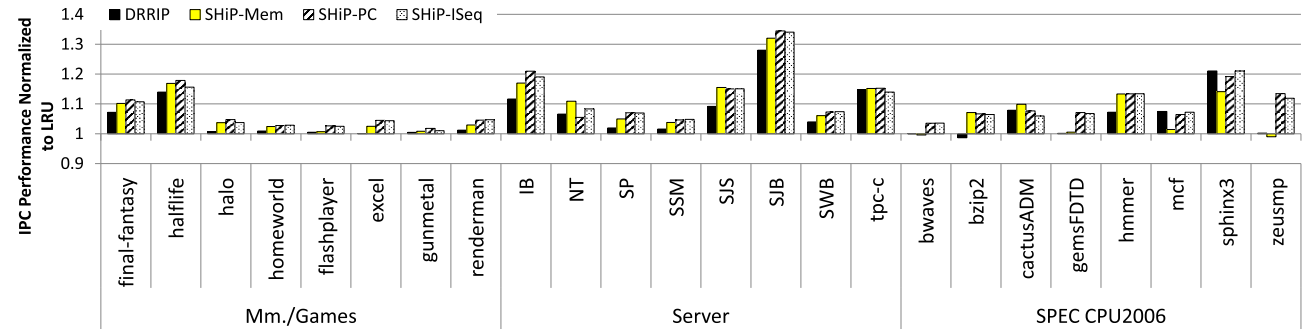

Figure 5: Performance comparison of DRRIP, SHiP-PC, SHiP-ISeq, and SHiP-Mem.

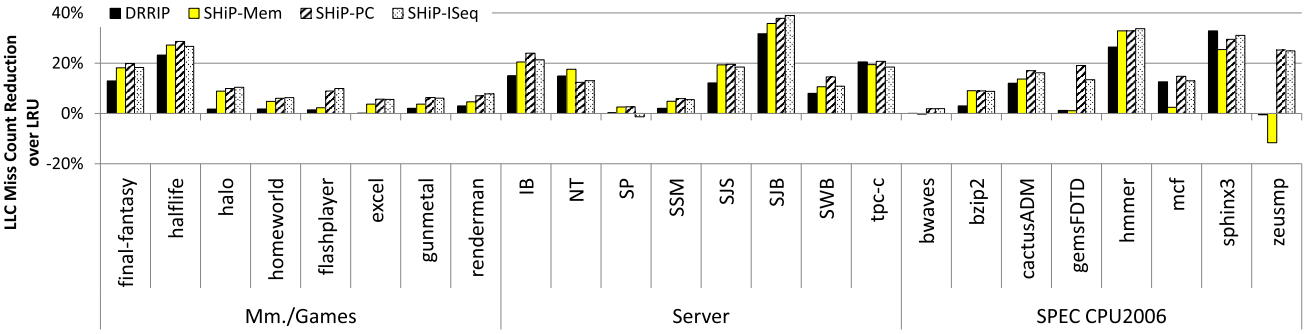

Figure 6: SHiP miss count reduction over LRU.

CPU2006 categories. From each category, we select eight benchmarks for which the IPC performance doubles when the cache size increases from 1MB to 16MB. Figure 4 shows the cache sensitivity of the selected applications. The SPEC CPU2006 workloads were collected using PinPoints [25] for the reference input set while the other workloads were collected on a hardware tracing platform. These workloads were run for 250 million instructions.

To evaluate shared LLC performance, we construct 161 heterogeneous mixes of multiprogrammed workloads. We use 35 heterogeneous mixes of multimedia and PC games, 35 heterogeneous mixes of enterprise server workloads, and 35 heterogeneous mixes of SPEC CPU2006 workloads. Finally, we create another 56 random combinations of 4-core workloads. The heterogeneous mixes of different workload categories are used as a proxy for a virtualized system. We run each application for 250 million instructions and collect the statistics with the first 250 million instructions completed. If the end of the trace is reached, the model rewinds the trace and restarts automatically. This simulation methodology is similar to recent work on shared caches [4, 9, 10, 21, 32].

5. Evaluation for Private Caches

Figures 5 and 6 compare the throughput improvement and cache miss reduction of the 24 selected applications. For applications that already receive performance benefits from DRRIP such as `final-fantasy`, `IB`, `SJS`, and `hmmer`, all SHiP schemes further improve performance. For these applications, performance gains primarily come from SHiP's ability to accurately predict the re-reference interval for incoming lines.

More significantly, consider applications, such as `halo`, `excel`, `gemsFDTD`, and `zeusmp`, where DRRIP provides no performance improvements over LRU. Here, SHiP-PC and SHiP-ISeq can provide performance gains ranging from 5% to 13%. The performance gains is due to the 10% to 20% reduction in cache misses. The results show that SHiP-PC and SHiP-ISeq both dynamically correlate the current program context to an expected re-reference interval.

In particular, SHiP-PC and SHiP-ISeq are especially effective in managing a commonly found access pattern in applications such as `halo`, `excel`, `gemsFDTD`, and `zeusmp`. Figure 7 captures a stream of memory references to a particular cache set in `gemsFDTD`. For this particular cache set, addresses `A`, `B`, `C`, and `D` are brought into the cache by instruction `P1`. Before getting re-referenced again, `A`, `B`, `C`, and `D` are evicted from the cache under both LRU and DRRIP because the number of distinct interleaving references exceeds the cache associativity. As a result, the subsequent re-references to `A`, `B`, `C`, and `D` by a different instruction `P2` result in cache misses.

However, under SHiP-PC, the SHCT learns the re-reference interval of references associated to instruction `P1` as *intermediate* and the re-reference interval of other references as *distant*. As a result, when the initial references `A`, `B`, `C`, and `D` are inserted to the LLC at `P1`, SHiP predicts these to have the intermediate re-reference interval while other interleaving references have the distant re-reference interval. This example illustrates how SHiP-PC and SHiP-ISeq can effectively identify the reuse pattern of data brought into the LLC by multiple signatures.

Regardless of the signature, SHiP outperforms both DRRIP and LRU. Though SHiP-Mem provides gains, using program context information such as instruction PC or memory instruction sequence provides more substantial performance gains. Among the three, SHiP-PC and SHiP-ISeq perform better than SHiP-Mem. On average, compared to LRU, SHiP-Mem, SHiP-PC and SHiP-ISeq improve throughput by 7.7%, 9.7% and 9.4% respectively while DRRIP improves throughput by 5.5%. Unless mentioned, the remainder of the paper primarily focuses on SHiP-PC and SHiP-ISeq.

5.1 Coverage and Accuracy

To evaluate how well SHiP identifies the likelihood of reuse, this section analyzes SHiP-PC coverage and prediction accuracy. Table 5 outlines the five different outcomes for all cache references under SHiP. Figure 8 shows results for SHiP-PC. On average, only 22% of data references are predicted to receive further cache hit(s) and are

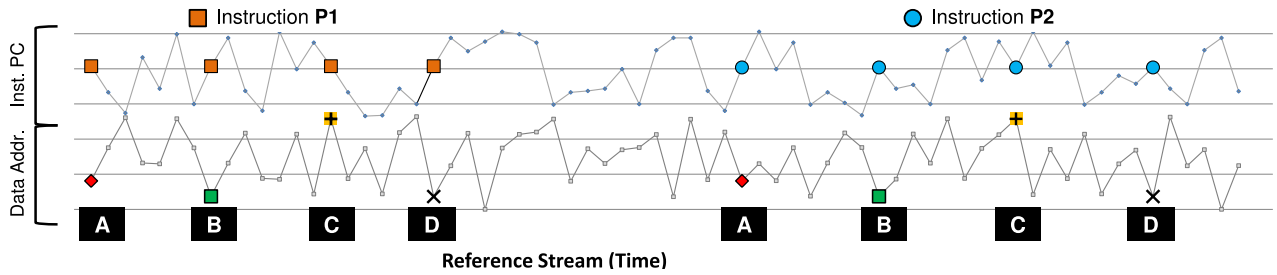

Figure 7: Cross-instruction reuse pattern: SHiP-PC can learn and predict that references A, B, C, and D brought in by instruction *P1* receive cache hits under a different instruction *P2*. SHiP learns the data references brought in by *P1* are reused by *P2* and, therefore, assigns an intermediate re-reference interval to A, B, C, and D. As a result, the second occurrences of A, B, C, and D by *P2* hit in the cache whereas, under either LRU- or DRRIP-based cache, A, B, C, and D all miss in the cache.

inserted to the LLC with the *intermediate* re-reference prediction. The rest of the data references are inserted to the LLC with the *distant* re-reference prediction.

The accuracy of SHiP's prediction can be evaluated by comparing against the actual access pattern. For cache lines filled with the distant re-reference (DR) interval, SHiP-PC is considered to make a misprediction if a DR-filled cache line receives further cache hit(s) during its cache lifetime. Furthermore, a DR-filled cache line under SHiP-PC could have received cache reuse(s) if it were filled with the intermediate re-reference interval. To fairly account for SHiP-PC's misprediction for DR-filled cache lines, we implement an 8-way first-in-first-out (FIFO) victim buffer per cache set[2]. For cache lines that are filled with the *distant* re-reference prediction, Figure 8 shows that SHiP-PC achieves 98% prediction accuracy. Few DR-filled cache lines see further cache reuse after insertion. Consider applications like `gemsFDTD` and `zeusmp` for which SHiP-PC offers large performance improvements. Figure 8 shows that SHiP-PC achieves more than 80% accuracy for cache lines predicted to receive further cache hit(s).

On average, for the cache lines filled with the *intermediate* re-reference (IR) prediction, SHiP-PC achieves 39% prediction accuracy. SHiP-PC's predictions for cache lines that will receive no further cache reuse(s) are conservative because the misprediction penalty for DR-filled cache lines is performance degradation, while the misprediction penalty for IR-filled cache lines is just the cost of missing possible performance enhancement. For the mispredicted IR-filled cache lines, SHiP learns that these cache lines have received no cache hit(s) at their eviction time and adjusts its re-reference prediction accordingly. Consequently, SHiP-PC trains its re-reference predictions gradually and more accurately with every evicted cache line.

Over all the evicted cache lines, SHiP-PC doubles the application hit counts over the DRRIP scheme. Figure 9 illustrates the percentage of cache lines that receive at least one cache hit during their cache lifetime. For applications such as `final-fantasy`, `SJB`, `gemsFDTD`, and `zeusmp`, SHiP-PC improves the total hit counts significantly. This is because SHiP-PC accurately predicts and retains cache lines that will receive cache hit(s) in the LLC. Consequently, the overall cache utilization increases under SHiP-PC. A similar analysis with SHiP-ISeq showed similar behavior.

[2]A victim buffer is used for evaluating SHiP prediction accuracy. It is not implemented in the real SHiP design. The per-set FIFO victim buffer stores DR-filled cache lines that have not received a cache hit before eviction.

5.2 Sensitivity of SHiP to SHCT Size

The SHCT size in SHiP should be small enough to ensure a practical design and yet large enough to mitigate aliasing between un-

Table 5: Definition of SHiP accuracy.

Re-Reference Interval Prediction	Outcome	Definition
Distant Re-Reference (DR)	accurate	DR cache lines receive no cache hit(s) before eviction.
	inaccurate	DR cache lines receive cache hit(s) before eviction.
	inaccurate	DR cache lines receive cache hit(s) in the victim buffer.
Intermediate Re-Reference (IR)	accurate	IR cache lines receive cache hit(s) before eviction.
	inaccurate	IR cache lines receive no cache hit(s) before eviction.

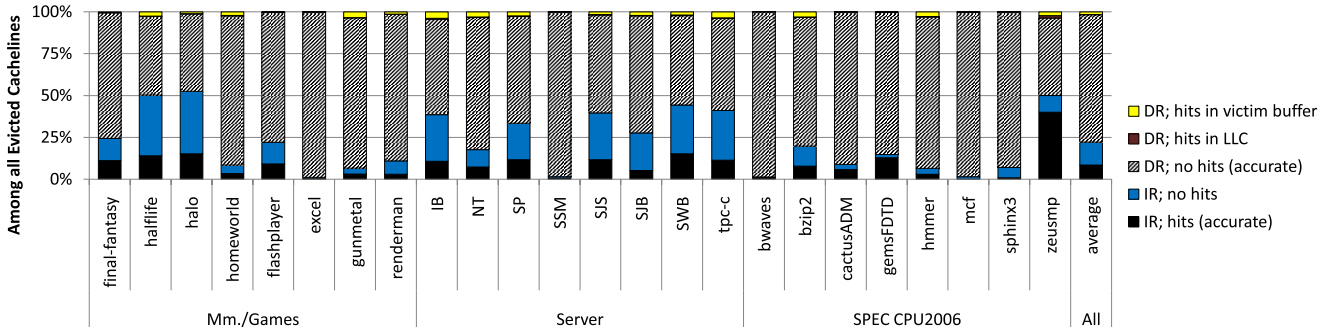

Figure 8: SHiP-PC coverage and accuracy.

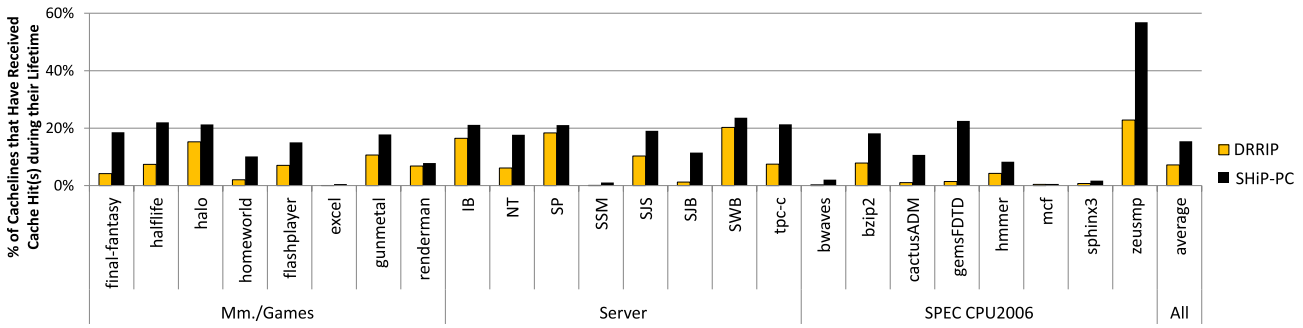

Figure 9: Comparison of application hit counts under DRRIP and SHiP-PC.

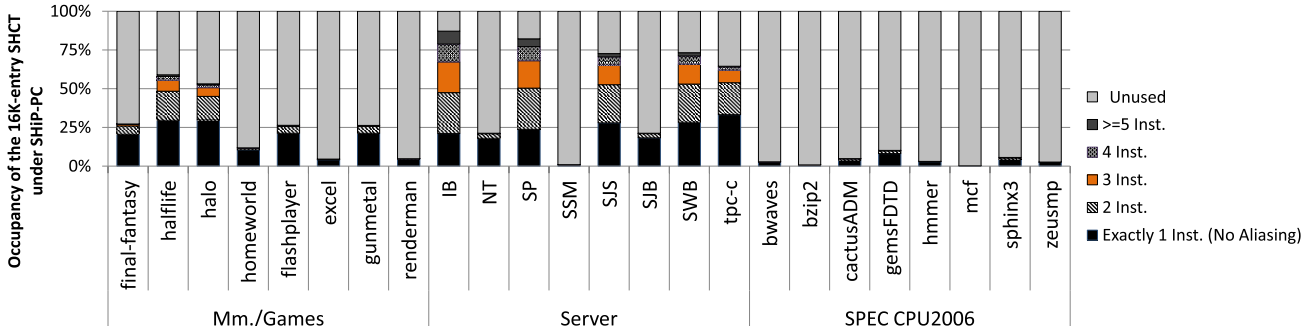

Figure 10: SHCT utilization and aliasing under the SHiP-PC scheme.

related signatures mapping to the same SHCT entry. Since the degree of aliasing depends heavily on the signature used to index the SHCT, we conduct a design tradeoff analysis between performance and hardware cost for SHiP-PC and SHiP-ISeq separately.

Figure 10 plots the number of instructions sharing the same SHCT entry for a 16K-entry SHiP-PC. In general, the SHCT utilization is much lower for the multimedia, games, and SPEC CPU2006 applications than the server applications. Since the instruction working set of these applications is small, there is little aliasing in the 16K-entry SHCT. On the other hand, server applications with larger instruction footprints have higher SHCT utilizations.

For SHiP-PC, we varied the SHCT size from 1K to 1M entries. Very small SHCT sizes, such as 1K-entries, reduced SHiP-PC's effectiveness by roughly 5-10%, although it always outperforms LRU even with such small table sizes. Increasing the SCHT beyond 16K entries provides marginal performance improvement. This is because the instruction footprints of all our workloads fit well into the 16K-entry SCHT. Therefore, we recommend an SCHT of 16K entries or smaller.

Given the same 16K-entry SHCT size, SHiP-ISeq exhibits a different utilization pattern because it uses the memory instruction sequence signature to index to the SHCT. For all applications, less than half of the 16K-entry SHCT is utilized. This is because certain memory instruction sequences usually do not occur in an application. Hence, this provides an opportunity to reduce the SHCT of SHiP-ISeq.

Instead of using the 14-bit hashed memory instruction sequence to index to the SHCT directly, we further compress the signature to 13 bits and use the compressed 13-bit signature to index an 8K-entry SHCT. We call this variation SHiP-ISeq-H. Figure 11(a) shows that the utilization of the 8K-entry SHCT is increased significantly. While the degree of memory instruction sequence aliasing increases as well, in particular for server workloads, this does not affect the performance as compared to SHiP-ISeq. Figure 11(b) compares the performance improvement for the selected sequential applications under DRRIP, SHiP-PC, SHiP-ISeq, and SHiP-ISeq-H over LRU. While SHiP-ISeq-H uses the 8K-entry SHCT (half of the default 16K-entry SHCT), it offers a comparable performance gain as SHiP-PC and SHiP-ISeq and improves performance by an average of 9.2% over LRU.

6. Evaluation for Shared Caches

6.1 Results for SHiP-PC and SHiP-ISeq

For shared caches, on average, DRRIP improves the performance of the 161 multiprogrammed workloads by 6.4%, while SHiP-PC and SHiP-ISeq improve the performance more significantly by 11.2% and 11.0% respectively. For an in-depth analysis, we randomly selected 32 multiprogrammed mixes of sequential applications representative of the behavior of all 161 4-core workloads[3]. With the default 64K-entry SHCT scaled for the shared LLC, Figure 12 shows that the performance improvement of the selected workloads under SHiP-PC and SHiP-ISeq is 12.1% and 11.6% over LRU respectively while it is 6.7% under DRRIP. Unlike DRRIP, SHiP-PC and SHiP-ISeq performance improvements are primarily due to fine-grained re-reference interval predictions.

Section 5.2 showed that aliasing within a sequential application is mostly constructive and does not degrade performance. However, for the multiprogrammed workloads, aliasing in the SHCT becomes more complicated. In a shared SHCT, aliasing not only comes from multiple signatures within an application but it also stems from signatures from different co-scheduled applications. Constructive aliasing helps SHiP to learn the correct data reuse patterns quickly because the multiple aliasing signatures from the different applications adjust the corresponding SHCT entry unanimously. Consequently, SHiP's learning overhead is reduced. On the other hand, destructive aliasing can also occur when the aliasing signa-

[3]The performance improvement of the randomly selected workloads is within 1.2% difference compared to the performance improvement over all 161 multi-programmed workloads (Figure 12).

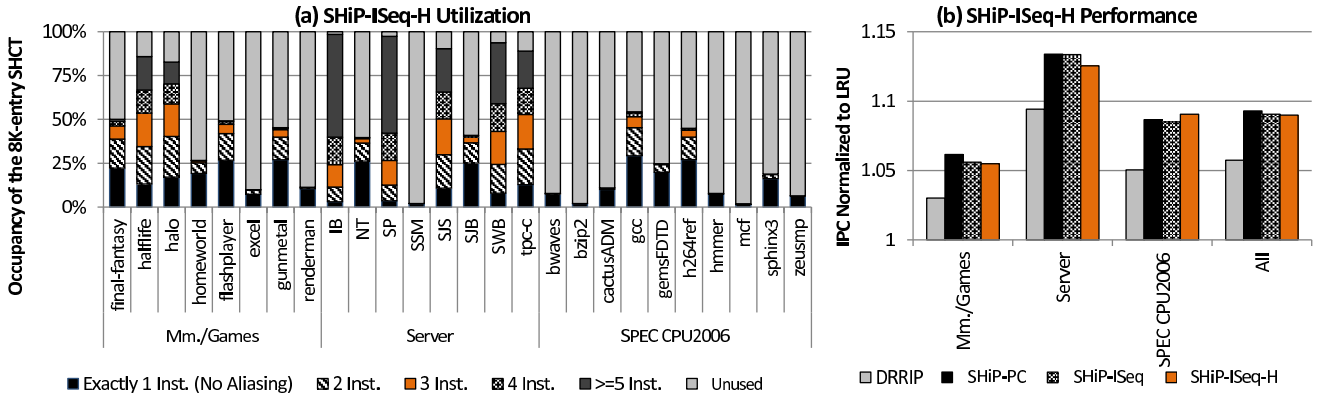

Figure 11: SHiP-ISeq-H (a) utilization and aliasing in SHCT (b) performance comparison with SHiP-ISeq.

tures adjust the same SHCT entry in opposite directions. This can affect SHiP accuracy and reduce its performance benefits.

To investigate the degree of constructive versus destructive aliasing among the different co-scheduled applications, we evaluate three different SHCT implementations: the unscaled default 16K-entry SHCT, the scaled 64K-entry SHCT, and the per-core private 16K-entry SHCT for each of the four CMP cores. The former two designs propose a monolithic shared SHCT for all four co-scheduled applications while in the latter design, each core has its own private SHCT to completely eliminate cross-core aliasing.

6.2 Per-core Private vs. Shared SHCT

Figure 13 illustrates the sharing patterns among all co-scheduled applications in the shared 16K-entry SHCT under SHiP-PC. The *No Sharer* bars plot the portion of the SHCT used by exactly one application. The *More than 1 Sharer (Agree)* bars plot the portion of the SHCT used by more than one application but the prediction results among the sharers agree. The *More than 1 Sharer (Disagree)* bars plot the portion of the SHCT suffering from destructive aliasing. Finally, the *Unused* bars plot the unused portion of the SHCT. The degree of destructive aliasing is fairly low across all workloads: 18.5% for Mm./Games mixes, 16% for server mixes, only 2% for SPEC CPU2006 mixes, and 9% for general multiprogrammed workloads.

Figure 14 compares the performance improvement for the three SHCT implementations in the SHiP-PC and SHiP-ISeq schemes. Although destructive aliasing does not occur frequently in the shared 16K-entry SHCT, multimedia, games, and server workloads still favor the per-core 16K-entry SHCT over the other shared SHCT designs. This is because, as shown in the private cache performance analysis, multimedia, games, and server applications have relatively larger instruction footprints. When the number of concurrent applications increases, the SHCT utilization increases as well and aliasing worsens. This problem can be alleviated by scaling the shared SHCT from 16K-entry to 64K-entry. However, the per-core private SHCT is the most effective solution.

Unlike the multimedia, games, and server workloads, the multiprogrammed SPEC CPU2006 application mixes receive the most performance improvement from the shared SHCT designs. As discussed in Section 5.2, a significant portion of the 16K-entry SHCT is unused for any SPEC CPU2006 application alone. When more SPEC CPU2006 applications are co-scheduled, the utilization of the shared SHCT increases but the shared 16K-entry SHCT is still sufficient. While the per-core private 16K-entry SHCT eliminates destructive aliasing completely, it improves the performance for the SPEC CPU2006 workloads less because of the learning overhead each private SHCT has to pay to warm up the table.

Overall, for the shared LLC, the two alternative SHCT designs, the shared 16K-entry SHCT and the per-core private 16K-entry SHCT, perform comparably to the shared 64K-entry SHCT in the SHiP-PC and SHiP-ISeq schemes.

7. SHiP Optimization and Comparison with Prior Work

While the proposed SHiP scheme offers excellent performance gains, to realize a practical SHiP design, we present two techniques to reduce SHiP hardware overhead: SHiP-S and SHiP-R. Instead of using all cache sets in the LLC, SHiP-S selects a few cache set samples to train the SHCT. Then, in the SHiP-R scheme, we explore the optimal width of the saturating counters in the SHCT. Finally, we compare the performance of all SHiP variants with the three state-of-the-art cache replacement policies: DRRIP, Segmented LRU (Seg-LRU), and Sampling Dead Block Prediction (SDBP) and discuss the design tradeoff for each of the schemes.

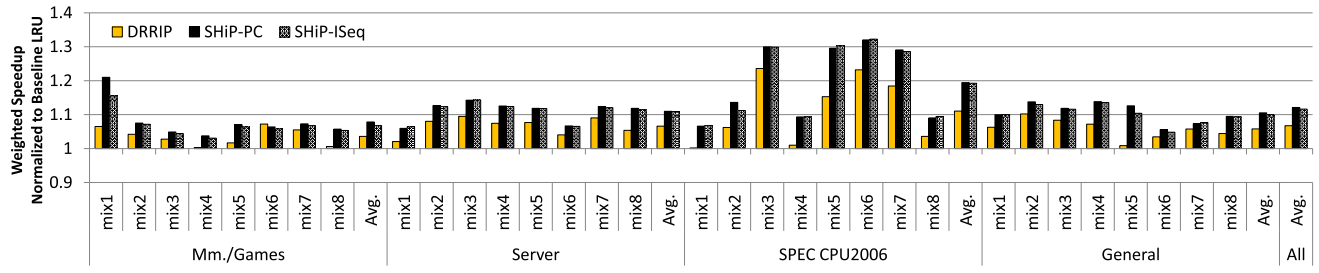

Figure 12: Performance comparison for multiprogrammed workloads under DRRIP, SHiP-PC, and SHiP-ISeq.

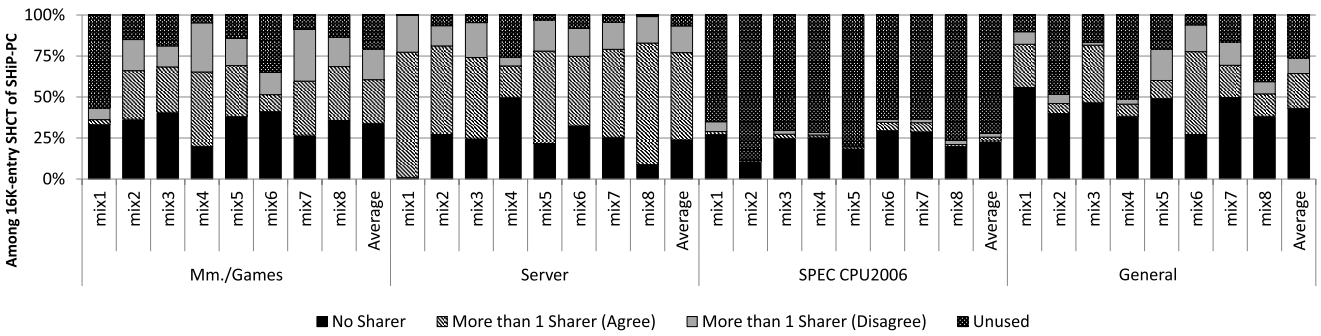

Figure 13: Utilization and aliasing for the shared 16K-entry SHCT under SHiP-PC.

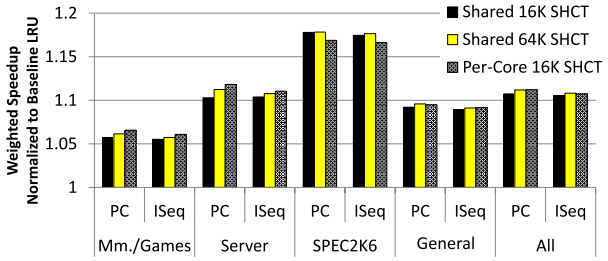

Figure 14: Performance comparison for per-core private vs. shared SHCT.

7.1 SHiP-S: Reducing Per-line Storage

Using every cache line's reuse outcome for SHiP training requires each cache line to store two additional information, 14-bit `signature_m` and 1-bit `outcome`, for the SHCT to learn the reuse pattern of the signature. We propose to use set sampling to reduce the per-cache line storage overhead of the default SHiP-PC and SHiP-ISeq schemes. SHiP-PC-S and SHiP-ISeq-S select a number of cache sets randomly and use cache lines in the selected cache sets to train the SHCT.

For the private 1MB LLC, using 64 out of the total 1024 cache sets is sufficient for SHiP-PC-S to retain most of the performance gain from the default SHiP-PC scheme. This reduces SHiP-PC's total storage in the LLC from 30KB to only 1.875KB. For the shared 4MB LLC, more sampling cache sets are required for SHCT training. Overall, 256 out of the total 4096 cache sets offers a good design point between performance benefit and hardware cost. Figure 15 shows that with set sampling, SHiP-PC and SHiP-ISeq retain most of the performance gains while the total per-line storage overhead is reduced to less than 2% of the entire cache capacity.

7.2 SHiP-R: Reducing the Width of Saturating Counters in the SHCT

We can further reduce SHiP hardware overhead by decreasing the width of SHCT counters. In the default SHiP scheme, SHCT uses 3-bit saturating counters. Using wider counters requires more hardware but ensures higher prediction accuracy for SHiP because only re-references with a strongly-biased signature are predicted to have the *distant* re-reference interval. On the other hand, using narrower counters not only requires less hardware but it also accelerates the learning time of the signatures.

Figure 15 compares the performance of the default SHiP-PC and SHiP-PC-R2 based on 2-bit saturating counters in the SHCT. For the private LLC (Figure 15(a)), the default SHiP-PC and the SHiP-PC-R2 schemes perform similarly while SHiP-PC-R2 uses 33% less hardware for the SHCT. We see a similar trend for the default SHiP-ISeq and SHiP-ISeq-R2.

For the shared LLC, using the 2-bit saturating counters in the SHCT accelerates SHiP's learning of signature reuse patterns. As a result, SHiP-PC-R2 and SHiP-ISeq-R2 both perform better than the default SHiP-PC and default SHiP-ISeq schemes.

7.3 Comparison with Prior Work

In addition to DRRIP, we compare the proposed SHiP scheme with two recently proposed cache management techniques, Segmented-LRU (Seg-LRU) [5] and Sampling Dead Block Prediction (SDBP) [16]. DRRIP, Seg-LRU, and SDBP are the top three best-performing cache replacement policies from JILP Cache Replacement Championship Workshop. Among the three, SDBP also uses additional information, such as instruction PCs, to assist its LLC management.

Figure 16 compares the performance improvement for sequential applications under DRRIP, Seg-LRU, SDBP, and our proposed SHiP schemes. For applications such as `SJS`, the additional instruction level information in SDBP, SHiP-PC, and SHiP-ISeq helps improve LLC performance significantly over the LRU, DRRIP, and Seg-LRU schemes. While SDBP, SHiP-PC, and SHiP-ISeq all improve performance for applications, such as `excel`, SHiP-PC and SHiP-ISeq outperforms SDBP for other applications, such as `gemsFDTD` and `zeusmp`. Furthermore, while SDBP performs better than DRRIP and Seg-LRU, its performance improvement for sequential applications varies. For example, `SP` and `gemsFDTD` receive no performance benefits under SDBP. Although SDBP and SHiP use instruction-level information to guide cache line insertion and replacement decisions, SHiP-PC and SHiP-ISeq improve application performance more significantly and more consistently by an average of 9.7% and 9.4% over LRU while SDBP improves performance by only 6.9%. For the shared LLC, SHiP-PC and SHiP-ISeq outperforms the three state-of-the-art cache replacement schemes by 11.2% and 11.0% over the LRU baseline while DRRIP, Seg-LRU, and SDBP improve performance by 6.4%, 4.1%, and 5.6%.

7.4 Sensitivity to Cache Sizes

To evaluate the effectiveness of SHiP-PC and SHiP-ISeq for various cache sizes, we perform a sensitivity study for both private and shared LLCs. We find that larger caches experience less contention, so the differences in replacement approaches are reduced. However, both SHiP-PC and SHiP-ISeq still continue to improve sequential application performance over the DRRIP and LRU schemes. For a typical 4MB shared LLC on a 4-core CMP system, SHiP-PC improves the performance of all 161 multiprogrammed workloads by an average of 11.2% over the LRU scheme while DR-

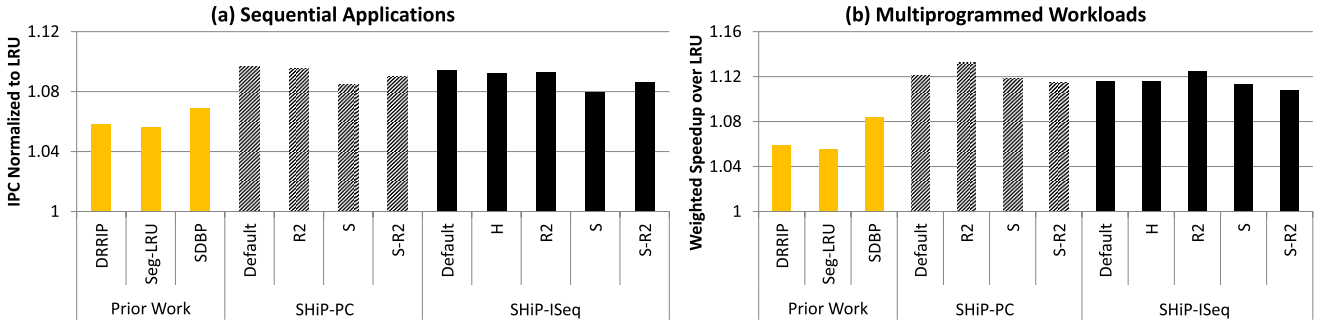

Figure 15: Performance comparison of SHiP realizations and prior work.

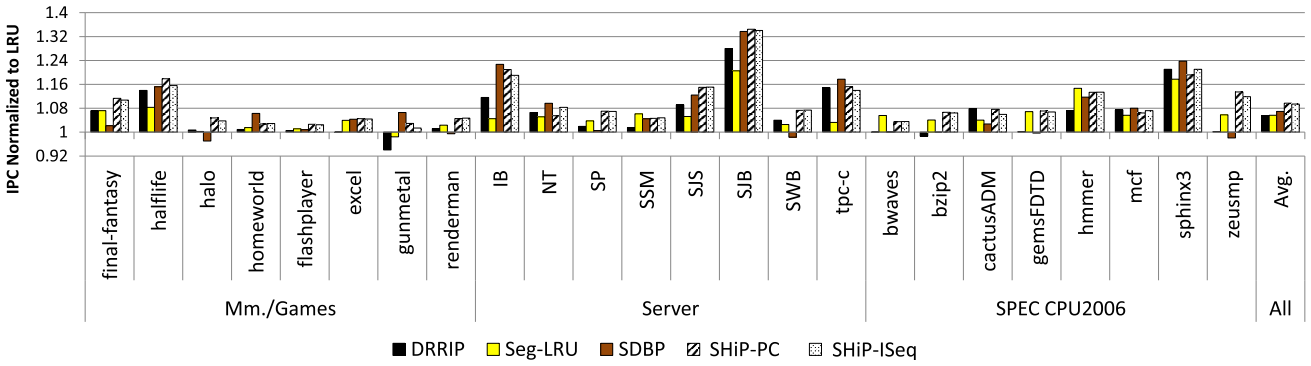

Figure 16: Comparing SHiP with DRRIP, Seg-LRU, and SDBP.

RIP improves the performance by 6.3%. As the shared cache sizes increase, SHiP-PC and SHiP-ISeq performance improvement remains significant. For a shared 32MB cache, the throughput improvement of SHiP-PC (up to 22.3% and average of 3.2%) and SHiP-ISeq (up to 22% and average of 3.2%) over LRU still doubles the performance gain under the DRRIP scheme (up to 5.8% and average of 1.1%).

7.5 Comparison of Various SHiP Realizations: Performance and Hardware Overhead

In summary, this paper presents a novel Signature-based Hit Predictor (SHiP) that learns data reuse patterns of signatures and use the signature-based information to guide re-reference prediction assignment at cache line insertion. The full-fledged SHiP-PC improves sequential application performance by as much as 34% and by an average of 9.7% over LRU.

In addition to the detailed performance analysis for the proposed SHiP scheme, we present two techniques that lead to practical SHiP designs. Figure 15 compares the performance improvement for the various implementations for SHiP-PC and SHiP-ISeq. Among the SHiP-PC variants, while using much less hardware, set sampling (SHiP-PC-S) reduces SHiP-PC performance gain slightly. Overall, SHiP-PC-S and SHiP-PC-S-R2 still outperform the prior art. The various SHiP-ISeq practical designs show similar trends.

Furthermore, while reducing the hardware overhead from 42KB for the default SHiP-PC to merely 10KB for SHiP-PC-S-R2, SHiP-PC-S-R2 can retain most of SHiP-PC's performance benefits and improve sequential application performance by as much as 32% and by an average of 9%. Table 6 gives a detailed comparison of performance improvement versus hardware overhead for the various cache replacement policies investigated in this paper. Overall, for a diverse variety of multimedia, games, server, and SPEC CPU2006 applications, the practical SHiP-PC-S-R2 and SHiP-ISeq-S-R2 designs use only slightly more hardware than LRU and DRRIP, and outperform all state-of-the-art cache replacement policies. Furthermore, across all workloads, the simple and low-overhead SHiP-PC-S-R2 and SHiP-ISeq-S-R2 schemes provide more consistent performance gains than any prior schemes.

8. Related Work

While we cannot cover all innovations in cache replacement research [1, 2, 3, 6, 10, 11, 14, 16, 18, 19, 20, 24, 26, 27, 28, 30, 31], we summarize prior art that closely resembles SHiP.

8.1 Dead Block Prediction

Lai et al. [18] proposed dead block correlating prefetchers (DBCP) that prefetch data into dead cache blocks in the L1 cache. DBCP encodes a trace of instructions for every cache access and relies on the idea that if a trace leads to the last access for a particular cache block the same trace will lead to the last access for other blocks. The proposed DBCP scheme can identify more than 90% of dead-blocks in the L1 cache for early replacement; however, a recent study [16] shows that DBCP performs less well at the last-level cache of a deep multi-level memory hierarchy.

Instead of using instruction traces to identify cache deadblocks, Liu et al. [20] proposed Cache-Burst that predicts dead blocks based on the hit frequency of non-most-recently-used (non-MRU) cache blocks in the L1 cache. Similarly, cache blocks that are predicted to have no more reuses become early replacement candidates. Cache-Burst, however, does not perform well for LLCs because cache burstiness is mostly filtered out by the higher-level caches. In addition, Cache-Burst requires a significant amount of

Table 6: Performance and hardware overhead comparison for prior work and SHiPs.

	LRU	DRRIP	Seg-LRU[5]	SDBP[16]	SHiP-PC*	SHiP-ISeq*
For 1MB LLC	8	4	8+7.68	8+13.75	4+6	4+6
Total Hardware (KB)	8	4	15.68	21.75	10	10
Selected App.	1	1.055	1.057	1.069	1.090	1.086
All App.	1	1.021	1.019	1.024	1.036	1.032
Max. Performance	1	1.28	1.21	1.33	1.32	1.33
Min. Performance	1	0.94	0.87	0.95	1.02	1.02

SHiP-PC* and SHiP-ISeq* use 64 sampling sets to train its SHCT with 2-bit counters (S-R2).

meta-data associated with each cache block.

To eliminate dead cache blocks in the LLC, Khan et al. [16] proposed Sampling Dead Block Prediction (SDBP) which predicts dead cache blocks based on the last-referenced instructions and replaces the dead blocks prior to the LRU replacement candidate. SDBP implements a three-level prediction table trained by a group of sampled cache sets, called sampler. Each cache block in the sampler remembers the last instruction that accesses the cache block. If the last-referenced instruction leads to a dead block, data blocks associated with this instruction are likely to be dead in the LLC.

A major shortcoming of SDBP is that its deadblock prediction relies on a low-associativity LRU-based sampler. Although Khan et al. claim that the LRU-based sampler is decoupled from the underlying cache insertion/replacement policy, our evaluations show that SDBP only improves performance for the two basic cache replacement policies, random and LRU. SDBP also incurs significant hardware overhead.

One can potentially describe SDBP as a signature-based replacement policy. However, the training mechanisms of both policies are fundamentally different. SDBP updates re-reference predictions on the last accessing signature. SHiP on the other hand makes re-reference predictions based on the signature that inserts the line into the cache. Correlating re-reference predictions to the "insertion" signature performs better than the "last-access" signature.

Finally, Manikantan, Rajan, and Govindarajan proposed NUcache [23] which bases its re-use distance prediction solely on instruction PCs. In contrast, SHiP explores a number of different signatures: instruction PC, instruction sequence, and memory region. Furthermore, while NUcache results in performance gains across a range of SPEC applications, this paper shows that there are significantly fewer unique PCs in SPEC applications (10's to 100's) than in multimedia, games, and server workloads (1,000's to 10,000's). This hinders NUcache's effectiveness for these workloads.

While NUcache is relevant to SHiP, NUcache requires significant modification and storage overhead for the baseline cache organization. Furthermore, SHiP's reuse prediction SHCT, on the other hand, is elegantly decoupled from the baseline cache structure. Overall, SHiP requires much less hardware overhead. Last but not least, this work explores three unique signatures to train SHiP. Instruction sequence is a novel signature, and others like instruction PC and memory address are novel in how they are applied.

8.2 Re-Reference Prediction

Instead of relying on a separate prediction table, Hu et al. [6] proposed to use time counters to keep track of the liveness of cache blocks. If a cache block has not been referenced for a specified period of time, it is predicted to be dead. These "dead" blocks become the eviction candidates before the LRU blocks [6, 31] for cache utilization optimization or can be switched off to reduce leakage [13]. In addition to the LRU counters, the proposed scheme keeps additional coarser-grained counters per cache line, which incurs more hardware requirement than SHiP.

Jaleel et al. [10] proposed SRRIP and DRRIP to learn reuse pattern of incoming cache blocks. Instead of storing the recency with each cache line, both SRRIP and DRRIP store the re-reference *prediction* with each cache line. Both use simple mechanisms to learn the re-reference interval of an incoming cache line. They do so by assigning the same re-reference prediction to the majority of cache insertions and *learning* the re-reference interval on subsequents. While simple, there is no intelligence in assigning a re-reference prediction. SHiP improves re-reference predictions by categorizing references based on distinct signatures.

Gao and Wilkerson [5] proposed the Segmented LRU (Seg-LRU) replacement policy. Seg-LRU adds a bit per cache line to observe whether the line was re-referenced or not. This is similar to the *outcome* bit stored with SHiP. Seg-LRU modifies the victim selection policy to first choose cache lines whose *outcome* is false. If no such line exists, Seg-LRU replaces the LRU line in the cache. Seg-LRU also proposes additional hardware to estimate the benefits of bypassing modifications to the hit promotion policy. Seg-LRU requires several changes to the replacement policy. On the other hand SHiP is higher performing, only modifies the insertion policy, and requires less hardware overhead.

9. Conclusion

Because LLC reference patterns are filtered by higher-level caches, typical spatial and temporal locality patterns are much harder to optimize for. In response, our approach uses signatures—such as memory region, instruction PC, or instruction path sequence—to distinguish instances where workloads are mixes of some highly re-referenced data (which should be prioritized) along with some distant-reuse data. In this paper we have presented a simple and effective approach for predicting re-referencing behavior for LLCs. The proposed SHiP mechanism, which accurately predicts the re-reference intervals for all incoming cache lines, can significantly improve performance for intelligent cache replacement algorithms. Over a range of modern workloads with high diversity in data and instruction footprint, we have demonstrated performance that is consistently better than prior work such as Seg-LRU and SDBP with much less hardware overhead. Although we evaluated our method on top of SRRIP, the re-reference predictor is a general idea applicable to a range of LLC management questions.

10. Acknowledgements

We thank the entire Intel VSSAD group for their support and feedback during this work. We also thank Yu-Yuan Chen, Daniel Lustig, and the anonymous reviewers for their useful insights related to this work. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0509402 and CNS-0720561. The authors also acknowledge the support of the Gigascale Systems Research Center, one of six centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. Carole-Jean Wu is supported in part by an Intel PhD Fellowship.

11. References

[1] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *Proc. of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[2] A. Basu, N. Kirman, M. Kirman, and M. Chaudhuri. Scavenger: A new last level cache architecture with global block priority. In *Proc. of the 40th International Symposium on Microarchitecture*, 2007.

[3] L. A. Belady. A study of replacement algorithms for a virtual storage computer. In *IBM Syst. J.*, volume 5, June 1966.

[4] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for LLCs. In *Proc. of the 42nd International Symposium on Microarchitecture*, 2009.

[5] H. Gao and C. Wilkerson. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *Proc. of the 1st JILP Workshop on Computer Architecture Competitions*, 2010.

[6] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proc. of the 29th International Symposium on Computer Architecture*, 2002.

[7] Intel Core i7 Processors http://www.intel.com/products/processor/corei7/.

[8] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proc. of the 4th Workshop on Modeling, Benchmarking and Simulation*, 2008.

[9] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proc. of the 17th International Conference on Parallel Architecture and Compilation Techniques*, 2008.

[10] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. of the 38th International Symposium on Computer Architecture*, 2010.

[11] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. of the International Conference on Measurement and Modeling of Computer Systems*, 2002.

[12] JILP Workshop on computer architecture competitions http://jilp.org/jwac-1/.

[13] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proc. of the 28th International Symposium on Computer Architecture*, 2001.

[14] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *Proc. of the 25th International Conference on Computer Design*, 2007.

[15] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi. Using dead blocks as a virtual victim cache. In *Proc. of the 19th International Conference on Parallel Architecture and Compilation Techniques*, 2010.

[16] S. M. Khan, Y. Tian, and D. A. Jiménez. Dead block replacement and bypass with a sampling predictor. In *Proc. of the 43rd International Symposium on Microarchitecture*, 2010.

[17] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. In *IEEE Trans. Comput.*, volume 57, April 2008.

[18] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. of the 28th International Symposium on Computer Architecture*, 2001.

[19] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes that least recently used and least frequently used policies. In *IEEE Trans. Comput.*, volume 50, December 2001.

[20] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proc. of the 41st International Symposium on Microarchitecture*, 2008.

[21] G. Loh. Extending the effectiveness of 3D-stacked DRAM caches with an adaptive multi-queue policy. In *Proc. of the 42nd International Symposium on Microarchitecture*, 2009.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[23] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: An efficient multicore cache organization based on next-use distance. In *Proc. of the 17th International Symposium on High Performance Computer Architecture*, 2011.

[24] N. Megiddo and D. S. Modha. A self-tuning low overhead replacement cache. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[25] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proc. of the 37th International Symposium on Microarchitecture*, 2004.

[26] P. Petoumenos, G. Keramidas, and S. Kaxiras. Instruction-based reuse-distance prediction for effective cache management. In *Proc. of the 9th International Conference on Systems, Architectures, Modeling and Simulation*, 2009.

[27] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for high performance caching. In *Proc. of the 35th International Symposium on Computer Architecture*, 2007.

[28] K. Rajan and G. Ramaswamy. Emulating optimal replacement with a shepherd cache. In *Proc. of the 40th International Symposium on Microarchitecture*, 2007.

[29] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[30] R. Subramanian, Y. Smaragdakis, and G. Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *Proc. of the 39th International Symposium on Microarchitecture*, 2006.

[31] C.-J. Wu and M. Martonosi. Adaptive timekeeping replacement: Fine-grained capacity management for shared CMP caches. In *ACM Trans. Archit. Code Optim.*, volume 8, February 2011.

[32] Y. Xie and G. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. of the 37th International Symposium on Computer Architecture*, 2009.

[33] T. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proc. of the 24th International Symposium on Microarchitecture*, 1991.