Improving Inclusive Cache Performance with Two-level Eviction Priority

Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, Xu Cheng
Microprocessor Research and Development Center, Peking University, Beijing, China
{lilingda, tongdong, xiezichao, lujunlin, chengxu}@mprc.pku.edu.cn

Abstract—Inclusive cache hierarchies are widely adopted in modern processors, since they can simplify the implementation of cache coherence. However, it sacrifices some performance to guarantee inclusion. Many recent intelligent management policies are proposed to improve the last-level cache (LLC) performance by evicting blocks with poor locality earlier. Unfortunately, they are inapplicable in inclusive LLCs.

In this paper, we propose Two-level Eviction Priority (TEP) policy. Besides the eviction priority provided by the baseline replacement policy, TEP appends an additional high level of eviction priority to LLC blocks, which is decided at the insertion time and cannot be changed during their lifetime in the LLC. When blocks with high eviction priority are not in inner caches anymore, they get evicted from the LLC preferentially. Thus, the LLC can retain more useful blocks to improve performance. TEP can cooperate well with various baseline replacement policies. Our evaluation shows that TEP with NRU can improve the performance of inclusive LLCs significantly while requiring negligible extra storage. It also outperforms other recent proposals including QBS, DIP, and DRRIP.

I. INTRODUCTION

The performance gap between processors and memory has been widened for several decades, and modern processors use multi-level cache hierarchies to bridge the gap. An important design choice of the multi-level cache hierarchy is whether the last-level cache (LLC) should guarantee inclusion [1], [2]. Since inclusive caches include all blocks in inner caches, they can filter unnecessary cache coherence messages to simplify the implementation of cache coherence protocol [1], [3]. Therefore, inclusive cache hierarchies are adopted by many recent desktop, embedded, and server processors [4], [5], [6].

However, compared to non-inclusive and exclusive caches, inclusive caches suffer performance loss for two reasons. One is that the effective cache space is reduced due to data duplication. The other reason is that, when a block is evicted from the LLC, that block should also be invalidated in inner caches to guarantee inclusion. Those blocks which are invalidated in inner caches to guarantee inclusion are called inclusion victims [7]. Since inclusion victims may show good temporal locality in inner caches, evicting them earlier can cause extra misses.

Recently many intelligent management policies are proposed to improve the LLC performance. By predicting the temporal locality of LLC blocks, those techniques evict blocks with poor locality earlier or directly bypass them from the LLC [8], [9], [10], [11], [12], [13], [14]. However, in inclusive LLCs, the early eviction of blocks increases the probability that those blocks are still in inner caches, and thus causes more inclusion victims, which may show good locality in inner caches. Therefore, those intelligent policies can degrade the performance of inclusive LLCs, and blocks with poor locality should not be evicted by the LLC until their locality is exhausted in inner caches.

In this paper, we propose an inclusive cache replacement policy called Two-level Eviction Priority (TEP) policy. TEP appends an additional level of eviction priority to LLC blocks. When a block is brought into the LLC, if it is predicted to have poor temporal locality, TEP marks it with high eviction priority. On a miss, if one of the victim candidates which have high eviction priority is not present in inner caches, it is chosen to be the victim preferentially. Otherwise, the baseline replacement policy, which provides the second level of eviction priority, selects the victim from the remaining victim candidates which have low eviction priority.

In doing so, blocks with poor locality are evicted right after they are not in inner caches, and thus they have short lifetime in the LLC while avoiding the eviction of inclusion victims. On the other hand, blocks with good locality are selected for replacement only when all blocks with poor locality are in inner caches. Thus, the LLC can retain them longer to capture more hits.

TEP can cooperate well with any baseline replacement policy, and it needs negligible modifications to the existing cache design. We evaluate TEP with NRU, LRU, and SRRIP [9]. Our experiments show that TEP can improve the performance of inclusive LLCs significantly. Especially, TEP with NRU reduces average misses by 11.0% compared to LRU and thus achieves a geometric mean speedup of 4.3% in a 512KB L2 cache. TEP with NRU also outperforms other recent proposals significantly, including QBS [7], DIP [8], and DRRIP [9], while TEP only requires roughly 2KB of storage overhead.

II. MOTIVATION

In order to illustrate why recently proposed intelligent replacement policies cannot be applied to inclusive LLCs, we evaluate the performance of DIP and DRRIP in both non-inclusive and inclusive LLCs.

This work is supported by the National Science and Technology Major Project of the Ministry of Science and Technology of China under grant 2009ZX01029-001-002-2.

1Unless stated otherwise, cache refers to the last-level cache in this paper, and the L2 cache is used as the LLC in this paper.

2We use 2-bit DRRIP in the experiments because it performs better.
The detail of our experiment configuration is described in Section IV.

The proposed policy should achieve two design goals: it should make effective use of the cache space to improve performance, while the number of inclusion victims is minimized.

III. Two-level Eviction Priority Policy

A. Overview

In order to avoid evicting inclusion victims, blocks should not be evicted from the LLC until their temporal locality in inner caches is exhausted. To improve the LLC performance, the lifetime of useless blocks in the LLC should be as short as possible. We propose Two-level Eviction Priority (TEP) policy for the inclusive cache replacement to achieve the two goals above.

TEP associates each LLC block with an Eviction Priority Bit (EPB) to indicate its first level of eviction priority. When a block is brought into the cache, a temporal locality predictor is employed to predict its locality. If it is predicted to have poor locality, its EPB is set to 1 to indicate its high eviction priority. Otherwise, its EPB is set to 0.

On a miss, among all victim candidates in the corresponding set, blocks whose EPBs are 1 are checked first. TEP starts the search from the first physical way to the last physical way. For a victim candidate, TEP identifies whether it is in inner caches or not. If it is not present in inner caches, it is selected for replacement. Otherwise, TEP checks the next candidate until a candidate which is not in inner caches is found, or all blocks whose EPBs are 1 have been checked. If all blocks whose EPBs are 1 are present in inner caches, the baseline replacement policy provides the second level of eviction priority to select a victim candidate among the remaining blocks whose EPBs are 0. If the victim candidate is in inner caches, its replacement state is updated to the Most Recently Used (MRU) position. Then the baseline replacement policy selects the next candidate to restart the process. Figure 2 shows an example of the victim selection process for an 8-way LRU-managed cache.

In doing so, for blocks with poor locality (those whose EPBs are 1), they get evicted from the LLC right after they are removed in inner caches. Thus, their lifetime in the LLC is shortened while not evicting inclusion victims. For blocks with good locality (those whose EPBs are 0), they are selected for replacement only when all blocks with high eviction priority reside in inner caches. Thus, the LLC can retain them longer to improve performance.
TEP can cooperate with any baseline replacement policy. In our experiments, we evaluate the performance of TEP when it works with the Not Recently Used (NRU) policy, LRU, and SRRIP. For TEP, there are two key problems. One is how to predict the temporal locality of blocks in the LLC. The other is how to identify whether a block is in inner caches or not. We will introduce them in the next two sections respectively.

B. Temporal Locality Prediction

Figure 3 shows the structure of our temporal locality predictor. On a cache miss, TEP uses a Replacement History Table (RHT) to record a pair of the incoming block (IB) and the victim block (VB), which is selected by the baseline replacement policy. The program counter (PC) of the instruction which accesses IB is also recorded. Then according to the following reuse sequence on an IB-VB pair, the temporal locality of IB can be determined:

- Condition 1: if IB is accessed earlier, it indicates that IB has good locality;
- Condition 2: if VB is accessed earlier, it indicates that IB has poor locality;
- Condition 3: if IB is not reused before eviction, it also indicates that IB has poor locality.

The first and third conditions are easy to understand, because a reused block is expected to have good locality, and no reused blocks should have poor locality. The heuristic of the second condition is that VB is expected to have the worst locality among all blocks in the cache. If VB is accessed earlier than IB, it indicates that the locality of IB is worse than that of VB. Thus, IB is considered to have poor locality.

A saturating counter table called Locality Prediction Table (LPT) is used to learn and predict the temporal locality of blocks. LPT learns the locality based on the PC of the instruction which accesses the block, because previous work has shown that the prediction using PC is more accurate compared to other methods[10], [12], [15]. All counters in the LPT are initiated to 0. When the locality of an IB-VB pair is learned in the RHT, the recorded PC of that pair is used to index a corresponding counter in the LPT. If that block shows good locality, the counter is increased by 1. Otherwise, it is decreased by 1.

On a miss, the incoming block consults the LPT with its PC to predict the locality. If its corresponding counter is less than 0, it is predicted to be poor locality, and its EPB is set to 1. Otherwise, its EPB is set to 0.

On an access to block x:

```c
for each valid entry A in the corresponding set of RHT
  if x.tag == A.IB_tag // Condition 1
    LPT[A.PC]++; 
    Invalidate A;
  else if x.tag == A.VB_tag // Condition 2
    LPT[A.PC]--; 
    Invalidate A;
  if x misses in the cache
    y = Select_Victim_Candidate();
    for each valid entry A in the corresponding set of RHT
      if y.tag == A.IB_tag // Condition 3
        B.IB_tag = x.tag;
        B.VB_tag = y.tag;
        LPT[A.PC]--; 
        Invalidate A;
        if RHT.Record(x) == true
          B = RHT.Select_Victim(x); // Select an entry to record x
          B.IB_tag = x.tag;
          B.VB_tag = y.tag;
          if LPT[x.PC] < 0
            x.EPB = 1;
          else
            x.EPB = 0;
```

Figure 4 shows the detail of our temporal locality prediction algorithm. We use a 128-entry 8-way set-associative RHT in our experiments. Besides PC, IB tag, and VB tag, each RHT entry contains 1 bit to indicate whether it is valid, and 3 bits for implementing the replacement policy of RHT, which is LRU. To reduce the hardware overhead, the RHT does not record the IB-VB pair on every miss. Only 1/128 of misses are recorded. Partial tags are also used to reduce the overhead of recording IB and VB, and the RHT keeps the lower 16 bits of IB and VB tags. For a 1024-entry LPT, the 11th to 2nd bits of PC are stored in the RHT4. The shortened 10-bit PC is delivered along with the cache request in the cache hierarchy like all prior PC based methods [10], [12], [16].

A lot of techniques can predict the temporal locality of LLC blocks [8], [9], [10], [12], [16], [17], [18], and they can all potentially be used as the temporal locality predictor for TEP. Therefore, TEP is a general framework for inclusive cache replacement.

C. Inner Cache Block Awareness

To make TEP be aware of whether an LLC block resides in inner caches, we use a query based method which is similar to QBS [7]. When a victim candidate is chosen, TEP sends its address to inner caches to query whether it is in them. Then inner caches look up that address and return the response. To avoid sending too many queries to boost the bus traffic, we set

\[ \text{EPB} = 0; \]

4 The benchmarks are compiled to Alpha binaries, and the lowest 2 bits of PC in Alpha instructions are always 0.
a threshold for the maximum number of queries allowed in one replacement. If in one replacement of the LLC, the number of queries has reached the threshold, the next candidate will be selected as the victim without querying inner caches. Our experiments show that a qualified victim can be found within 2 queries generally, and thus the threshold is set to 2. As stated in [7], the bandwidth requirement for queries is proportional to the number of LLC misses, which is very small. Thus, the extra traffic is negligible.

IV. EVALUATION

A. Experimental Methodology

We use a modified version of SimpleScalar [19] as our simulator. Table I shows the detailed configuration of the simulator. It models a 2-level cache hierarchy, and the L2 cache is used as the LLC. The inner cache size to the LLC size ratio is similar to that of modern processors [4], [5]. When inclusion is satisfied, the L2 cache sends back-invalidation messages to both the L1 icache and dcache on the replacement. Besides misses, the simulator also allocates MSHRs for our queries sent by the L2 cache to model the extra traffic introduced by those messages.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/Issue/Commit</td>
<td>4</td>
</tr>
<tr>
<td>Width</td>
<td>32-entry</td>
</tr>
<tr>
<td>Reorder Buffer</td>
<td>32-entry</td>
</tr>
<tr>
<td>Load/Store Queue</td>
<td>32-entry</td>
</tr>
<tr>
<td>L1 ICache</td>
<td>64B blocks, 32KB, 4-way, 1 cycle, LRU</td>
</tr>
<tr>
<td>L1 DCache</td>
<td>64B blocks, 32KB, 4-way, 1 cycle, LRU, 2 ports, 8 MSHRs, 16 WriteBuffers</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>64B blocks, 512KB, 16-way, 10 cycles, 32 MSHRs, 64 WriteBuffers</td>
</tr>
<tr>
<td>Memory Latency</td>
<td>150 cycles</td>
</tr>
<tr>
<td>RHT</td>
<td>32-entry, 8-way</td>
</tr>
<tr>
<td>LPT</td>
<td>1024-entry, 3-bit counters</td>
</tr>
</tbody>
</table>

We use the precompiled Alpha binaries of SPEC CPU2000 benchmarks in our experiments, which was available from www.simplescalar.com. A subset of memory intensive benchmarks whose compulsory misses are less than 50% are chosen, because compulsory misses cannot be reduced by any replacement policy. SimPoint [20] is used to select representative 250 million instructions to execute for each benchmark.

B. Results for TEP with NRU

At first we evaluate the performance of TEP when NRU is used as the baseline replacement policy. Besides TEP, we also investigate the performance of various replacement policies, including LRU in the non-inclusive cache (LRU_non), QBS, DIP, and DRRIP. In addition, we also study DIP+QBS and DRRIP+QBS, which extend DIP and DRRIP with QBS respectively. DIP+QBS and DRRIP+QBS query inner caches when selecting a victim candidate.

Figure 5 shows misses per thousand instructions (MPKI) of the L2 cache for various techniques in inclusive LLCs, which is normalized to that of LRU. TEP reduces the average MPKI by 11.0% compared to LRU, while the MPKI reduction of other techniques is less than 5%.

Figure 6 shows the speedup for various techniques. The speedup is computed by dividing the IPC of various techniques by that of LRU in inclusive caches. TEP achieves a geometric mean speedup of 4.3% compared to LRU. Compared with other techniques, in which the best performance improvement is 1.3% for DRRIP+QBS, TEP outperforms them significantly.

Since QBS is based on LRU, its performance is lower than that of LRU_non. DIP and DRRIP suffer performance loss since they cause more inclusion victims, as stated in Section II. For DIP+QBS and DRRIP+QBS, they insert most blocks with poor locality into the LRU position, which are likely in inner caches when they are selected for replacement. Thus, when QBS identifies that those blocks are in inner caches, they are considered to have good locality and updated to the MRU position. As a result, DIP+QBS and DRRIP+QBS have similar behavior and performance as LRU, and thus they lost the opportunity to improve performance. That is the reason why TEP uses one extra level of eviction priority to record blocks with poor locality consistently. Those blocks are checked repeatedly until they are not in inner caches, and thus TEP can evict them as early as possible.

Figure 7 shows the fraction of blocks which are predicted to be poor locality. For benchmarks which TEP performs well such as art and ammp, more than 50% of incoming blocks...
are predicted to have poor locality. The EPBs of those blocks are set to 1, and thus they are evicted earlier to release cache space for useful blocks.

C. Results for TEP with Other Policies

Besides NRU, we also evaluate the performance of TEP with other baseline replacement policies including LRU and SRRIP. As shown in Figure 8, the results are similar to that of TEP with NRU. TEP+LRU outperforms LRU by 4.2% and reduces average misses by 10.5%. TEP+SRRIP outperforms LRU by 3.8% and reduces average misses by 9.2%. Compared to SRRIP, the performance gain of TEP+SRRIP is 3.8%. Our results show that TEP can cooperate with various replacement policies to improve performance. Among them, NRU is the simplest and requires the least hardware overhead. Therefore, we focus on the study of TEP with NRU in the rest of this paper, and TEP refers to TEP with NRU in this paper.

D. Sensitivity to the Cache Size

Figure 9 shows the speedup for different cache sizes. On the other hand, TEP can improve the performance of inclusive caches for all cache sizes, although for small caches the performance gain is limited because there is less space to place useful blocks, and for large caches the working set is more likely to fit into the cache.

E. Sensitivity to the Size of RHT and LPT

We investigate the performance sensitivity of TEP to the RHT size. The results show that the performance is not sensitive to different RHT sizes. A 128-entry RHT can perform well enough and more entries are not necessary.

Figure 10 shows the performance sensitivity to different LPT sizes with a 128-entry, 8-way RHT. We vary the LPT size from 128 to 4096 and the LPT counter size is changed from 1 bit to 5 bits. The experimental results show that a 1024-entry LPT with 3-bit counters is enough.

F. Storage

For TEP, each cache block needs 1-bit EPB. Each RHT entry contains 1 valid bit, 3 replacement bits, 10 bits to keep the shortened PC of IB, 16-bit IB tag and 16-bit VB tag. The saturating counter of LPT is 3-bit. Therefore, TEP totally consumes \( (1 \times 8192 + (1 + 3 + 10 + 16 + 16) \times 128 + 3 \times 1024) \) bits = 2.09KB of extra storage, which is less than 0.5% of the total storage of a 512KB L2 cache. Since NRU needs 1KB of storage, TEP with NRU consumes roughly 3KB of storage in total. Compared with other recent proposals, TEP with NRU performs best while requiring a moderate storage overhead.
V. RELATED WORK

A. Inclusive Cache Management

In order to reduce inclusion victims, global replacement [21, 22] exposes the hits in inner caches to the LLC. On an inner cache hit, a message is sent to update the replacement state of that block in the LLC. For direct-associative network caches, Fletcher et al. propose three methods to reduce inclusion victims [23]: increasing the cache associativity, using a victim cache, and adding a snooper filter to relax the inclusion property. Jaleel et al. propose three Temporal Locality Aware policies to improve inclusive cache performance [7]: Temporal Locality Hint (TLH) conveys the temporal locality in inner caches by sending locality hints; Early Core Validation (ECV) invalidates inner cache blocks before being evicted by the LLC to derive their locality; Query Based Selection (QBS) queries inner caches to get the locality. However, those proposals only focus on reducing inclusion victims, and they cannot make effective use of cache space to improve performance.

B. Non-inclusive or Exclusive Cache Management

Many intelligent cache management policies are recently proposed to improve the LLC performance, and they are all designed for non-inclusive or exclusive LLCs. DIP [8] dynamically inserts incoming blocks into the LRU position to avoid thrashing. RRIP [9] distinguishes no reused blocks with others to evict them earlier. Pseudo-LIFO [24] uses a fill stack to keep blocks in the bottom of the stack stably. SHIP [10] proposes a signature based predictor for re-reference interval prediction.

Dead block prediction techniques improve the LLC performance by predicting the last touch of a block. Based on how to predict, they are classified into trace based [16, 25], counter based [11], and time based [17]. Cache burst predictor [18] makes prediction for a burst of accesses to improve accuracy. Sampling dead block prediction [12] learns a fraction of sets for low overhead and high prediction accuracy.


All those techniques can potentially be extended with TEP to apply them in inclusive LLCs. We leave them in the future work.

VI. CONCLUSION

The implementation of the cache coherence protocol can be simplified when using an inclusive cache hierarchy. However, the performance is limited to satisfy the inclusion property. When a block is evicted from the LLC, that block in inner caches must be invalidated, which makes recently proposed intelligent cache management policies inapplicable in inclusive LLCs. In this paper, we propose Two-level Eviction Priority (TEP) policy for inclusive LLCs. TEP marks blocks with poor temporal locality in the LLC. Once those blocks are not resident in inner caches, they will be selected for replacement immediately. As a result, without evicting inclusion victims, TEP shortens the lifetime of useless blocks in the LLC to improve performance. Our evaluation shows that TEP can significantly improve the performance of inclusive LLCs. Although we only evaluate TEP in single-thread environments, it is straightforward to apply TEP in multi-thread environments, and it is a part of our future work.