Temporal-Based Multilevel Correlating Inclusive Cache Replacement

YINGYING TIAN, Texas A&M University, AMD
SAMIRA M. KHAN, Carnegie Mellon University, Intel
DANIEL A. JIMÉNEZ, Texas A&M University

Inclusive caches have been widely used in Chip Multiprocessors (CMPs) to simplify cache coherence. However, they have poor performance compared with noninclusive caches not only because of the limited capacity of the entire cache hierarchy but also due to ignorance of temporal locality of the Last-Level Cache (LLC). Blocks that are highly referenced (referred to as *hot blocks*) are always hit in higher-level caches (e.g., L1 cache) and are rarely referenced in the LLC. Therefore, they become replacement victims in the LLC. Due to the inclusion property, blocks evicted from the LLC have to also be invalidated from higher-level caches. Invalidation of hot blocks from the entire cache hierarchy introduces costly off-chip misses that makes the inclusive cache perform poorly.

Neither blocks that are highly referenced in the LLC nor blocks that are highly referenced in higher-level caches should be the LLC replacement victims. We propose *temporal-based multilevel correlating cache replacement* for inclusive caches to evict blocks in the LLC that are also not hot in higher-level caches using correlated temporal information acquired from all levels of a cache hierarchy with minimal overhead. Invalidation of these blocks does not hurt the performance. By contrast, replacing them as early as possible with useful blocks helps improve cache performance. Based on our experiments, in a dual-core CMP, an inclusive cache with temporal-based multilevel correlating cache replacement significantly outperforms an inclusive cache with traditional LRU replacement by yielding an average speedup of 12.7%, which is comparable to an enhanced noninclusive cache, while requiring less than 1% of storage overhead.

Categories and Subject Descriptors: B.3.2 [**Design Styles**]: Cache Memories

General Terms: Design, Performance

Additional Key Words and Phrases: Inclusive caches, replacement policy, chip multiprocessors, last-level cache

ACM Reference Format:
Tian, Y., Khan, S. M., and Jiménez, D. A. 2013. Temporal-based multilevel correlating inclusive cache replacement. ACM Trans. Architec. Code Optim. 10, 4, Article 33 (December 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555290

1. INTRODUCTION

An inclusive cache hierarchy has been widely used in Chip Multiprocessors (CMPs) due to its simplicity in maintaining cache coherence [Baer and Wang 1988; Chen et al. 2006]. However, compared to exclusive [Jouppi and Wilton 1994] or noninclusive [Zahran et al. 2007; Zheng et al. 2004] cache hierarchies, inclusive cache hierarchies have

This work was done when Yingying Tian was interning at AMD Research.
This research was supported by Texas NHARP grant 010115-0079-2009 and National Science Foundation grant CCF-1162215.
Authors' addresses: Y. Tian and D. A. Jiménez, Department of Computer Science and Engineering, Texas A&M University; S. M. Khan, Computer Architecture Lab, Carnegie Mellon University.

DOI: http://dx.doi.org/10.1145/2555289.2555290

limited performance due to the inclusion property that all cache blocks in higher-level caches need be a subset of the shared Last-Level Cache (LLC). When the sum of the sizes of all core caches is comparable to the size of the LLC, overall capacity of the inclusive cache hierarchy becomes limited compared to exclusive and noninclusive caches and the performance becomes poor. Moreover, when cache blocks in the inclusive LLC are replaced, they must also be invalidated from all higher-level caches to maintain inclusion. Because temporal locality is hidden by higher-level caches, hot blocks that are highly referenced in higher-level caches are rarely accessed in the LLC and therefore become LLC replacement victims and get invalidated from the entire cache hierarchy, eventually causing cache misses and incurring hundreds of cycles of memory access penalties. Although the capacity of an inclusive cache hierarchy cannot be enlarged to that of a noninclusive or exclusive one under the same configuration, the performance can still be improved if the replacement and invalidation of blocks do not cause costly off-chip misses.

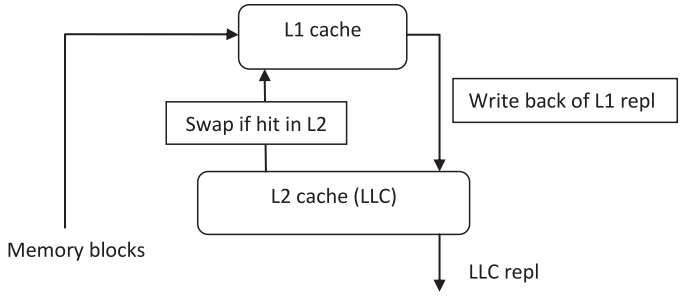
We propose a Temporal-Based Multilevel Correlating (TMC) cache replacement technique for inclusive cache hierarchies. It replaces LLC blocks that will not be re-referenced in either higher-level caches or in the LLC. Replacement and consequent invalidation of these blocks will not cause extra memory accesses. By contrast, replacing these blocks with useful ones as early as possible helps improve cache efficiency. This technique correlates LLC access patterns and temporal information passively hinted by higher-level caches to accurately choose LLC replacement candidates with minimal overhead. Compared to previous techniques, TMC inclusive cache replacement achieves significant performance improvement with minimal storage and communication overhead. Based on our experiments, in a dual-core CMP with a three-level inclusive cache hierarchy, the TMC cache replacement technique yields a speedup of 12.7% over an inclusive cache with LRU replacement.

Not only to achieve similar performance with noninclusive caches, TMC provides more efficient inclusive cache hierarchies by invalidating harmless and useless blocks as early as possible and placing useful blocks, which actually outperforms non-inclusive caches by 7% and achieves comparable performance to enhanced noninclusive caches based on our experiments, while consuming less than 1% of storage overhead compared to inclusive caches.

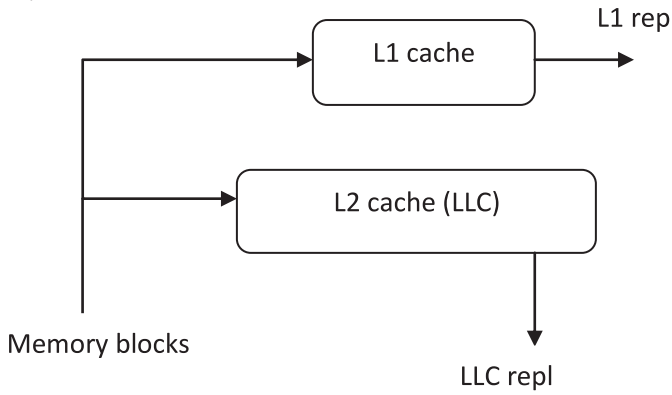
This article makes the following contributions:

—We propose TMC cache replacement for inclusive cache hierarchies. It chooses blocks that will not be re-referenced in all cache levels as LLC replacement candidates. Replacing these blocks with useful ones as early as possible significantly helps improve cache efficiency and overall performance.
—We propose to sample LLC access patterns and correlate them with temporal locality knowledge passively acquired from higher-level caches to choose temporal-aware LLC replacement candidates, which provides high-performance improvement while consuming minimal overhead.
—We show that inclusive caches with TMC is more efficient than not only the inclusive baseline but also the "upper-bound"– noninclusive caches. Inclusive caches with TMC perform as well as enhanced noninclusive caches while keeping the advantage of simplifying cache coherence of CMPs.

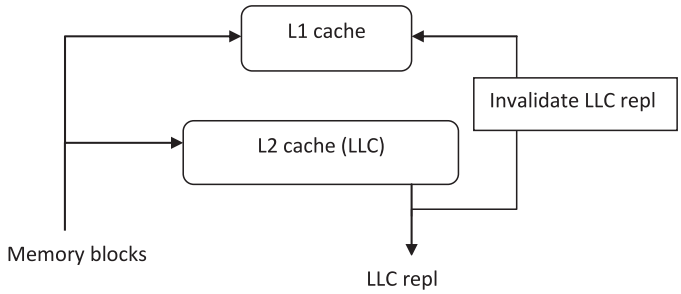
The organization of this article is as follows: Section 2 motives the proposed technique. Section 3 describes the TMC cache replacement in detail, and Section 4 discusses the experimental methodology we use, followed by expected results in Section 5. Section 6 introduces related work. Section 7 summarizes all work in this article and discusses future work.



(a) Block placement and replacement in an exclusive cache hierarchy.



(b) Block placement and replacement in a noninclusive cache hierarchy.



(c) Block placement and replacement in an inclusive cache hierarchy.

Fig. 1. Cache hierarchy designs.

2. MOTIVATION

In a multilevel cache hierarchy, there are three types of cache architectures: inclusive, exclusive, and noninclusive. Figure 1 illustrates the differences among them using a two-level cache hierarchy in examples. In an exclusive cache hierarchy (Figure 1(a)), data is either stored in the L1 cache or in the L2 cache but cannot be located in both. When a block is replaced from the L1 cache, it is inserted into the L2 cache; when a block is referenced in the L2 cache, it is swapped back into the L1 cache. The capacity of the exclusive cache hierarchy therefore equals to the sum of the capacities of the L1 and the L2 caches. As shown in Figure 1(b), a noninclusive cache does not have any requirements of the duplication of blocks. When a block is inserted into the noninclusive cache hierarchy, it is inserted in both the L1 and the L2 caches. Then

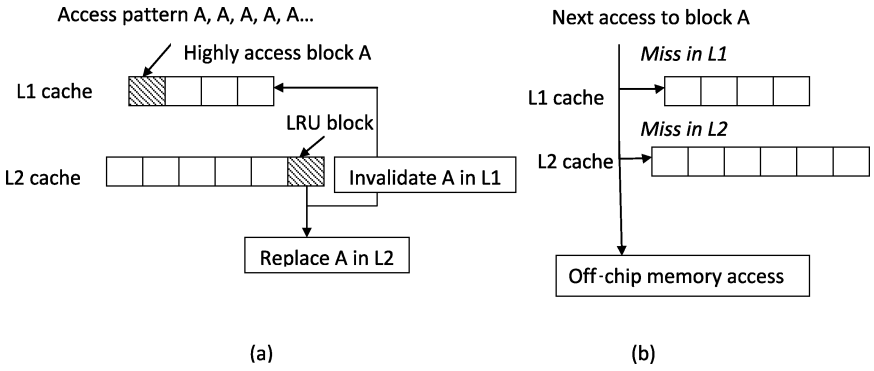


Fig. 2. An example of inclusion problem.

it is promoted/replaced according to the replacement policies applied in each cache level separately. A noninclusive cache hierarchy relaxes the constraint of restricted inclusion or exclusion, and the overall capacity is smaller than that of an exclusive cache hierarchy and greater than the capacity of an inclusive cache hierarchy. In an inclusive cache hierarchy, as shown in Figure 1(c), cache blocks stored in the L1 cache should also be stored in the L2 cache. When a block is evicted from the L2 cache, the corresponding block in the L1 cache (if present) has to be invalidated to maintain inclusion (referred to as back-invalidation [Jaleel et al. 2010]). Thus, the capacity of the whole inclusive cache hierarchy equals to capacity of its LLC (the L2 cache in this example).

Compared to exclusive and noninclusive cache hierarchies, inclusive cache hierarchies simplify cache coherence and are therefore widely used in CMPs [Casazza 2009]. However, inclusive caches have limited performance because of the inclusion property. All blocks stored in higher-level caches also have to be in the inclusive LLC. Thus, the overall cache capacity is smaller than noninclusive or exclusive caches with the same configuration. Moreover, to maintain inclusion, LLC replacement victims must be back-invalidated from higher-level caches. Because hot blocks are always hit in higher-level caches and rarely accessed in the LLC, blocks with high temporal locality in higher-level caches tend to become LLC replacement victims and are removed from the whole cache hierarchy. Further accesses to these hot blocks will cause hundreds of cycles of memory access penalty to access them from off-chip memory.

Figure 2 illustrates the problem of inclusive caches in a two-level inclusive cache hierarchy. In this example, the L1 cache is a 4-way set-associative cache, and the L2 cache (the LLC) is a 6-way set-associative inclusive cache. The memory access pattern is as simple as A, A, A, A, A..., where block A keeps high temporal locality. Due to the fact that temporal locality is hidden by higher-level caches, block A is highly accessed in the L1 cache and becomes the LRU replacement victim in the L2 cache (as shown in Figure 2(a)). Thus, block A is invalidated from the L1 cache to maintain inclusion. Further requests to block A hurt system performance by bringing it back from off-chip memory, as shown in Figure 2(b). Figure 3 shows the performance gap between noninclusive caches and inclusive caches for eight dual-core multiprogrammed workloads. The methodology used in this experiment is described in Section 4. Each mix in this figure is a combination of two randomly selected benchmarks from SPEC CPU 2006 [Henning 2006]. The average performance gap is 3.9%.

To bridge the performance gap between noninclusive and inclusive caches, one naive solution would be increase the size of the inclusive LLC. However, the chip area occupied by caches is already more than half of the overall chip area [Wendel et al. 2011;

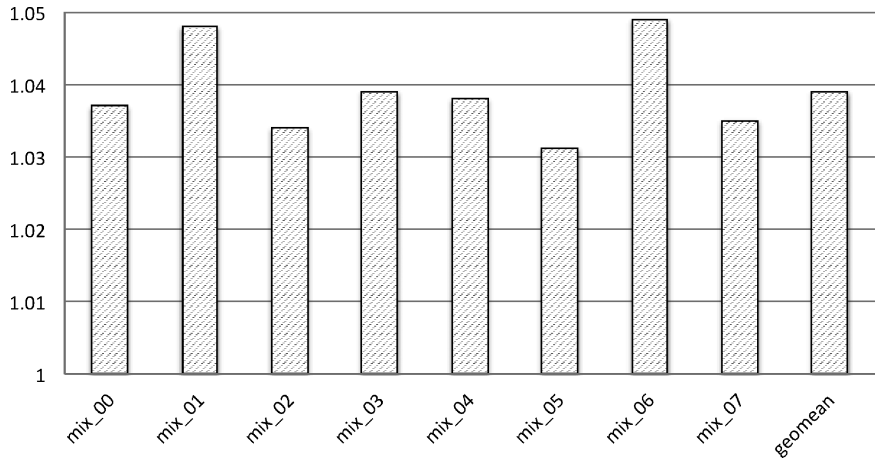


Fig. 3. Performance of noninclusive caches normalized to inclusive caches.

Kurd et al. 2010; Sanchez and Kozyrakis 2010], which contributes to significant power consumption. Simply increasing cache sizes will not help performance improvement.

Another way to improve inclusive caches is to intelligently choose LLC replacement candidates that have low temporal locality in higher-level caches; back-invalidation of these blocks will not cause performance loss. Previous work [Jaleel et al. 2010] identified blocks that have high temporal locality in higher-level caches and reduced the frequency of back-invalidating them, which makes performance of inclusive cache hierarchies achieve that of noninclusive caches. However, blocks that have poor temporal locality in higher-level caches may still have temporal locality in the LLC, and the replacement of these blocks will still hurt the overall performance. If a block will not be referenced in either higher-level caches or the LLC,[1] replacement and consequent back-invalidation of this block will not hurt performance. In fact, cache performance can be improved by replacing these known replaceable blocks with others deemed more useful. Inclusive caches are capable to outperform noninclusive caches in this way.

We categorize LLC blocks into three exclusive groups based on their temporal characteristics in both higher-level caches and the LLC:

—*HAH blocks*: blocks that are highly referenced in higher-level caches;
—*HAL blocks*: blocks that are highly referenced in the LLC;
—*LAL blocks*: blocks that have low temporal locality in both higher-level caches and the LLC, which should be the group of LLC replacement candidates.

The LLC replacement candidates chosen from LAL blocks will not hurt inclusive cache performance. By contrast, replacing these blocks with useful ones as early as possible helps improve cache efficiency.

Figure 4 shows the average percentages of different categories of blocks that are back-invalidated due to the LRU replacement in the LLC. On average, 72.51% of the back-invalidated blocks are referenced again before their eviction in corresponding sets of a noninclusive L1 cache. These blocks are HAH blocks and should not be invalidated from the cache hierarchy. There are 15.82% of blocks that are not hit in the L1 cache but are re-referenced in the LLC before being replaced from the LLC. These blocks are HAL

[1]At the time an invalidated block is requested again, if it is still kept in a corresponding cache of a noninclusive cache hierarchy with the same replacement policy, it is treated as being re-referenced before its eviction; otherwise, it will not be referenced until its eviction.

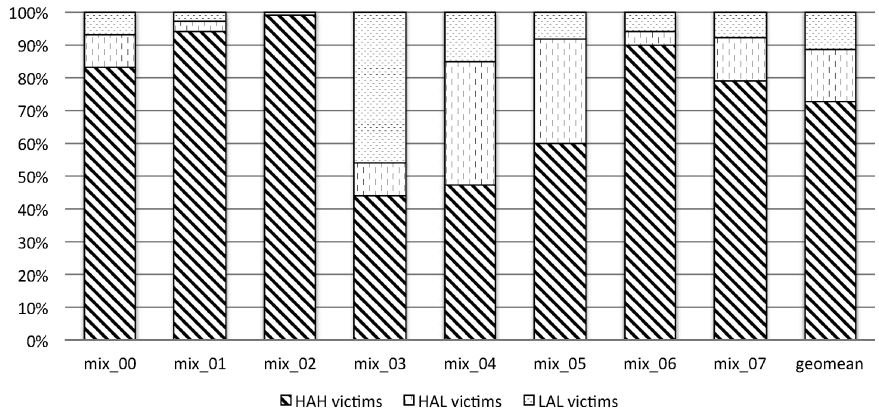


Fig. 4. Percentage of different categories of back-invalidated blocks due to LLC LRU replacement.

blocks, which have temporal locality in the LLC and should not be invalidated from whole cache hierarchy, either. The remaining 11.67% of back-invalidated blocks are not re-referenced until they are evicted from the LLC in a corresponding noninclusive cache hierarchy. These blocks are LAL blocks, and replacement of these blocks are harmless to the performance of inclusive caches and should be chosen as LLC replacement victims.

3. TEMPORAL-BASED MULTILEVEL CORRELATING CACHE REPLACEMENT

We propose TMC cache replacement to choose LLC blocks that have low temporal locality in all caches as inclusive LLC replacement candidates. The technique intelligently categorizes LLC blocks into three exclusive groups as stated previously using two-level categorization. It first categorizes LLC blocks based on their local temporal locality in the LLC into two groups: HAL and P-LAL (potential LAL blocks). Then it identifies HAH blocks from P-LAL blocks and replaces the other LAL blocks when needed. It uses a Correlating Temporal Locality (CTL) detector to detect LAL blocks with high accuracy. To first categorize HAL and P-LAL blocks, a CTL detector uses sampled Program Counters (PCs) to determine when an LLC block is likely to be a P-LAL block. The key intuition behind TMC is that if a memory access instruction PC leads to a P-LAL block, then there is a high probability that the same PC will lead to another P-LAL block. Previous work has found correlations between observed patterns of memory access instructions and cache accesses [Lai and Falsafi 2000; Lai et al. 2001; Somogyi et al. 2004; Lebeck and Wood 1995; Khan et al. 2010a, 2010b; Wu et al. 2011]. As stated in Lai and Falsafi [2000], the correlation arises because "program behavior is repetitive, e.g., a critical section used a fixed set of instructions to read and modify data." If an instruction is "repeatable and always leads to (and can be associated with) the same event, a predictor can dynamically learn the behavior and accurately predict the event." "Much as path-based predictors [Nair 1995] predict conditional branches dynamically based on correlating a sequence of basic-block addresses," a PC-based predictor predicts an event dynamically based on correlating PCs of memory access instructions. In the LLC, temporal locality is filtered by higher-level caches and memory access patterns are roughly consistent across groups of sets. Thus, the learning acquired through sampling a few sets generalizes to the entire LLC [Qureshi et al. 2007; Khan et al. 2010b]. Using sampled PC information to detect P-LAL blocks are accurate and consumes little. After the first-level categorization, the CTL detector detects HAH blocks from the P-LAL group, using temporal information passively acquired from higher-level caches. To get temporal information filtered by higher-level caches, the naive way is to send this

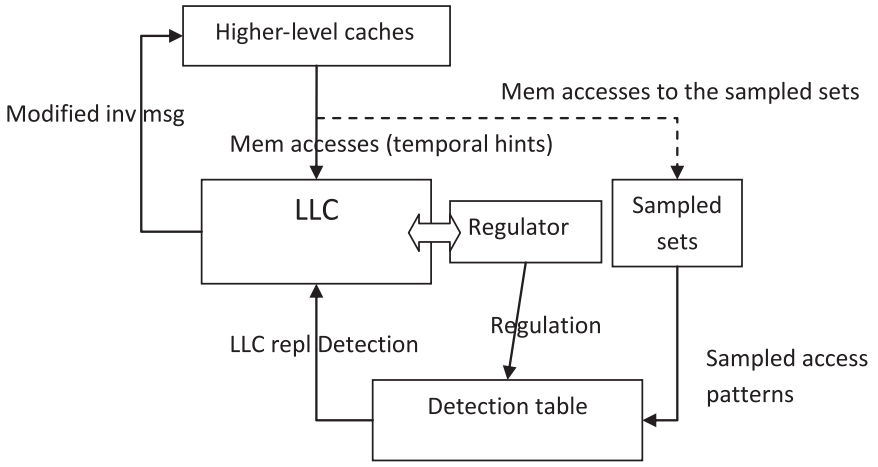


Fig. 5. Block diagram of CTL detector.

information to the LLC actively on every cache hit in higher-level caches. However, the number of cache hit in higher-level caches is extremely large, and sending that number of requests to the LLC will consume a lot of bandwidth and energy. Therefore, in TMC, the temporal locality of higher-level caches is passively acquired by the LLC on each LLC miss, which is far less than the number of caches hit in higher-level caches.

Compared to previous work, this technique has following advantages: (1) detecting temporal-aware LLC replacement candidates with high accuracy and minimal storage overhead, (2) correlating multilevel temporal information with minimal communication overhead, (3) self-training at runtime for accurate detection. We discuss the design of the CTL detector and how it works in detail in the following sections.

3.1. Correlating Temporal Locality Detector

A CTL detector consists of a detection table, a decoupled structure storing sampled LLC sets, a detection regulator, and a modified invalidation message format. Figure 5 gives a block diagram of a CTL detector, showing the structure and related communication.

3.1.1. Detection Table. The detection table is a hash table of saturating counters, indexing by a hashed PC. When an LLC block is referenced, the corresponding PC that accesses this block is hashed to access the detection table to determine its group. Each access to the detection table yields a confidence compared with a threshold; as long as the threshold is not reached, blocks accessed by that PC are likely to be HAL blocks; otherwise, blocks are grouped as P-LAL blocks.

To reduce the impact of conflicts in the table, the detection table uses the skewed organization [Seznec 1993; Michaud et al. 1997] of three tables. Each access to the detection table yields three values of counters that are summed up to compare with a threshold. In our experiments, the detection table has three 4,096-entry tables of 2-bit saturating counters, each indexed by a different hash function of a 15-bit partial PC. The skewed prediction table consumes a total of 3KB in storage.

3.1.2. Sampled Access Patterns. To reduce the storage and power overhead, the detection table is only updated on a small fraction of cache accesses referred to as sampled access patterns. As stated earlier, the intuition is that memory access patterns are roughly consistent across sets. Thus, the CTL detector keeps track of program behavior by sampling a small number of LLC sets using a decoupled structure containing only

partial tags and kept outside the LLC [Qureshi and Patt 2006]. This structure can be configured differently than the configuration of the LLC to provide improved detection accuracy [Khan et al. 2010b]. For instance, we find that for a 16-way set-associative LLC, a reduced associativity of 12 ways provides accurate detection with less state. The LLC is decoupled from the sampled sets and does not require keeping extra PC information to update the detection table. Thus, each cache block in the LLC only holds two extra bits of metadata to store the categorization information, which consumes less than 0.5% of a 2MB LLC, further reducing the storage overhead. The sampled sets are accessed in parallel with the LLC. When an LLC access occurs in a sampled set, the CTL detector hashes the PC that accessed this block to index the detection table and update the corresponding saturating counter. Accesses to blocks whose sets are not in the sampled sets will not update the detection table. In our experiments, the sampled sets contain only 64 sets of tags, randomly selected from the LLC. Each sampled set has 12 entries consisting of 15-bit partial tags, 15-bit partial PCs, and other metadata used for CTL detection, consuming 3.375KB of total storage overhead. Compared to accessing the detection table on each LLC access, the number of accesses to the detection table is reduced by more than 95%. Note that more sampled sets slightly improves the detection accuracy, whereas too many sets can increase destructive interference in the detection tables. Since power overhead is a serious issue in cache design, we choose to slightly sacrifice the performance improvement for far less power consumption.

3.1.3. Modified Invalidation Message Format. The P-LAL group consists of HAH blocks and LAL blocks. In the second-level categorization, a CTL detector randomly back-invalidates P-LAL blocks before replacing them from the LLC. The intuition is this: if the block is a HAH block, it will be requested soon by higher-level caches; otherwise, it is a LAL block and can be replaced. To invalidate blocks from higher-level caches, we simply modify the format of invalidation message instead of changing default inclusion protocol.

In a conventional inclusive cache hierarchy, on each LLC miss, the LLC sends an invalidation message to all higher-level caches with the physical address of the replacement victim. If the block is present in any caches, it is invalidated from those caches. Instead of generating extra messages, we modify the format of the invalidation message with extra physical address fields. On each LLC miss, the CTL detector sends one invalidation message encapsulating the physical address of a replacement victim together with N physical addresses of P-LAL blocks. There is no extra control message involved. The value of N is related to traffic overhead. Although the inclusion protocol is unchanged, the throughput of on-chip network is increased by N. A large N will also invalidate more higher-level cache blocks and may cause unnecessary cache misses. Based on our experiments, $N = 1$ is sufficient for accurate detection as well as minimal communication overhead. Higher-level caches de-encapsulate the invalidation message and invalidate blocks with addresses stored in the message. Higher-level caches do not send any acknowledgment or temporal information to the LLC. Temporal locality information is passively acquired by the LLC with subsequent LLC accesses.

3.1.4. Detection Regulator. A detection regulator is used to regulate previous P-LAL detection. If the block is a HAH block, it will be requested soon by higher-level caches and an LLC hit will occur. The detection regulator therefore gets the hint that the block should be kept in the higher-level cache(s). Thus, the previous P-LAL is remarked as a HAH block, and its replacement state is updated. If the tag of block is located in the sampled sets, the corresponding counters in the detection table are also updated. If the block is not requested after certain cycles, it is treated as harmless for replacement. The detection regulator then marks the block as a LAL block and reinforces previous detection if the block is in the sampled set. The number of cycles that the regulator

waits before tagging LAL blocks is related to the accuracy of detection. If it waits longer, there is a higher probability of making a more accurate detection. However, it also delays the procedure of grouping blocks. Based on our experiments, waiting until another LLC miss occurs is sufficient to make accurate and timely decisions.

3.2. How Does Temporal-Based Multilevel Correlating Work?

In this section, we describe the TMC algorithm in detail.

On each LLC access, the technique first checks whether the set of the requested block is in the sampled sets. If so, the sampled tag array is accessed in parallel with the LLC; otherwise, only the LLC is accessed. On an access to a sampled set, if the tag is in the set, it is a sampled hit, the partial PC stored in the corresponding tag entry is used to index to the detection table, and the counter of the detection table entry is decremented by one, indicating that the stored PC is likely to lead to a HAL block. The stored partial PC is updated to the PC that currently requests the block. The corresponding replacement status is updated (e.g., the accessed block is moved to the MRU according to LRU replacement policy). The categorization of the current block is decided by the detection table with the stored PC and comparing the corresponding counter with the threshold; if the threshold is not reached, the current block is marked as a HAL block; or it is marked as a P-LAL block. If the accessed block is not in the sampled set, it is a sampled miss and a replacement candidate is needed. If there is a LAL block marked in the sampled set, it is the replacement victim; if there is no LAL block in current set, a P-LAL block is chosen; if there is neither LAL nor P-LAL block, a normal LRU block is replaced. The partial PC stored in the replacement victim entry is indexed into the detection table, and the corresponding counter in the detection table entry is incremented by one, indicating that the stored PC is likely to lead to a P-LAL block. Then the partial tag, partial PC, and replacement status are updated. Finally, the group of the block is updated by hashing the partial PC into the detection table and comparing the corresponding counter with the threshold.

The LLC is accessed in parallel with the sampled sets. On LLC hit, the PC of the memory instruction that accesses this block is indexed into the detection table to determine the categorization of the accessed block. Replacement status and other metadata are also updated. On LLC miss, a LAL block is chosen for replacement. If there is no LAL block, a P-LAL is chosen; if there is no P-LAL block, the LRU block is replaced. The corresponding metadata of the incoming block are updated, and the categorization of the coming block is made by indexing the PC that caused the LLC miss into the detection table and comparing the counter with the threshold. Note that with TMC, the LLC may use less costly replacement policies (not recently used replacement, random replacement, etc.) to further reduce storage overhead because the sampled sets are decoupled from the LLC and the detection table is updated only with sampled information. To fairly evaluate our technique, we conservatively use the LRU replacement policy in our experiments to maintain consistency with other techniques.

To maintain inclusion, the address of the replacement victim is sent back to all higher-level caches for invalidation. Besides the address of the replaced block, the address of a P-LAL block (if there are P-LAL blocks marked, as shown in Figure 6(a)) is encapsulated into the back-invalidation message packet as well for temporal hints from higher-level caches. Both the replaced block and the selected P-LAL block is invalidated from all higher-level caches if presented (Figure 6(b)). It is to detect HAH blocks from consequent behaviors of higher-level caches according to the intuition: if the block is a HAH block, it will be requested soon by higher-level caches (Figure 6(c)); otherwise, it will not be requested until being evicted. If the block is re-referenced before the next LLC miss occurs, it is an LLC hit to a P-LAL block. Replacement status of this block is updated, indicating that it keeps temporal locality, and this block is

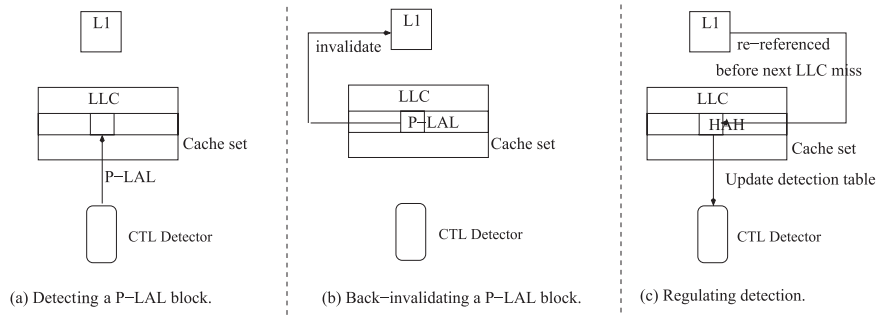


Fig. 6. Detect HAH blocks from P-LAL blocks.

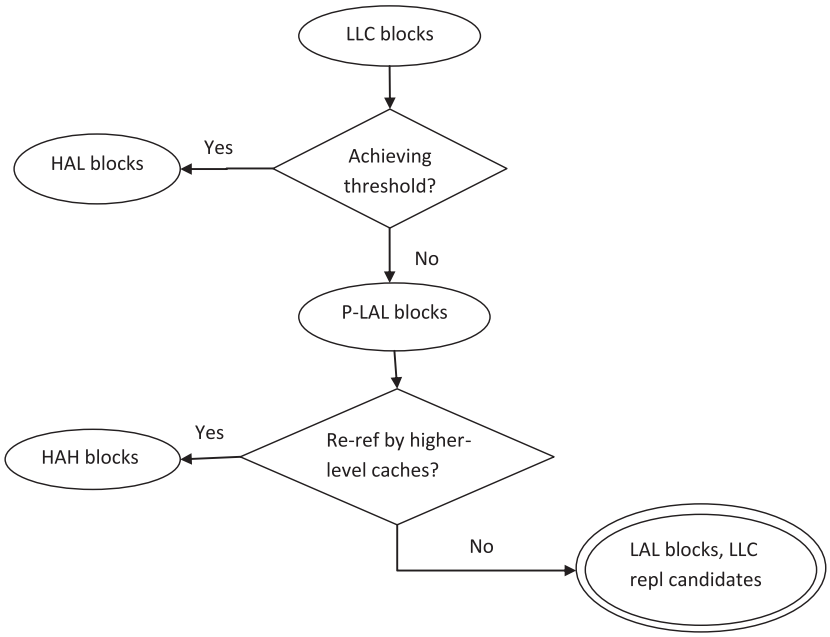


Fig. 7. Two-level detection of LLC block categorization.

marked as HAH instead of P-LAL. If the set where this block locates is sampled, the corresponding counter in the detection table is also updated. If the back-invalidated P-LAL block is not referenced until another LLC miss occurs, it is marked as a LAL block and can be replaced.

Figure 7 shows the two level detection of the categories of LLC blocks. HAL blocks should be kept in the LLC, and HAH blocks should be kept in the corresponding higher-level caches; LAL blocks can be replaced from the LLC and invalidated from the whole inclusive cache hierarchy without causing performance loss.

3.3. Comparison with Previous Work

3.3.1. Temporal Locality Aware Policy Suite. The Temporal Locality Aware (TLA) policy suite [Jaleel et al. 2010] was proposed to improve inclusive cache performance. It consists of three policies: Temporal Locality Hints (TLH), Early Core Invalidation (ECI), and Query-Based Selection (QBS). As claimed in Jaleel et al. [2010], TLH is only a limit

study; ECI is a lower traffic solution with limited performance; and QBS performs best among three TLA policies, achieving similar performance to a noninclusive cache. The goal of TLA is to identify hot blocks in higher-level caches (a.k.a. HAH blocks according to our definition) and to avoid replacing these blocks from the LLC. Although the replacement victims chosen by TLA are not highly accessed in higher-level caches, there is a chance that they will be re-referenced in the LLC (a.k.a. HAL blocks in our definition). Compared to TLA, TMC identifies hot blocks in higher-level caches and also hot blocks in the LLC, and avoids replacing these blocks from the LLC. Cache efficiency is further improved by bringing useful blocks into the cache as early as possible. Therefore, instead of achieving similar performance, TMC actually outperforms noninclusive caches significantly with low overhead.

Compared to QBS, the best management policy of TLA suite, TMC has not only better performance improvement but also lower communication overhead. On each LLC replacement, QBS chooses a block and queries to see if it is present in any core caches. If the block is located in some core caches, QBS has to find another block in the LLC and repeats the query procedure again until it finds a block absent in all core caches to replace. If the number of queries is unlimited, up to 1.5KB of data are transferred on-chip on each LLC miss in a dual-core CMP, compared to 32 bytes per LLC miss with TMC. The query number of QBS is limited, as stated in Jaleel et al. [2010], as at least two queries per LLC miss is required to achieve acceptable performance, the on-chip communication overhead is still six times more than that of TMC.

Compared to TLA suite, TMC requires extra PC information sent to the LLC. Sending this extra information to the LLC has been proposed by much previous work [Lai and Falsafi 2000; Lai et al. 2001; Somogyi et al. 2004; Lebeck and Wood 1995; Khan et al. 2010a, 2010b; Wu et al. 2011]. TMC has higher storage overhead, but it is as low as less than 1% of the capacity of the LLC in a dual-core CMP. TMC has lower communication overhead compared to QBS. TMC outperforms ECI and QBS by 10.7% and 8.6%, respectively.

3.3.2. Sampling Dead Block Prediction. Sampling Dead Block Prediction (SDBP) [Khan et al. 2010b] is a recently proposed dead block prediction technique. SDBP is designed to identify dead blocks in the LLC and replace them with live blocks as early as possible to improve cache efficiency. Compared to other dead block prediction techniques, SDBP uses far less overhead to make predictions with much higher accuracy. However, since SDBP has no awareness of temporal locality in core caches, predictions are made based on local information of LLC accesses. Therefore, the predicted dead block in the LLC can be highly referenced blocks in core caches, and the replacement of these blocks will cause costly off-chip cache misses that hurt the inclusive cache performance.

We compare our work with this local dead block prediction technique from performance to overhead. Based on our experiments, TMC achieves an average performance improvement of 5.2% over SDBP in an inclusive cache hierarchy. Moreover, TMC performs comparably to an enhanced noninclusive cache with SDBP, which utilizes SDBP in a noninclusive cache. The storage overhead of TMC is 4KB compared to SDBP, and the on-chip communication overhead is 32 bytes on each LLC miss.

The performance comparison will be shown in detail in Section 5.

4. EXPERIMENTAL METHODOLOGY

This section outlines the experimental methodology used in this study.

4.1. Simulation Environment

We use the MARSSx86 cycle-accurate simulator, a full-system simulation of the x86-64 architecture. We use the multicore implementations [Patel et al. 2011] with extensive enhancements for improved simulation accuracy and performance. It models an

out-of-order 4-wide, 5-stage pipeline with a 128-entry reorder buffer, coherent caches with MESI protocol, and on-chip interconnections. We modified the simulator to collect Instructions-Per-Cycle (IPC) figures as well as cache misses.

The microarchitectural parameters closely model Intel Core i7 [Casazza 2009] with the following parameters (the same as in Jaleel et al. [2010]): three-level cache hierarchy, L1, L2, and a shared LLC. The L1 and L2 caches are private in each core. The L1 I-cache and D-cache are 4-way 32KB each, and the L2 cache is unified 8-way 256KB. As in the Intel Core i7, inclusion is not enforced between private L1 and L2 caches. The shared LLC is a unified 2MB cache for dual-core CMP and 2MB per core for 4-core CMP and 8-core CMP. We simulate a dual-core CMP with 2MB LLC to compare with previous work [Jaleel et al. 2010] that ran experiments under this configuration. However, the configuration of 2MB per core LLC is more realistic in current industrial design. The block size of all caches in the hierarchy is 64 bytes. The access latencies for the L1, L2, LLC, and main memory are 1, 10, 24, and 250 cycles, respectively. The default replacement policy of each cache is the LRU replacement policy.

In TMC, there are 64 sampled sets of tags, evenly chosen from among the sets of the LLC. Note that for 2MB, 4MB, 8MB, and 16MB caches, the number of sampled sets is constant—that is, the storage overhead of sampled sets does *not* increase with core count. Each sampled set contains 12 entries consisting of a 15-bit partial tag, a 15-bit partial PC, and 2 bits of categorization. The detection table consists of three 4,096-entry tables of 2-bit saturating counters, also regardless of core count. For each LLC block, we store an additional 2 bits of categorization for TMC. Note that TMC does not make any prediction for prefetched blocks. Prefetched blocks are inserted and replaced using default LRU replacement policy. QBS requires a control mechanism to maintain information about the presence of a queried block in all higher-level caches.

4.2. Benchmarks

We use the SPEC CPU 2006 benchmark suite [Henning 2006]. Each benchmark is compiled for the x86-64 instruction set. Note that not all of the workloads will hurt inclusive cache performance. Based on our experiments, some workloads running with an inclusive LLC perform similar to a noninclusive LLC. We classify workloads into two categories: inclusion-sensitive and inclusion-insensitive workloads. To evaluate whether a certain technique can help improve the performance of both categories of workloads, we run 16 dual-core workloads: eight of them are inclusion sensitive (the selection criteria is that the performance gap between the inclusive cache and the noninclusive cache is larger than 3%), and the other eight workloads are inclusion insensitive (the performance gap is smaller than 3%). We also randomly selected five 4-core workloads whose average performance gap is 1.7%, and five 8-core workloads with an average performance gap of 1.2% to evaluate the scalability of all techniques. In each workload, benchmarks run simultaneously, restarting after one billion instructions until another two billion instructions (four billion instructions for 8-core workloads) are totally executed. Tables I, II, III, and IV show the workload mixes that we use in the experiments, respectively. We also simulated ECI, QBS, and SDBP in both the inclusive hierarchy and the noninclusive hierarchy for comparison.

5. RESULTS

This section discusses the results of our experiments. In the graphs that follow, several techniques are referred as abbreviations. Table V gives a legend for them. The *Inclusive Cache* stands for the baseline.

5.1. Performance Improvement with Inclusion-Sensitive Workloads

Figure 8 shows the number of LLC misses normalized to Inclusive Cache for all inclusion-sensitive workloads. On average, ECI reduces LLC misses by 3%, and QBS

Table I. Inclusion-Sensitive Dual-Core Workloads

Name	Benchmarks
mix-00	perlbench, mcf
mix-01	mcf, calculix
mix-02	hmmer, mcf
mix-03	gromacs, mcf
mix-04	gobmk, mcf
mix-05	gobmk, GemsFDTD
mix-06	gamess, sphinx3
mix-07	namd, xalancbmk

Table II. Inclusion-Insensitive Dual-Core Workloads

Name	Benchmarks
mix-00	calculix, GemsTDTD
mix-01	astar, tonto
mix-02	gcc, mcf
mix-03	gobmk, soplex
mix-04	sphinx3, milc
mix-05	perlbench, libquantum
mix-06	bzip2, hmmer
mix-07	gromacs, h264ref

Table III. 4-Core Workloads

Name	Benchmarks
mix-00	GemsFDTD, h264ref, tonto, lbm
mix-01	gobmk, sphinx3, xalancbmk, mcf
mix-02	namd, bzip2, gcc, mcf
mix-03	perlbench, gcc, namd, zeusmp
mix-04	sphinx3, gamess, zeusmp, perlbench

Table IV. 8-Core Workloads

Name	Benchmarks
mix-00	xalancbmk, tonto, mcf, sphinx3, libquantum, namd, gobmk, soplex
mix-01	perlbench, h264ref, gcc, hmmer, libquantum, soplex, calculix, GemsFDTD
mix-02	zeusmp, calculix, namd, gromacs, xalancbmk, bwaves, gamess, sphinx3
mix-03	omnetpp, h264ref, libquantum, gcc, hmmer, GemsFDTD, calculix, soplex
mix-04	astar, soplex, xalancbmk, GemsFDTD, h264ref, calculix, libquantum, hmmer

Table V. Legend for the Baseline and Various Cache Optimization Techniques

Name	Technique
Inclusive Cache	Inclusive baseline with default LRU policy in each cache
ECI	ECI cache management policy in an inclusive LLC
QBS	QBS cache management policy in an inclusive LLC
Inclusive SDBP	Dead block replacement with SDBP in an inclusive LLC
TMC	TMC cache replacement in an inclusive LLC
Noninclusive LRU	Noninclusive cache with default LRU policy in each cache
Noninclusive SDBP	Dead block replacement with SDBP in a noninclusive LLC

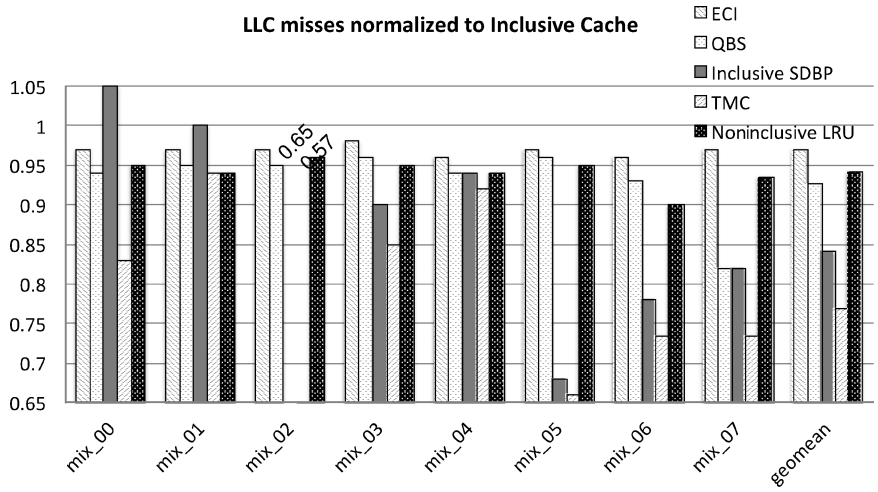


Fig. 8. LLC misses for inclusion-sensitive workloads.

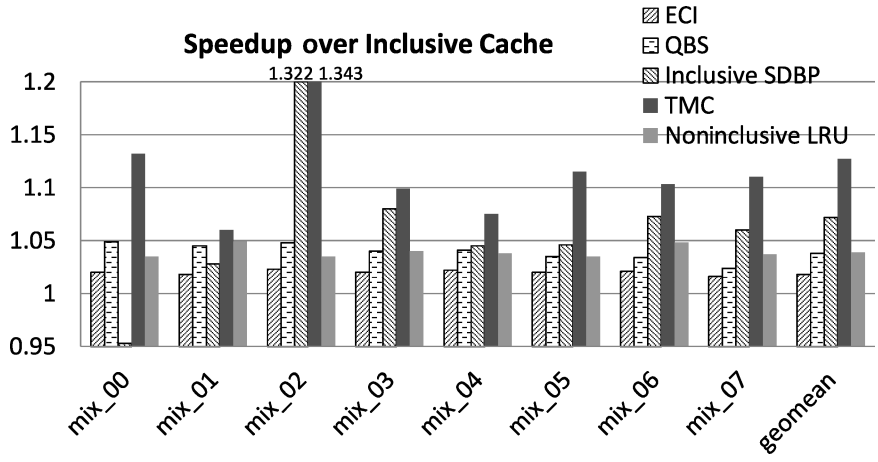


Fig. 9. Performance improvement over inclusive baseline for inclusion-sensitive workloads.

reduces it by 7.4%. Inclusive SDBP reduces LLC misses by 15.9%. For workload *mix_00*, instead of reducing the LLC misses, Inclusive SDBP increases the LLC misses by 5% because Inclusive SDBP is unaware of temporal information of other cache levels, and a predicted dead block in the LLC may still be alive in higher-level caches. TMC reduces the LLC misses for all workloads to generate an average reduction of LLC misses by 23.2%. Noninclusive LRU reduces the LLC misses of the inclusive baseline by 5.9%.

Reducing cache misses leads to improved cache performance. Performance improvement normalized to Inclusive Cache (the IPC of enhanced inclusive cache with certain technique divided by the IPC of Inclusive Cache) is shown in Figure 9. The eight dual-core workloads are inclusion-sensitive workloads with an average performance gap (the IPC of Noninclusive LRU divided by the IPC of Inclusive Cache) of 3.9%.

ECI improves the performance for all workloads and yields an average speedup of 1.8%, whereas the QBS policy improves performance by 3.8%, which is similar to the performance of a noninclusive LRU. Inclusive SDBP, using SDBP technique in inclusive caches, gives a geometric mean speedup by 7.2% over Inclusive Cache. However, for

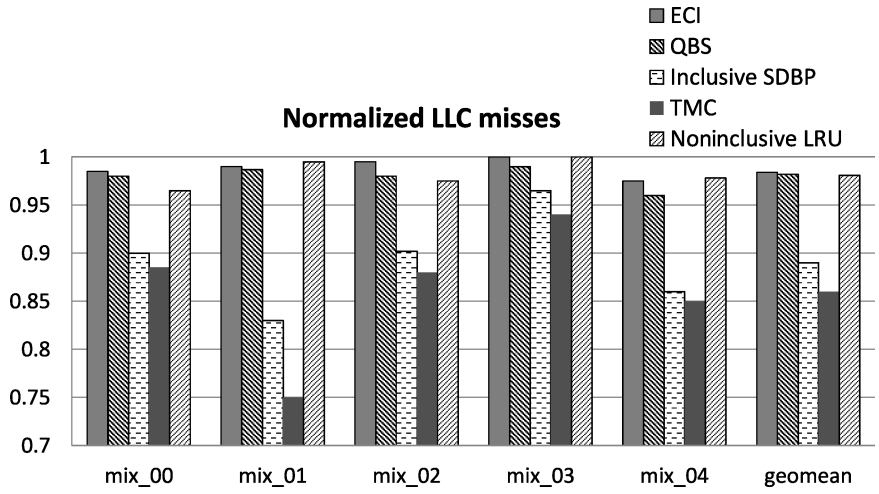


Fig. 10. LLC misses for inclusion-insensitive workloads.

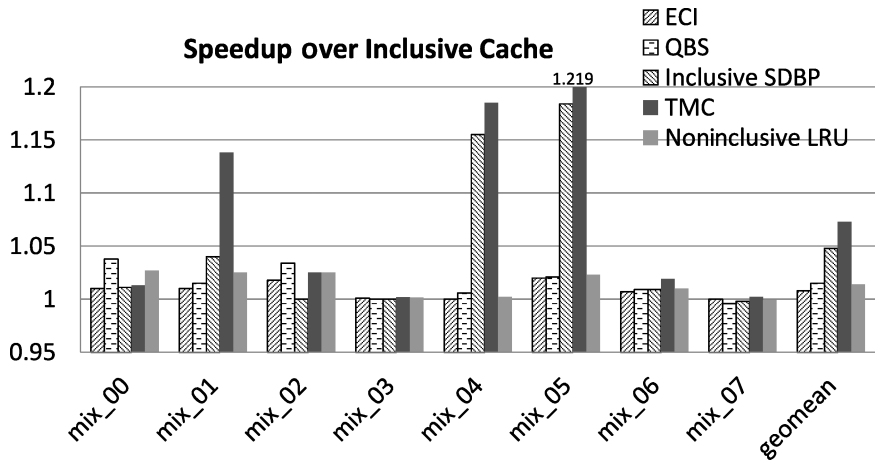


Fig. 11. Performance improvement over the inclusive baseline for inclusion-insensitive workloads.

workload *mix_00*, instead of improving the performance, Inclusive SDBP only achieves 95.3% of the inclusive baseline. TMC improves the performance for all workloads to produce a geometric mean speedup of approximately 12.7%.

5.2. Performance Improvement with Inclusion-Insensitive Workloads

As stated earlier, not all workloads are sensitive to the inclusive property. Before using a technique in inclusive caches, we must guarantee that the technique will not hurt the performance of insensitive workloads. We select eight insensitive workloads. None of the performance gaps of these eight workloads is greater than 3%, and the average performance gap is 1.4%. Figures 10 and 11 show the number of LLC misses and the performance improvement normalized to that of Inclusive Cache for the inclusion-insensitive workloads.

On average, ECI reduces LLC misses by less than 3%. Similar to ECI, QBS reduces it by 2.6%. Inclusive SDBP produces a reduction of LLC misses by less than 12%. For

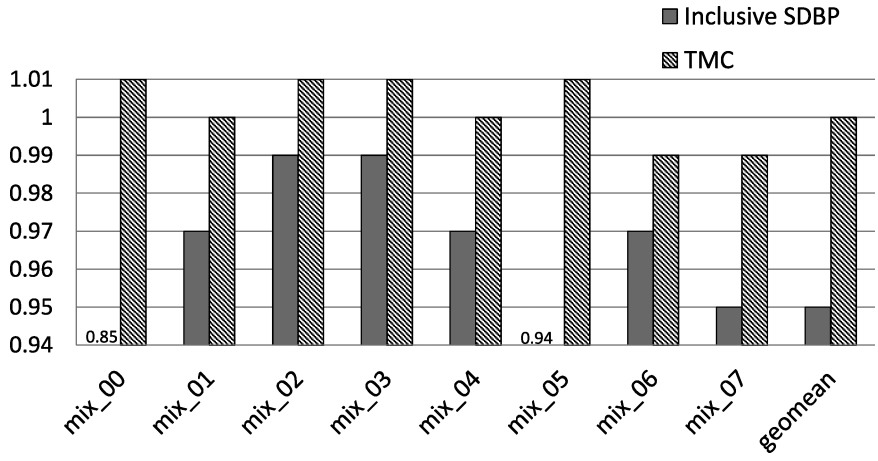


Fig. 12. Speedup normalized to enhance noninclusive cache.

workload *mix_06*, instead of reducing the LLC misses, Inclusive SDBP increases the LLC misses by 2.43%. TMC generates an average reduction of LLC misses by 22%. Noninclusive LRU reduces the LLC misses of the inclusive baseline by 4.7%.

As shown in Figure 11, ECI yields a slight speedup of 0.8% over Inclusive Cache without hurting the performance of any workload. QBS gives a geometric mean speedup of 1.5% over the inclusive baseline, performing comparably to a noninclusive LRU. However, it reduces the performance of workload *mix_07* by less than 1%. Inclusive SDBP achieves performance improvement of 4.8%. However, it also reduces the performance of *mix_07* by less than 1%. TMC improves the performance of all workloads and yields an average speedup of 7.3% over the inclusive baseline.

Based on the results, TMC performs well for both inclusion-sensitive and inclusion-insensitive workloads, whereas QBS and Inclusive SDBP hurt the performance compared to the baseline for some inclusion-insensitive workloads. ECI does not reduce the performance of any inclusion-insensitive workloads, but the performance improvement that it achieves is limited.

5.3. Compared to an Enhanced Noninclusive Cache

By accurately replacing LAL blocks from inclusive caches as early as possible, the performance of inclusive caches not only achieves but outperforms noninclusive caches using default LRU replacement policy. We observed that both Inclusive SDBP and TMC far outperformed Noninclusive LRU. To quantify how much performance improvement Inclusive SDBP and TMC can achieve over a noninclusive LRU, we compare Inclusive SDBP and TMC to Noninclusive SDBP that uses SDBP in a noninclusive LLC and can be treated as an upper bound of an enhanced noninclusive cache.

Figure 12 shows the improved inclusive cache performance normalized to the enhanced noninclusive cache in a dual-core CMP with inclusion-sensitive workloads.

Compared to Noninclusive SDBP, Inclusive SDBP reduces the inclusive cache performance for all eight workloads by 5%, on average. TMC reduces the performance for two workloads and improves four workloads to perform comparably to Noninclusive SDBP.

In conclusion, inclusive caches with TMC outperform noninclusive caches and even comparably to optimized noninclusive caches while maintaining the simplicity of cache coherence.

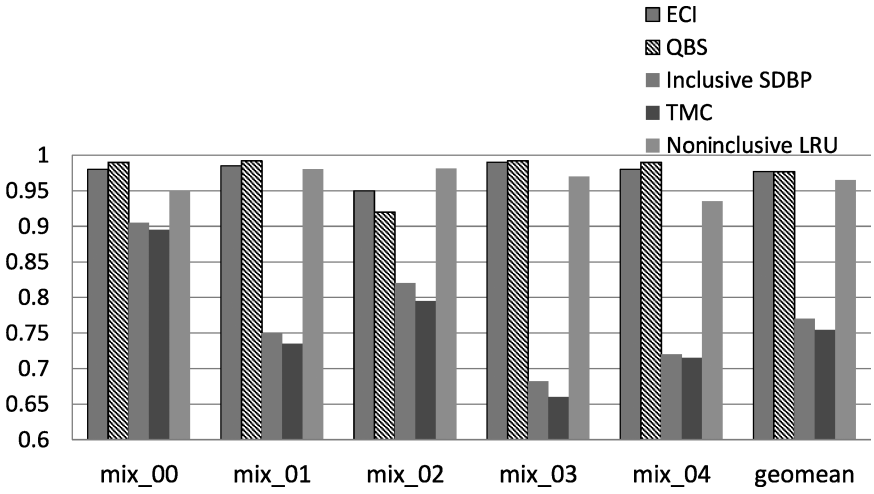


Fig. 13. Normalized LLC misses for 4-core workloads.

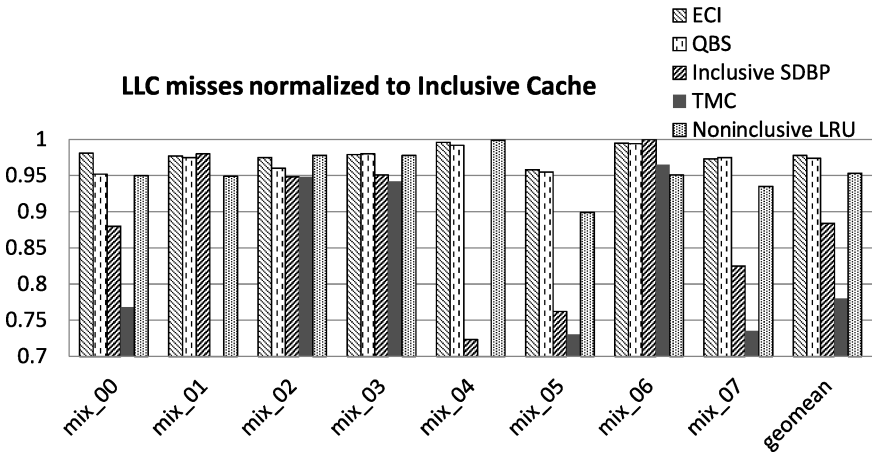


Fig. 14. Normalized LLC misses for 8-core workloads.

5.4. Scalability

To evaluate the scalability to different numbers of cores, we randomly select five groups of inclusion-insensitive 4-core workloads with an average performance gap of 1.7% and five groups of inclusion-insensitive 8-core workloads with an average performance gap of 1.2%. The capacity of the LLC in the experiments is 2MB per core—that is, 8MB for the 4-core configuration and 16MB for the 8-core configuration. Figures 13 and 14 show the normalized LLC misses of each technique on 4-core and 8-core workloads, respectively. Figures 15 and 16 show the performance improvement of each technique on both 4-core and 8-core workloads.

In a 4-core CMP, as shown in Figure 13, ECI reduces the average LLC misses of five workloads by 2.3%. QBS makes slight reduction of LLC misses for four workloads and generates an average reduction of 2.3%. Inclusive SDBP reduces the LLC misses by 23%, on average, whereas TMC yields a reduction of 24.6%. Compared to Inclusive Cache, Noninclusive LRU reduces the LLC misses by 3.5%.

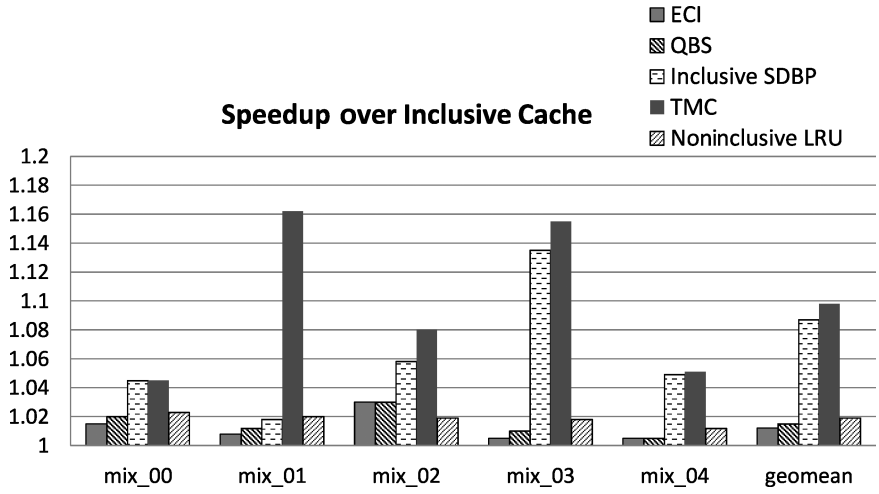


Fig. 15. Performance improvement for 4-core workloads.

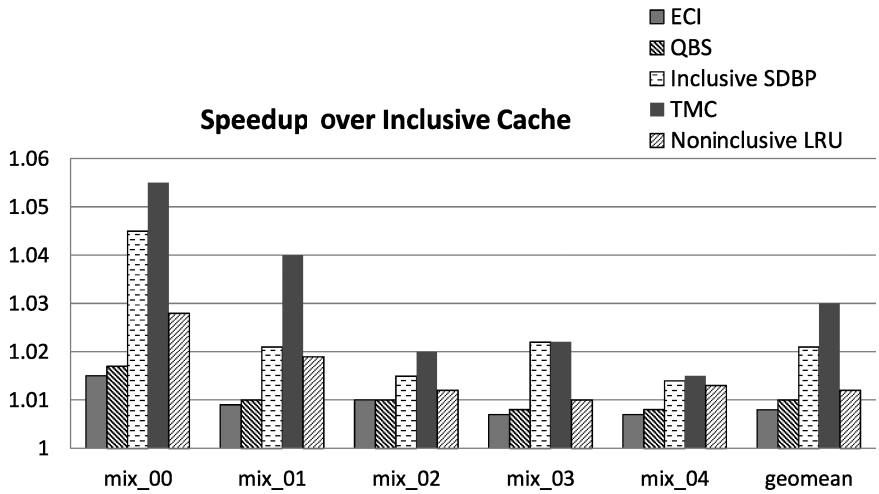


Fig. 16. Performance improvement for 8-core workloads.

The normalized LLC misses for each technique compared to an inclusive cache for 8-core workloads are shown in Figure 14. On average, ECI reduces LLC misses by 1.6%, whereas QBS produces a reduction of 1.8% to the inclusive baseline. Inclusive SDBP reduces the LLC misses by 11%, and TMC generates an average reduction of 14%. Noninclusive LRU reduces the LLC misses by 1.9% compared to the inclusive cache.

As shown in Figure 15, in a 4-core CMP, ECI yields an average speedup of 1.2% over Inclusive Cache, whereas QBS gives a speedup of 1.5%. Inclusive SDBP improves the inclusive cache performance by 8.7%, and TMC improves it by 9.8%. None of the techniques hurts cache performance for any 4-core workloads.

The performance improvement of each technique for 8-core workloads is shown in Figure 16. On average, ECI produces a slight speedup of 0.8% over the baseline, whereas QBS produces an average speedup of 1%. Inclusive SDBP improves the

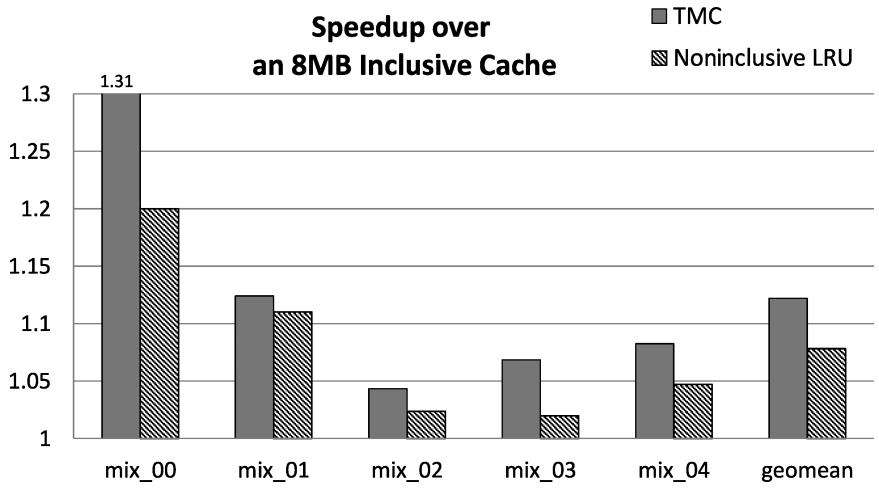


Fig. 17. Performance improvement for 8-core workloads on 8MB LLC.

performance of inclusive caches by 2.1%. TMC achieves up to 5.5% of performance improvement and yields an average speedup of 3% over the inclusive cache, which is still better than a noninclusive LRU that performs better than the baseline by 1.2%.

Note that for 8-core workloads, the performance improvement is lower compared to dual-core or 4-core workloads with any of the techniques. This is because our methodology scales the size of the LLC with the core count. The number of back-invalidated HAH blocks significantly increases when the size of the LLC is not significantly larger than the sum of all higher-level caches [Jaleel et al. 2010; Zahran et al. 2007], which hurts the performance of inclusive caches. In practical design, both Intel and AMD 8-core processors have more modest-sized LLCs [Casazza 2009; AMD 2012]. Previous TLA work also ran experiments on 8-core CMP with 8MB L3 cache. Thus, we ran more experiments to evaluate the scalability of TMC on 8-core CMP with a more practical 8MB LLC. Figure 17 shows the performance improvement of TMC and Noninclusive LRU normalized to 8MB Inclusive Cache. On average, TMC achieves 12.2% of speedup over an 8MB inclusive cache and scales well with the increased number of cores.

In general, all techniques are scalable to different numbers of cores. ECI and QBS help inclusive caches perform similarly to noninclusive caches, whereas Inclusive SDBP and TMC perform better than noninclusive caches.

Based on previous evaluation, inclusive caches with ECI cannot perform as well as noninclusive caches. With QBS, inclusive caches perform similarly to noninclusive ones by paying significantly high communication overhead. Both Inclusive SDBP and TMC help inclusive caches achieve better performance than noninclusive caches with reasonable overhead.

5.5. Detection Accuracy and Coverage

The detection of LAL blocks is not 100% accurate. A misprediction may cause extra delay. Mispredictions come from false positives and false negatives. False positives are more harmful because they detect useful blocks as LAL blocks to cause costly off-chip cache misses. The coverage of a detector is the ratio of LAL detection to all detection. Higher coverage means that the detector can help find more opportunity for the optimization. Figure 18 shows the coverage and false-positive rates of the CTL detector in TMC. On average, it detects a block as a LAL block for 45% of LLC accesses

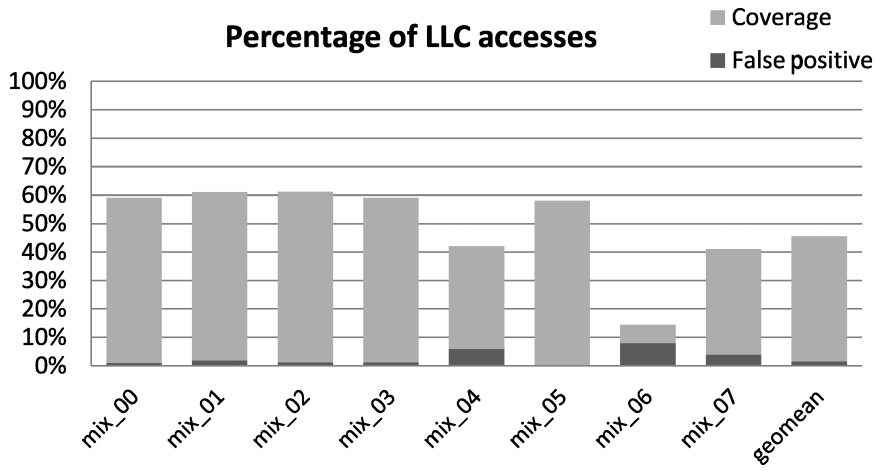


Fig. 18. Coverage and false-positive rate of CTL detector.

Table VI. Dynamic and Leakage Power of TMC (Watts)

	Dynamic Power	Leakage Power
Detector structure	0.078	0.005
Extra metadata	0.008	0.002
Total	0.086	0.007

and has a false-positive rate as low as 1.6%, explaining why it achieves high average speedup.

5.6. Overhead Analysis

5.6.1. Storage Overhead. In this section, we evaluate the storage and power overhead of TMC. The detection table consists of three 4,096-entry tables of 2-bit saturating counters, consuming a total of 3KB in storage. The sampled sets contain 64 sets. Each set has 12 entries consisting of 15-bit partial tags, 15-bit partial PCs, and a 2-bit categorization indicator, consuming 3.375KB of total storage overhead. To indicate the groups of LLC blocks, each LLC block also keeps a 2-bit indicator, consuming 8KB in total for a 2MB LLC. Thus, the CTL detector consumes a total of 14.375KB, which is less than 1% of the capacity of a 2MB LLC in a dual-core CMP.

5.6.2. Power Overhead. The storage overhead costs power overhead. Table VI shows the results of CACTI 6.5 simulations [Muralimanohar et al. 2009] to determine the leakage and dynamic power of the TMC technique. The sampled sets were modeled as the tag array of a cache with as many sets as in the sampler. The detection table was modeled as a tagless RAM with three banks accessed simultaneously. To attribute extra power to cache metadata (2 bits per cache block), we modeled the 2MB LLC both with and without the extra metadata, represented as extra bits in the data array, and report the difference between the two. As shown in Table VI, the dynamic power of the CTL detector is 0.078W and that of the extra metadata is 0.008W. So the total dynamic power of TMC is 0.086W. The leakage power of the CTL detector is 0.005W and that of the extra metadata is 0.002W. Therefore, the total leakage power of TMC is 0.007W. The baseline LLC has a dynamic power of 2.75W and a leakage power of 0.512W. Thus, the TMC technique consumes 3.1% of the dynamic power consumption and 1.4% of the leakage power budget.

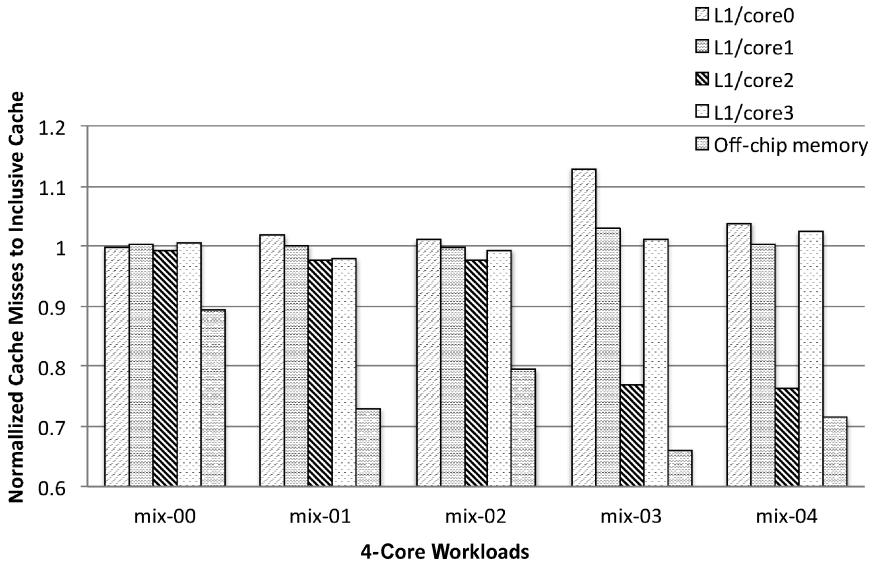


Fig. 19. Normalized communication overhead for 4-core workloads.

5.6.3. Communication Overhead. TMC tends to replace and back-invalidate LAL blocks. When the CTL detector that is warming up lacks knowledge, there might be back-invalidation of HAH blocks that increases the number of L1 misses and on-chip traffic between the L1 cache and the LLC due to the re-fetch of these HAH blocks. After the CTL detector has gone through sufficient training, the P-LAL blocks that the detector invalidates are likely to be LAL blocks, whose invalidation will not cause any future re-fetch and thus will not increase the number of L1 misses. By contrast, with the improved efficiency of the whole inclusive cache hierarchy, cache misses in *each level* will be reduced. Figure 19 shows the normalized number of L1 misses in each core of TMC to that of the inclusive cache in a 4-core CMP. Most of the L1 misses are TMC and similar to that in inclusive caches. In workload *mix-03*, there is an increased on-chip communication overhead of 12.9% between core 0 and the LLC but reduced overhead of 23.2% between core 2 and the LLC compared to the overhead in Inclusive Cache.

There is no re-fetch overhead to off-chip memory compared to the inclusive cache with default LRU replacement policy. The overall communication overhead to off-chip memory is reduced with TMC due to the increased efficiency of the LLC. On average, the off-chip memory accesses are reduced by 24.6%, as shown in Figure 19.

6. RELATED WORK

Previous work introduced several cache management policies to improve inclusive cache performance.

Global replacement policy [Zahran 2007] was designed to use one unified replacement policy to control the replacement of cache blocks in all caches of an inclusive cache hierarchy. This proposal was only evaluated with single-threaded workloads, and the results showed that the global replacement policy sometimes performed worse than the corresponding local replacement policy. Garde et al. [2008] analyzed the performance of global replacement policy by deconstructing the policy with reuse-distance analysis and evaluated it in a multicore inclusive cache hierarchy to show that the performance with global replacement policy was actually limited. Zahran and McKee [2010] proposed to make global cache placement decisions based on access patterns of dierent blocks. This

technique is not designed for inclusive caches because it violates the inclusion property by placing some blocks only into higher-level caches but not the LLC. It can be treated as a noninclusive cache managed by a global placement policy to achieve the capacity of an exclusive cache.

TLA [Jaleel et al. 2010] inclusive cache management policy suite was designed to improve the performance of inclusive caches by reducing the frequency of invalidation of inclusion victims that have high temporal locality in core caches. Successful TLA policies can identify cache blocks that have high temporal locality in core caches and avoid evicting these blocks from the LLC. It consists of three policies: TLH, ECI, and QBS. TLA policies can only identify a limited number of highly referenced blocks in core caches. Moreover, even though the replaced candidates in the LLC do not have high temporal locality in core caches, they may still be live blocks in the LLC whose invalidation will also hurt the cache performance by incurring hundreds of cycles of memory access penalty.

Gaur et al. [2011] proposed a bypass and insertion algorithm for exclusive last-level caches in the LLC to improve the cache performance, but it was only designed for exclusive caches.

Dead block prediction [Lai et al. 2001] is the technique to predict if cache blocks are likely to be dead after certain references. Cache efficiency can be improved if the dead blocks can be replaced with live blocks as early as possible. However, since the dead block predictors cannot acquire temporal locality information, predictions are made locally within the certain cache level. Therefore, this technique cannot solve inclusive problems. There are several dead block prediction techniques introduced in previous work, such as Trace-Based Predictor [Lai et al. 2001], Time-Based Predictor [Hu et al. 2002], Counter-Based Predictor [Kharbutli and Solihin 2008], and SDBP [Khan et al. 2010b]. SDBP [Khan et al. 2010b] is a recently proposed dead block prediction technique that samples a small number of sets to predict dead blocks in the LLC. Compared to other dead block predictors, SDBP uses far less overhead to make predictions with much higher accuracy. Unfortunately, SDBP has no awareness of temporal information, so the predicted dead blocks in the LLC are likely to be high-referenced blocks in core caches. The replacement of these dead blocks in the LLC will cause costly off-chip cache misses, which hurt the inclusive cache performance.

7. CONCLUSION

TMC inclusive cache replacement can efficiently detect LLC blocks that keep low temporal locality in the whole inclusive cache hierarchy as the LLC replacement candidates. It uses sampled local access information and correlated temporal hints from all levels of caches to make accurate detection with minimal storage and communication overhead. This is the first work that introduces correlated temporal information from multiple cache levels. TMC inclusive cache replacement outperforms previous cache management techniques. We have also compared our technique to an optimized noninclusive cache with an intelligent cache management technique and have found that they achieved similar performance improvement. Thus, our technique is comparable to enhanced noninclusive cache management techniques while maintaining inclusion.

This article investigates inclusive cache management with multiprogrammed workloads. In future work, we would extend this work to multithreaded workloads and exploit temporal shared data in the whole cache hierarchy.

REFERENCES

AMD. 2012. AMD FX processors. (2012). Retrieved November 13, 2013, from http://www.amd.com/us/products/desktop/processors/amdfx/.

BAER, J.-L. AND WANG, W.-H. 1988. *On the Inclusion Properties for Multi-Level Cache Hierarchies*. Vol. 16. IEEE Computer Society Press.

CASAZZA, J. 2009. Intel Core i7-800 processor series and the Intel Core i5-700 processor series based on Intel microarchitecture (Nehalem). White paper, Intel Corp.

CHEN, X., YANG, Y., GOPALAKRISHNAN, G., AND CHOU, C. T. 2006. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Formal Methods in Computer Aided Design, 2006 (FMCAD'06)*. IEEE, Los Alamitos, CA, 81–88.

LAI, A.-C. AND FALSAFI, B. 2000. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 139–148.

LAI, A.-C., FIDE, C., AND FALSAFI, B. 2001. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*. 144–154.

GARDE, R. V., SUBRAMANIAM, S., AND LOH, G. H. 2008. Deconstructing the inefficacy of global cache replacement policies. In *Proceedings of the 7th Workshop on Duplicating, Deconstructing, and Debunking (WDDD'08)*.

GAUR, J., CHAUDHURI, M., AND SUBRAMONEY, S. 2011. Bypass and insertion algorithms for exclusive last-level caches. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ACM, New York, NY, 81–92.

HENNING, J. L. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Architecture News 34*, 4, 1–17.

HU, Z., KAXIRAS, S., AND MARTONOSI, M. 2002. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*. IEEE, Los Alamitos, CA, 209–220. http://dl.acm.org/citation.cfm?id=545215.545239.

JALEEL, A., BORCH, E., BHANDARU, M., STEELY JR., S. C., AND EMER, J. 2010. Achieving non-inclusive cache performance with inclusive caches: Temporal Locality Aware (TLA) cache management policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE, Los Alamitos, CA, 151–162. http://dx.doi.org/10.1109/MICRO.2010.52.

JOUPPI, N. P. AND WILTON, S. J. E. 1994. Tradeoffs in two-level on-chip caching. *ACM SIGARCH Comput. Architecture News 22*, 2, 34–45.

KHAN, S. M., JIMÉNEZ, D. A., BURGER, D., AND FALSAFI, B. 2010a. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, 489–500. http://dx.doi.org/10.1145/1854273.1854333.

KHAN, S. M., TIAN, Y., AND JIMENEZ, D. A. 2010b. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE, Los Alamitos, CA, 175–186. http://dx.doi.org/10.1109/MICRO.2010.24.

KHARBUTLI, M. AND SOLIHIN, Y. 2008. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput. 57*, 4, 433–447. http://dx.doi.org/10.1109/TC.2007.70816.

KURD, N. A., BHAMIDIPATI, S., MOZAK, C., MILLER, J. L., WILSON, T. M., NEMANI, M., AND CHOWDHURY, M. 2010. Westmere: A family of 32nm IA processors. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'10)*. IEEE, Los Alamitos, CA, 96–97.

LEBECK, A. R. AND WOOD, D. A. 1995. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 48–59.

MICHAUD, P., SEZNEC, A., AND UHLIG, R. 1997. Trading conflict and capacity aliasing in conditional branch predictors. *ACM SIGARCH Comput. Architect. News 25*, 2, 292–303. http://dx.doi.org/10.1145/384286.264211.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. P. 2009. CACTI 6.0: A tool to model large caches. Technical report HPL-2009-85. HP Laboratories.

NAIR, R. 1995. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 15–23.

PATEL, A., AFRAM, F., CHEN, S., AND GHOSE, K. 2011. MARSSx86: A full system simulator for x86 CPUs. In *Proceedings of the 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC'11)*. 1050–1055.

QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY, S. C., AND EMER, J. 2007. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Comput. Architecture News, 35*, 381–391.

QURESHI, M. K. AND PATT, Y. N. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'39)*. IEEE, Los Alamitos, CA, 423–432. http://dx.doi.org/10.1109/MICRO.2006.49.

SANCHEZ, D. AND KOZYRAKIS, C. 2010. The zcache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Los Alamitos, CA, 187–198.

SEZNEC, A. 1993. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. 169–178.

SOMOGYI, S., WENISCH, T. F., HARDAVELLAS, N., KIM, J., AILAMAKI, A., AND FALSAFI, B. 2004. Memory coherence activity prediction in commercial workloads. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture (WMPI'04)*. ACM, New York, NY, 37–45. http://dx.doi.org/10.1145/1054943.1054949.

WENDEL, D. F., KALLA, R., WARNOCK, J. D., CARGNONI, R., CHU, S. G., CLABES, J. G., DREPS, D., HRUSECKY, D., FRIEDRICH, J., ISLAM, S., ET AL. 2011. POWER7, a highly parallel, scalable multi-core high end server processor. *IEEE J. Solid-State Circuits 46*, 1, 145–161.

WU, C.-J., JALEEL, A., HASENPLAUGH, W., MARTONOSI JR., M., STEELY, S. C., AND EMER, J. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44'11)*. ACM, New York, NY, 430–441. http://dx.doi.org/10.1145/2155620.2155671.

ZAHRAN, M. 2007. Cache replacement policy revisited. In *Proceedings of the 6th Workshop on Duplicating, Deconstructing, and Debunking (ISCA)*.

ZAHRAN, M., ALBAYRAKTAROGLU, K., AND FRANKLIN, M. 2007. Non-inclusion property in multi-level caches revisited. *Int. J. Comput. Appl. 14*, 2, 99.

ZAHRAN, M. AND MCKEE, S. A. 2010. Global management of cache hierarchies. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF'10)*. ACM, New York, NY, 131–140. http://dx.doi.org/10.1145/1787275.1787315.

ZHENG, Y., DAVIS, B. T., AND JORDAN, M. 2004. Performance evaluation of exclusive cache hierarchies. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Los Alamitos, CA, 89–96. http://dl.acm.org/citation.cfm?id=1153925.1154591.

Received June 2013; revised October 2013; accepted November 2013