Fine Grain Cache Partitioning using Per-Instruction Working Blocks

Jason Jong Kyu Park
University of Michigan
Ann Arbor, MI, U.S.A.
jasonjk@umich.edu

Yongjun Park
Hongik University
Seoul, Korea
yongjun.park@hongik.ac.kr

Scott Mahlke
University of Michigan
Ann Arbor, MI, U.S.A.
mahlke@umich.edu

***Abstract*—A traditional least-recently used (LRU) cache replacement policy fails to achieve the performance of the optimal replacement policy when cache blocks with diverse reuse characteristics interfere with each other. When multiple applications share a cache, it is often partitioned among the applications because cache blocks show similar reuse characteristics within each application. In this paper, we extend the idea to a single application by viewing a cache as a shared resource between individual memory instructions.**

To that end, we propose *Instruction-based LRU* (*ILRU*), a fine grain cache partitioning that way-partitions individual cache sets based on per-instruction working blocks, which are cache blocks required by an instruction to satisfy all the reuses within a set. In *ILRU*, a memory instruction steals a block from another only when it requires more blocks than it currently has. Otherwise, a memory instruction victimizes among the cache blocks inserted by itself. Experiments show that *ILRU* can improve the cache performance in all levels of cache, reducing the number of misses by an average of 7.0% for L1, 9.1% for L2, and 8.7% for L3, which results in a geometric mean performance improvement of 5.3%. *ILRU* for a three-level cache hierarchy imposes a modest 1.3% storage overhead over the total cache size.

***Keywords*-Cache Replacement Policy; Fine Grain Cache Partitioning;**

I. Introduction

While a traditional least-recently used (LRU) replacement policy in a cache closely approximates Belady's optimal replacement algorithm [1], it often fails when cache blocks with diverse reuse characteristics interfere with each other. Past research on a shared cache has shown that a thread-aware shared cache management is effective in providing increased performance [2], [3], [4], [5]. These management schemes partition the shared cache among the threads or applications to reduce inter-thread interference because blocks show similar reuse characteristics within each application.

We apply the same strategy to view the cache as a shared resource between individual memory instructions even if only a single application is running. Because cache blocks accessed or inserted by the same instruction show similar reuse characteristics, partitioning a cache among the individual instructions can reduce interference between them. In that sense, a cache replacement policy becomes a resource management policy, where a memory instruction that inserts a block has to choose between taking more blocks by stealing one from another instruction or maintaining the same blocks by evicting one owned by itself.

A traditional least-recently used (LRU) policy assumes that the miss frequency of an instruction correlates to the number of the cache blocks to give to that instruction. Prior works on bypass have shown that the cache misses are not the right measure to choose the number of cache blocks for an instruction, especially in the last level cache (LLC) [6], [7], [8], [9]. In these works, many of the cache misses are bypassed to avoid the unnecessary consumption of blocks. However, their solutions do not extend to upper level caches (closer to the CPU). They assume that cache blocks with no temporal locality are abundant, which is correct only in the LLC due to the filtered memory references.

To improve the cache performance in every level of cache with a single, uniform replacement policy, we propose *Instruction-based LRU* (*ILRU*), that partitions individual sets of the cache based on the per-instruction working blocks [1]. When a memory instruction experiences a cache miss, its number of working blocks is compared to its currently occupying blocks in that set. When the instruction already has enough blocks, it victimizes among those inserted by itself, so that it never occupies more than necessary. It steals a block from another instruction when it is occupying fewer than its working blocks. The notion of working blocks naturally subsumes the behavior of bypass when an instruction requires zero blocks. Because *ILRU* chooses to evict the LRU block when victimizing among the blocks occupied by itself, it behaves exactly the same as the traditional LRU when an instruction requires all the blocks in a set.

Prior shared cache studies have employed auxiliary tag directories or shadow tags per core to estimate the partition size for each core [5], [10]. However, the same approach cannot be utilized for *ILRU* because there are too many memory instructions compared to the number of cores. Instead, *ILRU* introduces the concept of a *hit position* to allow in-place prediction with the cache behavior under *ILRU*, and uses a single shared shadow tag per set among

[1] We intentionally avoid the term "*working set*" and use "*working blocks*" because working set traditionally means the amount of memory in *bytes* required by a process in a given time interval *regardless of the cache sets*. Working blocks are *the number of cache blocks* required by an instruction to support all the reuses in a given time interval *within a set*.

the instructions. For each instruction, a hit position measures how many cache blocks (including the hit block) *inserted by the same instruction* were ahead of the hit block in the LRU chain. The instruction must have occupied at least the same number of blocks in that set as the hit position to satisfy that reuse. If the instruction had fewer blocks in the set, the current hit block would have been evicted resulting in a miss. To have small, shared shadow tags, only the bypassed blocks and the blocks evicted from non-LRU positions are inserted into the shadow tags. If an entry in the shadow tags hit, the instruction that inserted the entry is predicting lower number of working blocks than the actual number.

To reduce the overall hardware storage overhead, *ILRU* first utilizes set sampling [11]. In set sampling, only a few sampling sets are monitored to measure the hit position and the rest of the sets follows the decision made by the sampling sets. Because shadow tags are only required by the sampling sets, the overhead of shadow tags can be significantly reduced. However, identifying the inserting instruction for every cache block in non-sampling sets still imposes large storage overhead. *ILRU* utilizes a hot instruction table, which keeps track of the last N instructions that recently inserted a block in the cache. The hot instruction table allows to precisely track these N instructions, but other instructions will alias with those N instructions. In lower level caches, the number of inserting instructions is small enough that the alias problem has negligible impact on performance.

This paper makes the following contributions:

- We show the opportunity of viewing a cache as a shared resource between individual memory instructions. The notion of working blocks for individual instructions subsumes the bypassing when zero blocks are required, and the traditional LRU replacement when all the blocks are required.
- We propose *Instruction-based LRU*, a fine grain cache partitioning strategy, that utilizes per-instruction working blocks. *ILRU* improves cache performance at every level of the cache hierarchy.
- We introduce the concept of a *hit position* to allow in-place prediction for the per-instruction working blocks. *ILRU* employs shared shadow tags to monitor whether the predicted number of working blocks is correct.
- We implement *ILRU* with low hardware cost using set sampling and a hot instruction table. The techniques rely on sampling only a few sets to approximate the whole cache behavior and capturing only the last N instructions that inserted a block in lower level caches.

II. Motivation

Per-instruction working blocks are the number of cache blocks required by an instruction to satisfy all the reuses during a block's lifetime within a set. In this section, we first show the per-instruction working blocks at an intuitive level with several examples. We then observe the additional opportunities of identifying per-instruction working blocks over an ideal bypass technique that can only benefit when there are zero working blocks. We finally derive the concept of a hit position, which predicts the per-instruction working blocks for the lifetime of a cache block.

A. Per-Instruction Working Blocks

Figure 1 illustrates the number of working blocks for instruction A, and the cache behavior under LRU on top and ideal working block replacement (IWBR) on bottom. IWBR perfectly knows per-instruction working blocks. The figure shows memory references, and the resulting LRU chain of a single set under LRU and IWBR with a hit/miss. The sequence of memory references are identical for the same working blocks in both LRU and IWBR. A white box indicates an access or insertion of a block by instruction A, whereas a grey box indicates an access or insertion by other instructions. A tag is given inside the box to distinguish the accesses. Examples show when the number of working blocks for instruction A are zero, one, and two, respectively.

In Figure 1 (a) and (d), the block inserted by instruction A is never re-referenced. In this case, instruction A requires no blocks, and the block can be bypassed. Moreover, bypassing block 4 results in cache hits for the references to blocks 2 and 1 with IWBR. Instruction A may not require cache blocks for various reasons, e.g., it may be an instruction in a loop with a stride larger than the block size with no re-references from other instructions. Iterating over large linked lists can result in similar behavior. In lower level caches, there are more such cases because reuses are filtered by upper level caches. The notion of per-instruction working blocks can detect bypass candidates.

Figure 1 (b) and (e) depict a simple example, where an instruction with a small stride is referencing an array within a loop. The block inserted by instruction A experiences a burst of accesses from instruction A. In the example, the size of the array is large enough so that the same set experiences another burst of accesses from instruction A. Here, all the accesses to the blocks inserted by instruction A are satisfied even if instruction A only had one block in the set, e.g., if the block 8 replaced block 4, which was inserted by instruction A. The number of working blocks for instruction A thus equals to one. Subsequent references to blocks 1 and 0 are hits under IWBR, and misses under LRU.

Figure 1 (c) and (f) illustrate an example, where the number of working blocks for instruction A is two. When a multi-row filter is applied in 2D image, re-references can occur from other instructions in some distant time because a pixel is re-referenced after traversing through the entire x-axis of the image. In this case, the first block inserted by instruction A is re-referenced after the insertion of the second block by instruction A. Instruction A requires two working blocks to support all reuses. Under IWBR, the last reference to block 0 is a hit, which LRU fails to capture.

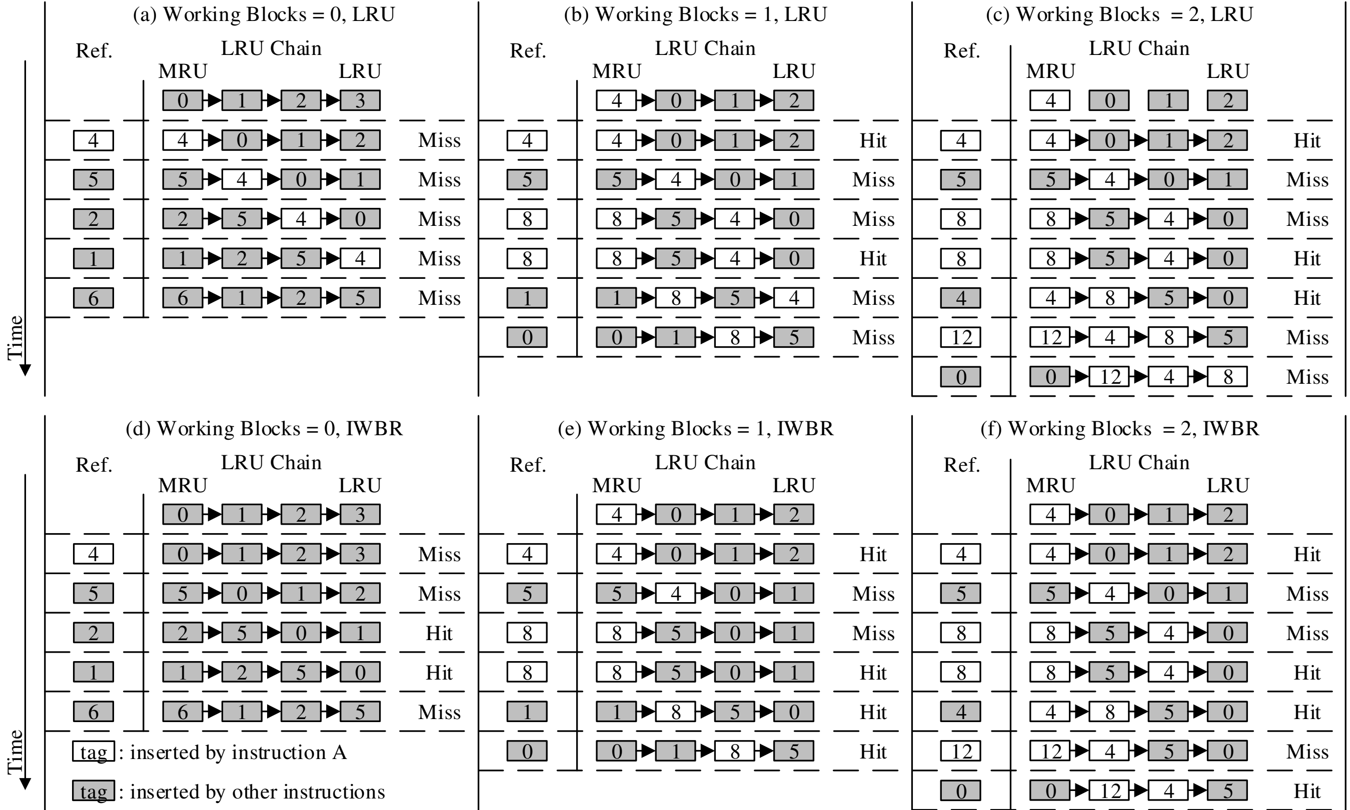


Figure 1. The cache behavior when instruction A has working blocks of (a) zero under LRU, (b) one under LRU, (c) two under LRU, (d) zero under ideal working block replacement (IWBR), (e) one under IWBR, and (f) two under IWBR. Note that reference streams are identical for the same working blocks in both LRU and IWBR.

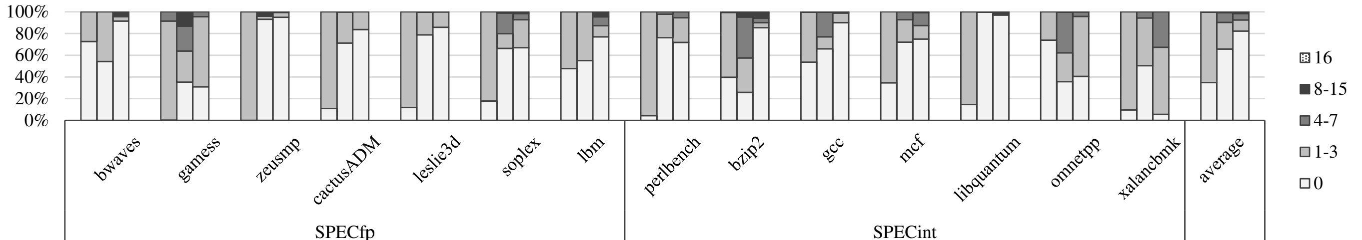


Figure 2. Distribution of the number of working blocks for instructions. The number of working blocks in the range of 1-3, 1-7, 1-15 are additional opportunities over ideal bypassing for L1, L2, and L3, respectively.

At maximum, an instruction may require all the blocks in a set. In such case, evicting a cache block without the notion of the per-instruction working blocks is indistinguishable from IWBR. The maximum case rarely occurs because it is likely that contending memory instructions insert cache blocks into a set before an instruction fully occupies the set.

B. Opportunity

Figure 2 shows the distribution of the number of working blocks for dynamic instructions in a memory-intensive subset of SPEC CPU2006 benchmark suite. We discuss how the subset is selected in Section V. We assume that L1, L2, and L3 caches are 32kB 4-way, 256kB 8-way, and 2MB 16-way associative, respectively. A zero working block indicates a bypass opportunity. Any other number of working blocks below the associativity of a cache indicates additional opportunities for viewing a cache as a shared resource between individual instructions over ideal bypassing. The figure illustrates that these additional opportunities between zero and the associativity are 65%, 33%, and 18% in L1, L2, and L3, respectively. Thus, for example, 65% of the L1 evictions offer the opportunity to improve hit rate by selecting a non-LRU block to replace.

C. Hit Position

To facilitate working block guided replacement, the remaining question is how to measure or predict the per-instruction working blocks. We can discover insights on how

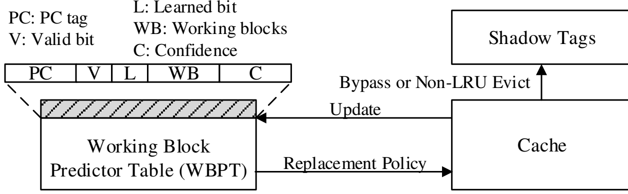


Figure 3. Architectural overview of *ILRU*.

to predict per-instruction working blocks from Figure 1. If we only look at the *cache blocks inserted by the same instruction* as the hit block, the number of working blocks exactly matches with the *hit position* (the distance from the MRU block). The intuition behind the hit position is simple. Suppose the instruction was forced to have fewer number of blocks than the current hit position. The instruction would have replaced the hit block with another block while handling previous cache misses, which would convert the current hit to a miss. For example, if the instruction A in Figure 1 (f) had one working block, block 8 would have replaced block 4. The re-reference to block 4 after insertion of block 8 would become a cache miss. Therefore, an instruction requires at least as many blocks as the hit position. If multiple evicting blocks have smaller hit position compared to the currently predicted working blocks, it is safe to assume that the working blocks have decreased due to the program phase change. Figure 1 (e) and (f) also show that the prediction needs an initial learning period. At the time of the access to block 8, the sequence of the references are the same, but the number of working blocks is different. The prediction can be correctly made only after we have seen all the reuses until the first block's eviction. During the initial learning period, instructions are assumed to require all the cache blocks, e.g., the traditional LRU.

III. ARCHITECTURE

ILRU partitions the cache using per-instruction working blocks. To achieve such partitioning, *ILRU* maintains a working block predictor table (WBPT) to keep track of the predictions for the per-instruction working blocks. Because per-instruction working blocks depend on the underlying cache configuration, a WBPT exists for each level of cache.

Figure 3 illustrates the architectural components of *ILRU* in a single level of the cache hierarchy. Arrows indicate interactions between architectural components. In addition, metadata in a WBPT entry is listed in detail. The WBPT is indexed by PC. Like other previous works based on PC-indexed tables [12], [13], we found out that using a 15-bit partial PC is sufficient to achieve the same performance as using the full PC. The partial PC is stored in the load-store queue and is transferred with the memory reference through all levels of cache. As discussed in Section II-C, a WBPT entry also has a learned bit, which indicates whether the initial learning period has finished or not. The number of working blocks is $(log_2 Associativity + 1)$ bits wide, and the confidence is two bit saturating counters. Shadow tags mimic cache behavior when *ILRU* is not applied so that *ILRU* can correct mispredictions. The additional metadata for each tag in the cache and shadow tags is shown in Table I and is discussed in Section IV-B.

A. Cache Lookup

Cache lookup is unmodified from a traditional cache because *ILRU* only modifies the replacement policy. LRU stack is updated as in the traditional LRU replacement. Updating the WBPT or accessing shadow tags are not on the critical path. They can be delayed one cycle after the cache access if there exist any timing problems.

B. Replacement Policy

Figure 4 illustrates how replacement works with *ILRU*. The replacement candidate decision is based on the learned bit, the number of working blocks, and the number of blocks occupied by the inserting instruction in the set. Note that the confidence does not affect the replacement policy. When the learned bit is not set, the instruction is in the initial learning period. The instruction is assumed to require all the blocks in the set, which is essentially the same as the traditional LRU replacement as shown in Figure 4 (a). When the learned bit is set, the number of working blocks and the number of occupying blocks for the inserting instruction are compared. In Figure 4 (b), instruction B already occupies three blocks in the set, which is equal to or more than the number of working blocks. In this case, *ILRU* victimizes the LRU block among the blocks inserted by instruction B, which is indicated by the dashed circle. However, if an instruction occupies fewer than its working blocks as shown in Figure 4 (c), *ILRU* victimizes the LRU block to acquire more blocks. Note that instruction B loses its working block to instruction A due to the contention; however, the predictor will adjust the working blocks afterwards to resolve the contention. A block is bypassed when an instruction requires zero blocks.

C. Occupying Blocks

Occupying blocks is detected similarly to the prior works on a shared cache, where they check whether a block belongs to cores or threads [2], [3], [5], [10], [14]. The detailed calculation is performed in two steps. First, 15-bit partial PC comparison is performed, which is faster than tag comparison. Second, results of the comparisons are added to calculate occupying blocks. The same method is used to calculate hit position in Section IV-A, where the second step is subtraction instead of addition because LRU status already contains information on the hit position without taking instructions into account and which blocks are ahead. Note that choosing a victim can take multiple cycles, which only has to be faster than accessing the lower level cache.

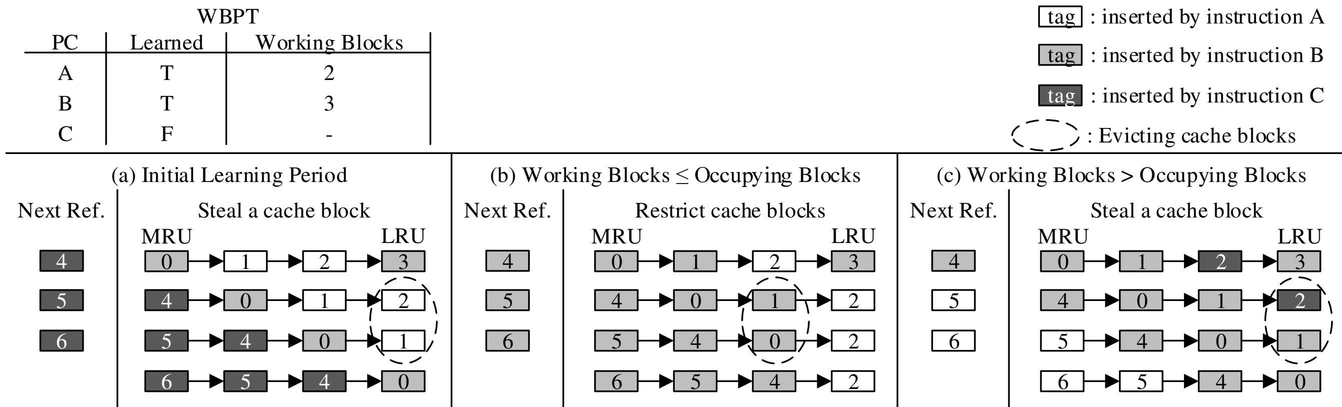


Figure 4. *ILRU* replacement policy: (a) instruction C steals a block from another instruction in the initial learning period, (b) instruction B restricts its blocks by evicting one of the occupying blocks, and (c) instruction A steals a block from another instruction because it occupies fewer than its working blocks.

D. Shadow Tags

In-place prediction of *ILRU* cannot react when the prediction should be corrected. To address this issue, we utilize shadow tags [5], [10] to simulate the cache behavior as if *ILRU* was not applied. Shadow tags are essentially the same as tag arrays in the cache except for the metadata.

In *ILRU*, shadow tags store the bypassed blocks and non-LRU evictions for all memory instructions. Shadow tags have a special metadata called life, which is the same as its expected LRU position in the corresponding set before eviction. The life monotonically decreases afterwards as the corresponding set is accessed. When the life becomes zero, the shadow tag is invalidated. Each shadow tag stores 16-bit partial address tags, 15-bit partial PC, and $log_2 Associativity$ bit life. A cache block still has to be fetched from lower level caches even if it hits in the shadow tags. When a shadow tag hits, it is invalidated because its behavior can be tracked in the cache again.

IV. Predictor

ILRU predicts per-instruction working blocks using a hit position. A hit position measures the position of the hit block *considering only the blocks inserted by the same instruction*. It corresponds to how many blocks the instruction had to maintain for the current hit to take place. Because *ILRU* uses hit position for the prediction, the contending memory instructions are taken into account automatically. If multiple instructions require blocks that exceed the associativity in total, they will compete with each other. However, an instruction with earlier reuses will eventually keep its working blocks because only it continues to observe hits, while the others lose their working blocks as they observe misses due to contention.

ILRU allocates a WBPT entry to an instruction when cache misses. The initial learning period for the WBPT entry ends when a block inserted by the instruction is evicted for the first time.

A. Updating the WBPT

To understand high level behavior on the measurement of hit position, we visualize an LRU chain with blocks that have same instruction identifier as the hit block when there is a cache or shadow tag hit. Figure 5 (a) illustrates how the WBPT is updated when the cache hits. The LRU chain of the instruction, which inserted the hit block, is shown to clarify the hit position. The number of working blocks is increased to the hit position only when the hit position is larger than the current prediction. The confidence metadata is set to the maximum value if the number of working blocks is updated. The maximal hit position (MHP) metadata for the hit block is updated accordingly as well. To avoid updating the WBPT for every cache hit, the WBPT is only updated when the hit position is greater than or equal to its MHP. We did not show this comparison in the figure for the sake of brevity.

Figure 5 (b) illustrates the WBPT updates when a shadow tag hits. When a shadow tag hits, the LRU chain of the instruction is constructed assuming that the shadow tag is in its expected LRU position, which corresponds to the life metadata. Otherwise, shadow tags are completely ignored when forming the LRU chain of an instruction. As the last line of Figure 5 (b) shows, the inserted block is treated as if it was inserted by instruction A, not instruction C, because a shadow tag indicates a misprediction for instruction A. In fact, it can be thought of as copying the metadata from the shadow tag to the inserted block.

Figure 5 (c) depicts how the WBPT is updated when a block is evicted. The WBPT is only updated when the evicted block's MHP is smaller than the current prediction. *ILRU* decreases the confidence first, and decreases the working blocks only when the confidence is zero. The MHP metadata for a newly inserted block is reset to zero.

B. Reducing Hardware Overhead

Prior works [11], [5], [10], [7] have shown that sampling a few sets in the LLC can closely approximate the behavior

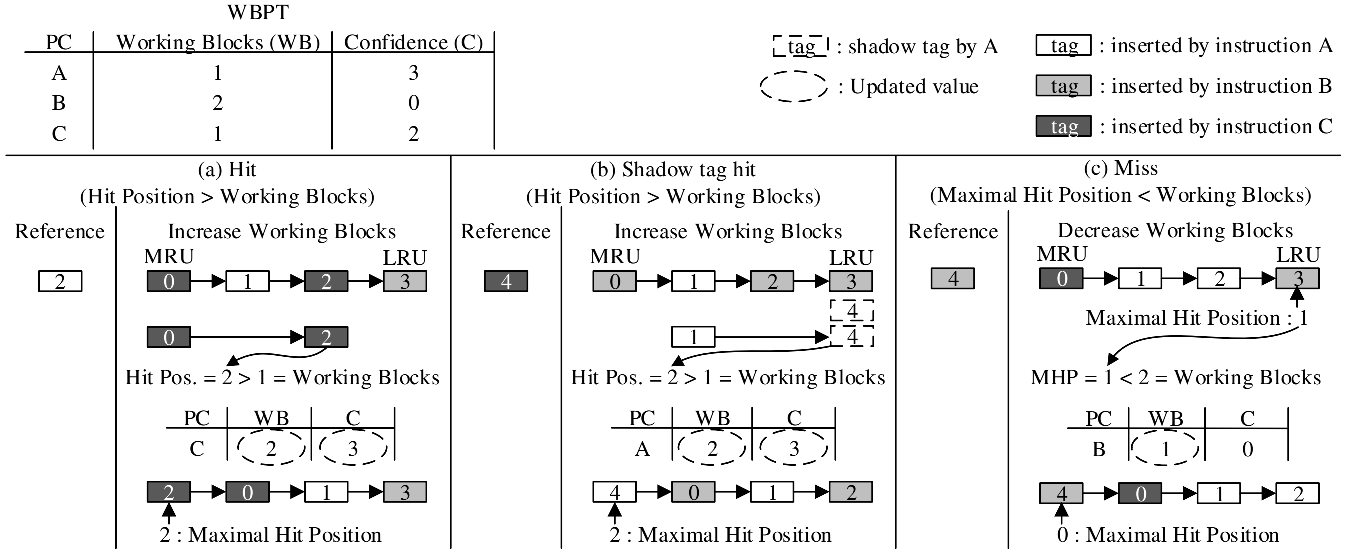


Figure 5. Updating the predictor: increasing the predicted working blocks (a) when cache hits and the hit position is larger than the current prediction; (b) when a shadow tag hits and the hit position is larger than the current prediction; and (c) decreasing the predicted working blocks when the maximal hit position is smaller than the current prediction when cache misses.

in the entire LLC. Similarly, we use set sampling to reduce the hardware overhead of the shadow tags in *ILRU*. We refer to the sets that update the working blocks as sampling sets, and the others that follow the prediction as non-sampling sets. Sampling sets have corresponding shadow tags, while non-sampling sets do not.

Sampling is useful when memory access patterns are regular across the sets so that the whole cache behavior can be approximated with few sampling sets. It is likely that the memory references are more regular in lower level caches because irregularities are filtered by upper level caches. Like prior works, we have 32 sampling sets out of 2048 sets in the LLC. We sample all the sets in L1, and 128 sets out of 512 sets in L2. To choose the sampling sets, we use the static assignment proposed in dynamic set sampling [11]. Furthermore, *ILRU* reverses the assignment between the L2 and the LLC so that the LLC can capture and compensate for the patterns that were not sampled by the L2 cache. Through experiments, we have found out that shadow tags in the sampling sets can be sized to have only half of the associativity of the set that they are attached to with negligible impact on the performance.

To reduce the hardware overhead of identifying instructions, *ILRU* utilizes a hot instruction table, which keeps track of the last N instructions that inserted a cache block. A hot instruction table with 16 entries can distinguish instructions with a 4-bit ID. A hot instruction table is fully-associative and maintained in LRU order. When an instruction inserts a block into the cache and it is not resident in the hot instruction table, it replaces the LRU instruction in the hot instruction table and uses that ID. The instruction aliases with the replaced instruction. We discuss the impact of aliasing in Section V-C.

Table I. EXTRA METADATA IN TAG ARRAYS

Set	Field	Bitwidth	Description
Sampling	PC	15 bits	Instruction identifier
	MHP	$log_2 Assoc + 1$ bits	Maximal hit position
Non-sampling	ID	4-5 bits	Instruction identifier
Shadow tags	Tag	16 bits	Partial address tag
	PC	15 bits	Instruction identifier
	Life	$log_2 Assoc + 1$ bits	Expected LRU position

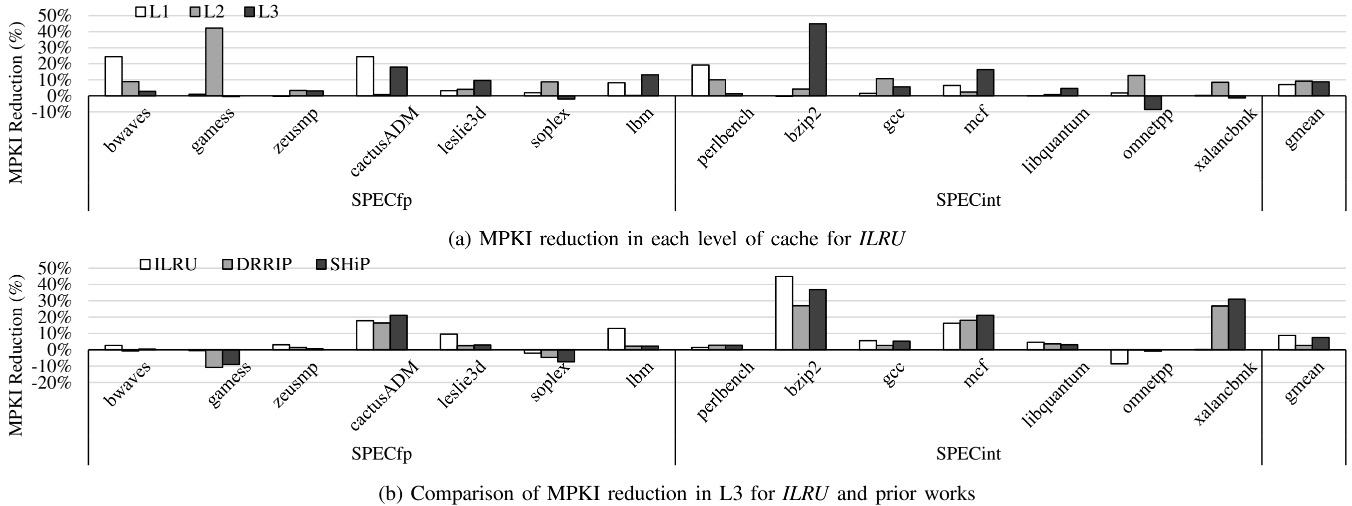
The metadata for each tag entry are summarized in Table I. In non-sampling sets, the overhead of the additional metadata is less than one-third of that in the sampling sets.

C. ILRU *on a Shared LLC*

ILRU partitions a cache at an instruction granularity. In a shared LLC, *ILRU* continues to partition at the instruction granularity by differentiating the instructions from different threads. When the partial PC is sent to the shared LLC to identify an instruction, the logical thread ID is prepended to the partial PC. Once the instructions are differentiated, the predictor and the replacement policy will adapt accordingly because the prediction itself considers all the contending memory instructions.

V. RESULTS

The Pin-based [15] Graphite [16] simulator is used to evaluate *ILRU*. An in-order core with an out-of-order memory system is modeled. The processor has a 8-entry load buffer as well as a 8-entry store buffer. A three-level cache system is used: the L1 instruction cache uses traditional LRU replacement, and the unified L2 and L3 cache are non-inclusive. Unless specified otherwise, a hot instruction table is 32-entry for L2 and 16-entry for L3, and the WBPT is



(a) MPKI reduction in each level of cache for *ILRU*

(b) Comparison of MPKI reduction in L3 for *ILRU* and prior works

Figure 6. MPKI reduction over LRU.

Table II. SYSTEM CONFIGURATION

Architecture	Parameters
Processor	In-order core with out-of-order memory 8-entry load buffer, 8-entry store buffer
L1 I-Cache	32kB, 4-way, 64B blocks 1 cycle access
L1 D-Cache	32kB, 4-way, 64B blocks 1 cycle access 512-entry, 4-way WBPT
L2 Unified Cache	256kB, 8-way, 64B blocks 10 cycle access 512-entry, 4-way WBPT 32-entry Hot instruction table
L3 Unified Cache	2MB, 16-way, 64B blocks 30 cycle access 512-entry, 4-way WBPT 16-entry Hot instruction table
Main Memory	200 cycle access

4-way 512-entry for all the caches. Table II summarizes the system configuration parameters.

Memory-intensive benchmarks from SPEC CPU2006 are used to evaluate *ILRU*. Benchmarks are fast forwarded for 5 billion instructions and simulated for 1 billion instructions. Caches are warmed up during fast forwarding. To select memory-intensive benchmarks, we choose benchmarks with at least 1% IPC improvement when one of the caches was doubled in size under traditional LRU replacement, which amount to 14 benchmarks. We neglect the non-memory-intensive benchmarks as there is negligible change in performance with any of the simulated replacement policies.

ILRU is evaluated against DRRIP [17], and SHiP [13]. We use 2-bit DRRIP with $\epsilon = 1/32$, 32 sampling sets, and 10-bit PSEL counter. We use SHiP-PC-S with 16K-entry SHCT, and 14-bit partial PC as the signature. We only apply DRRIP and SHiP to the L3 cache, and employ traditional LRU replacement on the L1 and L2 caches because DRRIP and SHiP do not improve cache performance in upper level caches, as mentioned in [17].

We also compare *ILRU* with TA-DRRIP [17] in a multi-core, shared LLC environment. Each multi-programmed workload is a combination of concurrently running single-threaded benchmarks. Each single-threaded benchmark is run on each individual core. Due to the number of possible combinations, we study all the combinations in a 2-core configuration, and 501 randomly generated combinations in a 4-core configuration. The size of the LLC is 2MB multiplied by the number of cores. Each simulation is run until *all* benchmarks execute 1 billion instructions. When one benchmark finishes earlier than the others, it is rewinded and re-run from the beginning like prior works [5], [18], [19], [14], [17], [13], [20]. The reported results are collected only for the first 1 billion instructions window.

A. Performance

Figure 6 (a) shows MPKI reduction of *ILRU* over traditional LRU replacement in each level of caches. *ILRU* reduces the MPKI by an average of 7.0% for L1, 9.1% for L2, and 8.7% for L3. Cache is utilized more efficiently in *ILRU* compared to LRU because *ILRU* allows each instruction to have only working blocks which are often smaller than the miss frequency. Figure 6 (b) shows MPKI reduction of *ILRU* and prior works over traditional LRU replacement in the LLC. *ILRU* reduces more misses compared to DRRIP and SHiP, which reduce MPKI by an average of 3.3% and 4.9%, respectively. Note that MPKI reduction in L3 for *ILRU* is the same in both figures.

Figure 7 presents the IPC improvement for *ILRU* and prior works over the traditional LRU replacement. For *ILRU*, we further breakdown the improvement by showing the contribution from each level of cache. *ILRU* improves IPC, on average, by 5.3% in total, where L1, L2, and L3 contribute 0.4%, 1.2%, and 3.7%, respectively. DRRIP and SHiP improve performance by an average of 2.2% and 2.8%, respectively. *ILRU* outperforms DRRIP and SHiP in most

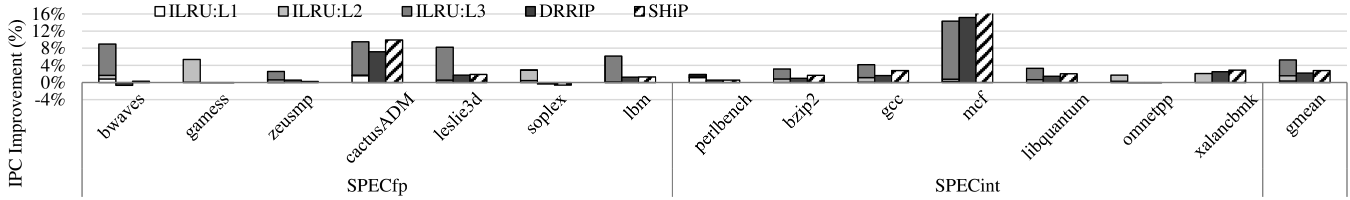


Figure 7. IPC improvement over LRU. *ILRU* is shown with contributing factors from L1, L2, and L3.

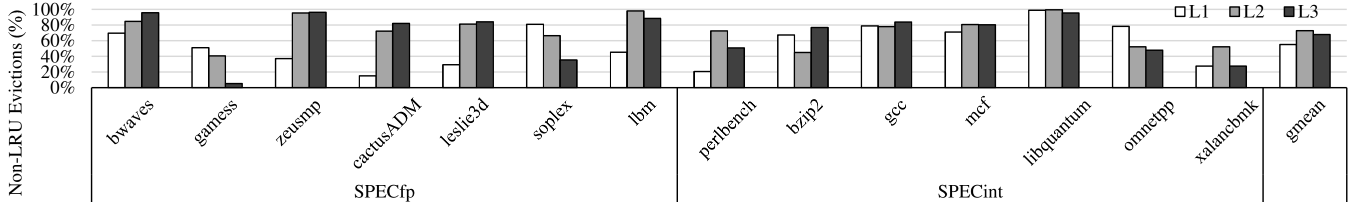


Figure 8. A fraction of non-LRU evictions (including bypasses) for *ILRU* in different levels of caches.

cases because fine grain cache partitioning at the instruction granularity can reduce the interferences not only between non-temporal and other references, but also between instructions with diverse reuse characteristics. IPC improvement does not directly correlate with the MPKI reduction for a number of reasons. First, out-of-order memory support can hide miss latencies using memory level parallelism (MLP). Second, the absolute MPKI might be so small that a large percentage change in MPKI is negligible in terms of its influence on the overall performance.

Workloads such as $gamess$, $soplex$, $bzip2$, and $omnetpp$ have smaller MPKI in the L3 compared to other workloads, although their MPKIs in upper level caches are large. Among them, $soplex$, and $omnetpp$ suffer from the increase in L3 MPKI for *ILRU*, DRRIP, and SHiP. In these benchmarks, L3 MPKI is so small that mispredictions are not observed in the sampling sets yet while non-sampling sets incur more misses. If we simulate for long enough period, these misses will balance out with the reduced misses from correct predictions. Also, their impact on performance is negligible due to the small value of the original L3 MPKI under traditional LRU replacement as shown in Figure 7. $bzip2$ is an opposite case, where a 45% reduction in L3 MPKI contributes only 2.3% performance improvement.

ILRU discovers an interesting case in $xalancbmk$. The performance achieved by DRRIP and SHiP from the L3 is equally gained from the L2 for *ILRU*. *ILRU* does not reduce L3 MPKI at all because the opportunity no longer exists as it was already exploited in L2. It is better to reduce misses in upper level caches because it consumes less dynamic power. Further experiments showed that DRRIP and SHiP also achieve similar improvement for L2 in $xalancbmk$, but they do not provide benefit in general. In $gamess$, only *ILRU* can achieve a performance improvement. Because $gamess$ has sudden drop in the number of misses when moving from L2 to L3, only *ILRU* can only improve the performance of $gamess$ by reducing L2 misses. In $bwaves$, *ILRU* needs to be applied to all levels of cache to achieve a gain in performance. The reduction in L1 misses converts to performance improvement after the few remaining misses with long latencies are removed in lower level caches.

B. Non-LRU Evictions

Figure 8 depicts the fraction of cache block evictions in *ILRU* that are non-LRU evictions including bypasses. On average, the fraction of non-LRU evictions is 55% for L1, 73% for L2, and 68% for L3. In $zeusmp$, and $libquantum$, most of the evictions are non-LRU evictions, but their MPKI reductions and performance improvements are small. These workloads have almost the same MPKI in all levels of cache, meaning that misses in L1 also result in misses in lower level caches. In such cases, misses are unlikely to be reduced even if *ILRU* has large number of non-LRU evictions because cache capacity is the main reason for the misses. A large number of non-LRU evictions shows that the LRU block is not necessarily the best candidate for eviction. *ILRU* can reduce the interference between memory instructions more effectively than the traditional LRU.

C. Sensitivity of ILRU

Figure 9 illustrates the sensitivity of *ILRU* to various parameters. Figure 9 (a) and (b) show the performance improvement over LRU when we change the number of entries in the hot instruction table for L2 and L3, respectively. All denotes the use of the 15-bit partial PC to identify instructions instead of using the hot instruction table. The size of a hot instruction table can influence the performance in two ways. Destructive aliasing, where the number of hot instructions is larger than the size of the hot instruction table, causes *ILRU* to evict the wrong blocks. Figure 9 (a) shows such case, where increasing the number of entries for the hot instruction table in the L2 shows more performance improvement. Constructive aliasing, where multiple infrequent

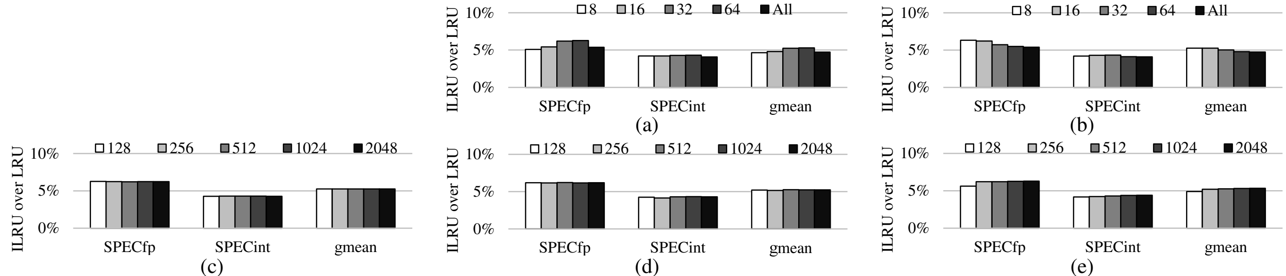


Figure 9. Sensitivity of *ILRU* to the size of a hot instruction table in (a) L2, and (b) L3; and to the size of a WBPT in (c) L1, (d) L2, and (e) L3.

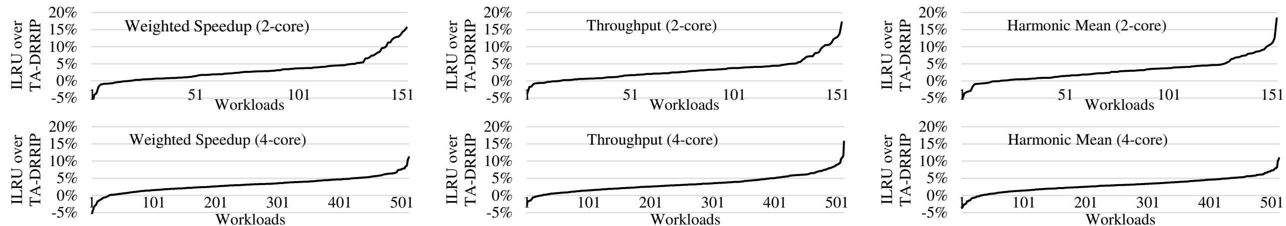


Figure 10. *ILRU* performance improvement over TA-DRRIP on a shared cache for 2-core (top) and 4-core (bottom) workloads.

instructions alias, causes these instructions to occupy fewer ways together. Constructive aliasing occurs in Figure 9 (b) and explains the higher performance improvements in 32 and 64 entries compared to using the partial PC in Figure 9 (a), where the partial PC identifies instructions perfectly. For the L2 hot instruction table, we chose 32 entries, which had peak performance with the least hardware overhead. The L3 hot instruction table had peak performance at 8 entries in SPECfp and at 16 entries in SPECint. We conservatively picked 16 entries for the L3 hot instruction table.

Figure 9 (c), (d) and (e) depict the sensitivity of *ILRU* to the size of the WBPT. Although it is a small difference, L1 and L2 showed peak performance at 512 entries. L3 shows small performance increase once the size of the WBPT exceeds 256 entries. Although 256 entries are enough on average-wise, a few benchmarks lose performance with 256 entries compared to 512 entries. Thus, we conservatively chose 512 entries for the WBPTs in all of the evaluations.

D. ILRU *on a Shared LLC*

Figure 10 shows the performance of *ILRU* over TA-DRRIP on a shared LLC for 2-core and 4-core configurations. We use three commonly used metrics to measure the performance of multi-programmed workloads: weighted speedup, throughput, and harmonic mean of IPC improvements. The weighted speedup indicates the reduction in total execution time. The throughput is the raw performance improvement per cycle. The harmonic mean balances between fairness and performance. On average, *ILRU* shows 3.2%, 3.2%, and 3.0% improvements over TA-DRRIP in weighted speedup, throughput, and harmonic mean, respectively, in the 2-core configuration, and 3.2%, 3.4%, and 3.0% improvements in the 4-core configuration.

For more than 90% of the multi-programmed workloads, *ILRU* improves over TA-DRRIP in all three metrics. TA-DRRIP performs better than *ILRU* when a multi-programmed workload consists of mcf, where DRRIP gained 1% more performance than *ILRU* in the single core configuration. However, *ILRU* still improves significant performance compared to LRU. The result shows that *ILRU* is still robust even in the shared environment, where contending memory instructions come from another application.

E. Hardware Overhead

Table III presents the storage overhead of *ILRU* in the whole system. A partial PC is additionally stored in a 8-entry load buffer and a 8-entry store buffer in a core, and miss status handling registers (MSHRs) in each level of cache. Each level of cache has overheads from the metadata in sampling sets and non-sampling sets. A WBPT exists for each cache. A hot instruction table exists for L2 and L3, but not for the L1 cache. The number of sampling sets is 128 sets, 128 sets, and 32 sets for L1, L2, and L3, respectively. Sampling sets also have shadow tags, which are 2-way, 4-way, and 8-way associative for L1, L2, and L3, respectively. Non-sampling sets require less overhead compared to sampling sets, and are the majority of the sets in lower level caches. In total, *ILRU* requires 29.7 kB additional storage in a three level cache hierarchy system, which is 1.3% of the total cache size. If the LLC is shared among multi-cores, the second column tells whether the additional storage overhead has to be replicated as the number of cores increases. The overhead from sampling sets and non-sampling sets in the LLC, which takes more than half of the total overhead, can be shared in multi-core environment.

The most significant portion of the logic overhead is instruction comparison, which needs the same number of comparators as the tag comparison. CACTI [21] estimates the area of the comparators for tag comparison to be less than 0.1% of the cache area. Therefore, doubling the number

Table III. HARDWARE OVERHEAD

Core	Per core	Partial PC in LD/ST buffer	16-entry	0.03 kB
L1	Per core	Sampling sets	128 sets	2.2 kB
		Non-sampling sets	0 sets	0 kB
		WBPT	512-entry	0.9 kB
		MSHR	8-entry	0.01 kB
L2	Per core	Sampling sets	128 sets	4.5 kB
		Non-sampling sets	384 sets	1.9 kB
		WBPT	512-entry	0.9 kB
		Hot instruction table	32-entry	0.08 kB
		MSHR	8-entry	0.01 kB
L3	Shared	Sampling sets	32 sets	2.3 kB
		Non-sampling sets	2016 sets	15.8 kB
		MSHR	16-entry	0.03 kB
	Per core	WBPT	512-entry	1.0 kB
		Hot instruction table	16-entry	0.04 kB
Total				29.7 kB

of comparators incurs negligible logic overhead.

To model the power overhead of *ILRU*, we used 32nm technology LSTP (Low Standby Power) process in CACTI. We gathered the average number of accesses to *ILRU* structures in the simulated workloads for the dynamic activities. The WBPT was modeled as a 4-way associative cache structure with a 8-bit tag (7-bit as a set index) and a 1B block. Shadow tags were modeled as a tag array of a cache with the corresponding sets and ways, where the data array was ignored. Extra metadata were modeled by increasing the storage from the original cache structure. Dynamic power overhead of *ILRU* in three-level cache hierarchy is 2.1%, while static power overhead is 2.2%.

VI. RELATED WORK

Numerous studies have been proposed to improve cache performance over LRU in both single-threaded and multi-programmed workloads. We categorize prior works into 1) modifying replacement policies, and 2) partitioning shared caches for different optimization objectives.

A. Replacement Policies

The problem of LRU replacement has been studied extensively, especially for the LLC. Bypass [6], [22], [23], [7], [8], [9] chooses not to insert blocks because the incoming block has no or distant reuse. Dead block prediction [24], [25], [26], [12] finds a dead block, which is no longer reused. Dead block prediction looks either trace, time, or reference count to detect a dead block. DIP [27] retains the working set for thrashing patterns by inserting an incoming block into LRU position. Another set of replacement policies predicts the re-reference interval to determine a victim block [28], [17], [29], [13]. For example, RRIP [17] predicts that incoming blocks have intermediate re-reference interval. Other works viewed replacement policies in different perspectives. Pseudo-LIFO [19] proposes that the blocks at the top of a fill stack have higher eviction probability. PDP [20] periodically computes a protecting distance, which is the number of accesses to a cache set to protect a newly inserted block from evictions without polluting the cache.

PC or instruction trace have been also used in many literatures for cache management by either predicting whether the incoming block should be bypassed [6], whether the block is dead [12], or whether the block has distant re-reference interval [13]. *ILRU* also is based on the idea of using an instruction as an identifier for the cache management. However, while these techniques predict a binary outcome, *ILRU* predicts the working block of an instruction, which is a multi-valued outcome hence covers more diverse scenarios.

ILRU primarily differs from prior works by viewing a cache as a shared resource between individual memory instructions. By reducing the interference, *ILRU* can improve cache performances in all levels of cache and for more diverse applications.

B. Cache Partitioning

Cache partitioning has been a focus in many shared cache management studies. Because a shared cache has to support multi-programmed and multi-threaded workloads, cache partitioning has to consider several objectives such as performance [4], [14], fairness [30], and quality of service (QoS) [31], [32]. Suh et al. [2], [3] estimate the marginal gain of increasing cache sizes and utilize this information to partition the cache. UCP [5] takes a similar approach, but incorporates a monitor that is independent of the observed cache behavior. Dybdahl et al. [10] optimizes for performance in NUCA by having private and shared partitions, and employs shadow tags to measure which partition benefits more with an additional cache block. While *ILRU* shares the idea of way partitioning as most of the prior cache partitioning studies, *ILRU* partitions for each instruction rather than for each application, resulting in performance improvement for single application as well as multiple applications. Also, *ILRU* uses in-place prediction and a single shared shadow tag rather than a private shadow tag per core.

VII. CONCLUSION

In this paper, we presented *Instruction-based LRU*, a fine grain cache partitioning using per-instruction working blocks for every level of the cache hierarchy. *ILRU* views a cache as a shared resource between individual memory instructions. When an instruction misses in a cache, it steals a block from another only when it needs more to capture the reuse characteristic. Otherwise, it evicts among the blocks inserted by itself, so that it never occupies more than its working blocks. *ILRU* naturally extends to shared caches by prepending the instruction identifier with a logical thread ID. Evaluations show that *ILRU* improved performance in all level of caches, and in more diverse workloads compared to prior LLC-only techniques. Overall, *ILRU* reduced the number of misses by an average of 7.0% for L1, 9.1% for L2, and 8.7% for L3, and improved the average performance by 5.3%. In the case of a 4-core configuration, *ILRU* improves over TA-DRRIP in more than 90% of the multi-programmed

workloads, with an average of 3.2%, 3.4%, and 3.0% for weighted speedup, throughput, and harmonic mean metrics, respectively. *ILRU* for a three-level cache hierarchy imposes a modest 1.3% storage overhead over the total cache size with 2.1% dynamic power overhead, and 2.2% static power overhead.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers as well as the fellow members of the CCCP research group for their valuable comments and feedbacks. This work is supported in part by the National Science Foundation under grant SHF-1217917 and by the Defense Advanced Research Projects Agency (DARPA) under the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) program.

REFERENCES

[1] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[2] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. of the 8th International Symposium on High-Performance Computer Architecture*, Feb. 2002, pp. 117–128.

[3] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, Apr. 2004.

[4] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource," in *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006, pp. 13–22.

[5] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance runtime mechanism to partition shared caches," in *Proc. of the 39th Annual International Symposium on Microarchitecture*, 2006, pp. 423–432.

[6] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A Modified Approach to Data Cache Management," in *Proc. of the 28th Annual International Symposium on Microarchitecture*, Dec. 1995, pp. 93–103.

[7] 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.

[8] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and insertion algorithms for exclusive last-level caches," in *Proc. of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 81–92.

[9] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng, "Optimal bypass monitor for high performance last-level caches," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012, pp. 315–324.

[10] H. Dybdahl and P. Stenstrom, "An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors," in *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, 2007, pp. 2–12.

[11] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *Proc. of the 33rd Annual International Symposium on Computer Architecture*, Jun. 2006, pp. 167–178.

[12] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proc. of the 43rd Annual International Symposium on Microarchitecture*, Dec. 2010, pp. 175–186.

[13] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. S. Jr., and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proc. of the 44th Annual International Symposium on Microarchitecture*, 2011, pp. 430–441.

[14] Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, Jun. 2009, pp. 174–183.

[15] 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 '05 Conference on Programming Language Design and Implementation*, Jun. 2005, pp. 190–200.

[16] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *Proc. of the 16th International Symposium on High-Performance Computer Architecture*, Jan. 2010, pp. 1–12.

[17] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 60–71.

[18] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. S. Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008, pp. 208–219.

[19] M. Chaudhuri, "Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches," in *Proc. of the 42nd Annual International Symposium on Microarchitecture*, 2009, pp. 401–412.

[20] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proc. of the 45th Annual International Symposium on Microarchitecture*, 2012, pp. 389–400.

[21] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," Hewlett-Packard Laboratories, Tech. Rep. HPL-2009-85, Apr. 2009.

[22] W. A. Wong and J.-L. Baer, "Modified LRU policies for improving second-level cache behavior," in *Proc. of the 6th International Symposium on High-Performance Computer Architecture*, Jan. 2000, pp. 49–60.

[23] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.

[24] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proc. of the 28th Annual International Symposium on Computer Architecture*, Jun. 2001, pp. 144–154.

[25] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: Predicting and optimizing memory behavior," in *Proc. of the 29th Annual International Symposium on Computer Architecture*, Jun. 2002, pp. 209–220.

[26] 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 Annual International Symposium on Microarchitecture*, 2008, pp. 222–233.

[27] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 381–391.

[28] G. Keramidas, P. Petoumenos, and S. Kaxiras, “Cache replacement based on reuse-distance prediction,” in *Proc. of the 2007 International Conference on Computer Design*, Oct. 2007, pp. 245–250.

[29] 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*, Feb. 2011, pp. 243–253.

[30] S. Kim, D. Chandra, and Y. Solihin, “Fair cache sharing and partitioning in a chip multiprocessor architecture,” in *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004, pp. 111–122.

[31] R. Iyer, “Cqos: A framework for enabling qos in shared caches of cmp platforms,” in *Proc. of the 2004 International Conference on Supercomputing*, 2004, pp. 257–266.

[32] J. Chang and G. S. Sohi, “Cooperative cache partitioning for chip multiprocessors,” in *Proc. of the 2007 International Conference on Supercomputing*, Jun. 2007, pp. 242–252.