Li LD, Lu JL, Cheng X. Retention benefit based intelligent cache replacement. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(6): 947–961 Nov. 2014. DOI 10.1007/s11390-014-1481-2

Retention Benefit Based Intelligent Cache Replacement

Ling-Da Li[1,2,3] (李凌达), *Student Member, CCF, ACM*, Jun-Lin Lu[1,2,3,*] (陆俊林), *Member, CCF* and Xu Cheng[1,2,3] (程 旭), *Member, CCF*

[1]*Microprocessor Research and Development Center, Peking University, Beijing 100871, China*

[2]*Engineering Research Center of Microprocessor and System, Ministry of Education, Beijing 100871, China*

[3]*School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*

E-mail: {lilingda, lujunlin, chengxu}@mprc.pku.edu.cn

Received December 19, 2013; revised May 6, 2014.

Abstract The performance loss resulting from different cache misses is variable in modern systems for two reasons: 1) memory access latency is not uniform, and 2) the latency toleration ability of processor cores varies across different misses. Compared with parallel misses and store misses, isolated fetch and load misses are more costly. The variation of cache miss penalty suggests that the cache replacement policy should take it into account. To that end, first, we propose the notion of retention benefit. Retention benefits can evaluate not only the increment of processor stall cycles on cache misses, but also the reduction of processor stall cycles due to cache hits. Then, we propose Retention Benefit Based Replacement (RBR) which aims to maximize the aggregate retention benefits of blocks reserved in the cache. RBR keeps track of the total retention benefit for each block in the cache, and it preferentially evicts the block with the minimum total retention benefit on replacement. The evaluation shows that RBR can improve cache performance significantly in both single-core and multi-core environment while requiring a low storage overhead. It also outperforms other state-of-the-art techniques.

Keywords retention benefit, replacement, last-level cache

1 Introduction

Cache performance, especially last-level cache performance, is crucial to the system performance due to the increasing memory access latency. In order to reduce the processor stall time on cache misses, a large number of cache management policies are proposed recently to improve cache performance. Most of these proposals focus on reducing the absolute cache miss count[1-8], and they implicitly assume that all cache misses result in equal performance degradation. However, this assumption is inaccurate in modern systems for two reasons.

The first reason is that memory access latency is not uniform in modern systems. There are many sources of the variation of memory latency. First, in modern DRAM systems, memory requests need additional serving time when the request queue is not empty or bank conflicts occur, and memory requests that access the same row as previous requests are served faster. Second, the interconnect network introduces variable access latency. For instance, requests to remote memory usually spend more time than those to local memory. Moreover, the disparity in memory latency will become even larger in future systems. Due to the stringent power and energy limitation in modern system design, heterogeneous multi-core systems will become more and more popular in the future. In such systems, multiple different memory techniques are used simultaneously (e.g., eDRAM, PCRAM, STT-RAM[9]), where the access latency depends on the type of the accessed memory. Besides, such systems also tend to use hybrid memory hierarchies (e.g., 3D-stacked DRAM caches[10]). In such scenarios, access latency depends on which level of the memory provides the data.

The other reason for the disparity in miss penalty is that to reduce the processor stall time on cache misses, modern processors make use of various techniques such as non-blocking caches[11] and prefetching[12] to serve multiple cache misses in parallel. Using these techniques makes the processor stall time on a cache miss depend not only on its memory access latency, but also on the situation of other concurrent misses. For instance, the memory access latency of cache misses

Regular Paper

The work was supported in part by the National Science and Technology Major Project of the Ministry of Science and Technology of China under Grant No. 2009ZX01029-001-002-2.

*Corresponding Author

©2014 Springer Science + Business Media, LLC & Science Press, China

which can be served in parallel can be partly hidden by that of other concurrent misses, and these misses thus have small performance impact. On the other hand, cache misses that occur in isolation can cause significant performance loss.

The variation of miss penalty motivates the requirement for a miss penalty aware cache replacement policy, which should focus on reducing the aggregate miss penalty (i.e., processor stall cycles on cache misses) rather than the aggregate miss count. Some recently proposed cache replacement policies have taken into account the miss penalty on replacement[13-18]. These proposals compute and record the miss penalty when a block is inserted into the cache, and preferentially replace blocks with small miss penalty.

However, only considering the miss penalty is not enough. For instance, assume that block A has a large miss penalty of 200 cycles on its insertion and is only accessed once, while block B has a small miss penalty of 50 cycles and receives 9 hits when being resident in the cache, and on each hit it prevents the processor from stalling for another 50 cycles. In such a scenario, reserving B will save an aggregate penalty of 500 cycles, while it is 200 cycles for A, and thus the cache should reserve B instead of A. If we only consider the one-time penalty on cache misses, block A will incorrectly be preferred. Therefore, besides the penalty on cache misses, it is also important to evaluate the potential saved penalty due to cache hits, and the cache replacement policy should take into account both of them to make replacement decisions.

To evaluate both the penalty on cache misses and the saved penalty on cache hits, we propose the notion of *retention benefit*. The retention benefit of a block represents the increment of processor stall cycles assuming that it is not resident in the cache. We also propose a simple method to compute the retention benefit for cache requests in superscalar processors.

Then, we propose Retention Benefit Based Replacement (RBR) to maximize the aggregate retention benefits of blocks reserved in the cache. RBR associates each cache block with a retention benefit value (RBV) to record its total retention benefit. On each access (including hits and misses) to a cache block, RBR computes the retention benefit and then accumulates it to the RBV of that block. On replacement, the block with the minimum RBV is selected as the victim block.

RBR can address the retention benefit variation of different cache requests from both intra-program and inter-program, and it is also prefetch-aware. Besides, since RBR can adapt to the variation of miss penalty intelligently, it can be applied in future systems without requiring any extra effort of redesign and verification.

We model a modern desktop system to evaluate the performance of RBR in the last-level cache. Our evaluation shows that RBR can improve cache performance significantly while requiring a low storage overhead. On average, RBR outperforms LRU (Least Recent Used) by 6.3% and 5.3% for single-core and 4-core workloads respectively in the absence of prefetching. In the presence of prefetching, its average performance improvement is 6.7% and 5.9% for single-core and 4-core workloads respectively. It also outperforms other state-of-the-art techniques including MLP-aware replacement[16], DIP[3], RRIP[6], PIPP[5], and UCP[2].

The rest of this paper is organized as follows. Section 2 discusses some related work. Section 3 introduces the notion of retention benefit and its computation method. Section 4 describes the design and implementation of RBR. Section 5 shows the experimental methodology, and then Section 6 analyzes the results. Finally, Section 7 concludes this paper.

2 Related Work

Extensive research has been done to improve cache performance. Based on the goal, these work can be classified into two major categories: miss count based policies which aim to reduce the miss count, and miss penalty based policies which aim to reduce the miss penalty. We will first introduce the primary work of these two types of policies respectively in this section. Then we will introduce some memory aware cache management policies.

2.1 Miss Count Based Policies

A lot of studies propose to improve cache performance by reducing the miss count. DIP[3] attempts to insert most incoming blocks into the LRU position to avoid thrashing when the working set is larger than the cache size. Pseudo-LIFO[19] uses a fill stack instead of the LRU stack, and prioritizes to replace blocks on the top of fill stack. Keramidas *et al.* proposed to explicitly predict the reuse distance to guide replacement[20]. RRIP[6] distinguishes reused blocks with no reused ones, and evicts no reused blocks preferentially. SHiP[8] can further improve the performance of RRIP with a signature-based re-reference interval predictor, and their signatures include memory region, PC, and instruction sequence. PACMan[21] extends RRIP to make it prefetch-aware. Duong *et al.* proposed to protect cache blocks within a predicted reuse distance[22].

Dead block prediction techniques try to identify blocks that will not be accessed again (i.e., dead

blocks), and evict them preferentially to improve performance. Dead block prediction can be classified into three categories based on how to identify dead blocks: trace-based[1], time-based[23], and counter-based[4]. By making prediction for continuous access sequences, cache burst predictor[24] can improve dead block prediction accuracy. SDBP[7] samples a part of sets to reduce conflicts in the predictor for low overhead and high prediction accuracy.

Bypass techniques improve cache performance by bypassing blocks with poor locality. Based on how to predict the locality, these studies can be classified into address-based[25-28], block-based[29-30], and PC-based[31-33]. LRF[34] combines address-based and PC-based methods to improve performance. DSB[35] adjusts the bypass probability based on which one is accessed first between the incoming block and the victim block. Gaur *et al.* proposed a bypass and insertion algorithm for exclusive last-level caches[36]. It classifies blocks based on their accessed time in the cache hierarchy. OBM[37] makes bypass decisions by predicting the behavior of the optimal bypass.

To improve shared cache performance, some recent work proposed to partition shared caches to minimize the aggregate miss count of multi-core processors. UCP[2] collects the cache utility information for each core. Then based on the collected utility information, UCP decides the cache allocation for each core to minimize the total miss count. TADIP[38] extends DIP to select the best insertion policy for each core. PIPP[5] changes the insertion and promotion policy of different cores to partition shared caches implicitly. NUcache[39] improves shared cache performance by only retaining blocks accessed by selected PCs. Vantage[40] partitions shared caches at cache block granularity to make it applicable in many-core systems. PriSM[41] partitions shared caches by controlling the eviction probabilities of different cores. There are also some studies that focus on improving the fairness or QoS of shared cache[42-44]. Although we mainly focus on improving performance in this paper, our policy can also be extended to improve fairness or QoS.

Miss count based cache management policies implicitly assume that all cache misses are equally important. However, the penalty of different misses can change dramatically in modern systems. Thus, only reducing the miss count is not enough, and it is important for cache management policies to take into account the variation of miss penalty.

2.2 Miss Penalty Based Policies

Jeong and Dubois first proposed to take into account the miss cost in cache management[13,45-46]. They use two static miss costs to represent the miss cost to a local memory and the miss cost to a remote memory respectively in CC-NUMA multiprocessors, and several replacement policies are proposed to extend LRU to consider the miss cost. In uniprocessor environment, Jeong *et al.* proposed to distinguish the miss penalty between load and store misses[14]. However, they do not distinguish the miss penalty between different load misses.

Critical cache[15] and LACS[17-18] both estimate the miss penalty using the number of issued instructions during the cache miss. Critical cache dedicates a part of cache to preserve critical loads. LACS replaces blocks with small miss penalty preferentially, and it can dynamically adapt to different applications and execution phases. Issued instruction number is more related to the processor performance. However, when memory access latency is long enough, the number of issued instruction will show no difference, and thus using it to estimate miss penalty is not accurate. Besides, such information is difficult to be obtained by the last-level cache. On the other hand, our method uses cache serving time to estimate miss penalty, and thus it does not need to obtain additional information from the processor.

Qureshi *et al.* proposed to take into account the variation of miss penalty due to the ability of serving multiple misses in parallel in modern processors, which is called the memory level parallelism (MLP) cost[16]. They use the reciprocal of the in-flight miss number to represent the MLP cost. On replacement, the MLP-aware replacement policy selects the block with the minimum weighted sum of MLP cost and LRU stack position as the victim. However, their method does not distinguish fetch and load misses with store misses and is only applied in uniprocessors. MLP-DCP[47] and MCFQ[48] partition shared cache based on the MLP cost, and their MLP cost computation methods are similar to that of [16].

Compared to our work, these previous miss penalty based policies only take into account the miss penalty on the insertion of blocks, and they do not consider the reduced penalty on the hits of cache blocks. Therefore, these policies cannot make replacement decisions based on miss penalty information alone, and they also need recency information provided by LRU. On the other hand, our policy takes into account both the penalty on cache misses and the saved penalty on cache hits. Therefore, our policy can make replacement decisions based on penalty information alone, and it does not need any recency information.

2.3 Memory Aware Policies

In the main memory, the access latency of read requests can increase due to conflicts caused by write-

back and other requests. To address this problem, some recently proposed polices schedule writebacks earlier or combine multiple writeback requests to reduce the interference introduced by writebacks[49-52]. Lee *et al.* proposed to take advantage of the memory characteristics[53]. These policies can reduce the memory access latency of read requests. Our policy can cooperate with these policies to improve performance further. However, it is beyond the scope of this paper, and we leave it as part of our future work.

3 Retention Benefit

In order to reduce the total processor stall time on cache misses, caches should retain blocks that may hurt the performance most on misses. Therefore, we propose the notion of *retention benefit* to evaluate the reduction of processor stall cycles on a cache access if the accessed block is resident in the cache. In the rest of this section, we will introduce how to compute the retention benefit for cache miss and hit requests in modern superscalar processors respectively. Then the next section will introduce how to compute the aggregate retention benefits of cache blocks and use them to guide cache replacement.

3.1 Retention Benefit Computation for Cache Misses

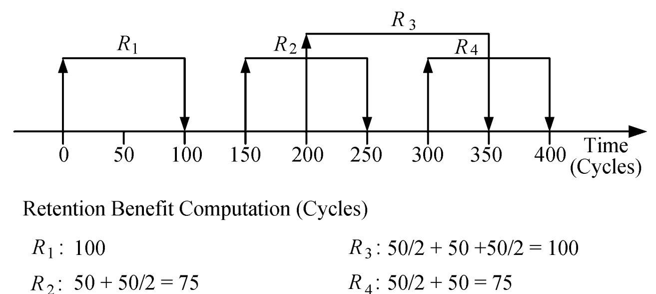
The retention benefit of a missing block reflects the increment of processor stall cycles on its miss. We describe how to compute the retention benefit for cache misses based on the request type.

Data Writeback. Writeback requests are generated by inner caches on dirty block replacement. They do not affect the processor performance, and thus their retention benefits are set to 0.

Data Store. Modern processors usually employ a structure called *store queue* to keep in-flight store instructions, and store instructions do not access caches until they retire from the processor. As a result, store misses hurt the performance only when the store queue is full, which occurs rarely. Therefore, the retention benefits of store misses are set to 1 to indicate their small performance influence.

Instruction Fetch and Data Load. We use a method similar to the MLP cost computation method proposed by Qureshi *et al.*[16] for computing retention benefits of instruction fetch and data load misses. After a long-latency fetch or load miss (e.g., last-level cache miss), the processor will run out of resources and stall soon. Therefore, the processor stall cycles on a miss can be approximated by the number of cycles that the cache spends on serving the miss. If multiple misses occur in parallel, the cache serving cycles can be divided equally for all in-flight misses.

To compute the retention benefits for fetch and load misses, we extend the MSHR (miss status holding register)[11] existing in current cache design. MSHR is used to record in-flight cache misses, and each miss is allocated an MSHR entry before the miss request is sent to memory. We append each MSHR entry with a field RB to compute its retention benefit. When an MSHR entry is allocated on a fetch or load miss, its RB is initialized to 0. Then, its RB is increased by 1/ (the number of concurrent fetch and load misses) every cycle, which indicates that all concurrent fetch and load misses share the responsibility for this stall cycle. We use a counter $C_{\text{load_miss}}$ to count the number of concurrent fetch and load misses. When the memory returns data to an MSHR entry, its RB represents the retention benefit of the recorded miss request. Fig.1 presents an example for computing the retention benefit, where R_1, R_2, R_3, and R_4 denote different fetch and load miss requests.



Retention Benefit Computation (Cycles)

R_1: 100

R_2: 50 + 50/2 = 75

R_3: 50/2 + 50 +50/2 = 100

R_4: 50/2 + 50 = 75

Fig.1. Example for retention benefit computation of instruction fetch and data load misses.

To compute $1/C_{\text{load_miss}}$, a naive implementation will need floating-point operations. We convert complex floating-point operations to fixed-point operations using a 64-entry lookup table (the maximum possible value of $C_{\text{load_miss}}$ is 64 in our experiments). Besides, as discussed in [16], it is not necessary to associate an adder with each MSHR entry for its retention benefit computation, and the time sharing four adders has a negligible impact on the value of retention benefit.

Compared to the original method proposed by Qureshi *et al.*[16], our method can distinguish the types of different requests; while their method treats all types of requests identically, which makes it less accurate.

The retention benefit of a miss request depends on the current situation of other cache requests. For instance, if request D converts to a hit in Fig.1, the retention benefit of request C will increase. However, Qureshi *et al.* have demonstrated that in most cases, the situations of other cache requests are similar for two

requests to the same block[16]. Therefore, previous retention benefits of a block can be used to predict future ones of the same block.

Prefetch. Prefetch misses do not stall the processor. However, they may be valuable since subsequent demand requests may use prefetched data. Thus, the retention benefits of prefetch misses are set to 1.

For demand misses, prefetch misses can affect their retention benefits in two ways. One is that for a prefetch request which is not timely, the following demand request to the same block cannot get its data on time and misses in the cache, but with a relative small retention benefit. In such scenarios, we only need to reset the *RB* of the corresponding MSHR entry to 0 and restart the retention benefit computation when the address of an in-flight prefetch request is found to be identical to that of the current demand request. The other way is that prefetch requests contend with demand requests for the memory bandwidth, which will eventually affect the memory access latency of demand requests. However, our method for retention benefit computation takes into account variable memory latency. Consequently, our retention benefit computation method is applicable in the presence of prefetching.

3.2 Retention Benefit Computation for Cache Hits

As discussed in Section 1, it is not enough to only compute retention benefits for cache misses, and it is important to also compute retention benefits for cache hits. The retention benefit on a cache hit can be viewed as the increment of processor stall cycles when the hit converts to a miss.

Data Writeback and Store. We use the same method as that in the miss situation, since the performance impact is negligible on their misses.

Prefetch. The retention benefit of a prefetch hit is set to 0 since the prefetched block is already in the cache, and thus the prefetch request provides no benefit.

Instruction Fetch and Data Load. Computing retention benefits on instruction fetch and data load hits is much more challenging compared with that on their misses. Therefore, we pretend the hit request is a miss and then use the retention benefit computation method for fetch and load misses. At first, we need to estimate the memory access latency when the hit request converts to a miss. Since we model a desktop-like system in our experiments, where the variation of memory latency is not very large, we use a static estimated memory access latency (200 cycles) in our experiments. For more complex systems where memory latency can change dramatically, a small address based memory latency predictor can be potentially employed, since cache block address usually determines its location in the memory system, and thus roughly determines its access latency.

With the estimated memory access latency, we also need the information about other concurrent fetch and load misses to compute the retention benefit. To simplify the computation, we use the current number of fetch and load misses ($C_{\rm load_miss}$) plus 1 (the pretended miss itself) to approximate the average concurrent miss number during the serving time of the pretended miss. Consequently, the retention benefit of instruction fetch or data load hit can be computed using the following formula:

$$\text{retention benefit} = \frac{\text{estimated memory latency}}{C_{\rm load_miss} + 1}.$$

A more accurate method to compute retention benefits for fetch and load hits is to design a structure HSHR (Hit Status Holding Register), which is similar to the MSHR. On each fetch or load hit, an HSHR entry is allocated to compute its retention benefit using the same method as that of MSHR. However, our analysis shows that the performance of this method is similar to that of the simple method above, and using HSHR also incurs extra hardware cost. Therefore, we use the simpler method in this paper.

3.3 Analysis of Retention Benefit

Using the method described above, we analyze retention benefits of cache misses for SPEC CPU2006 benchmarks in an LRU-managed 2MB L2 cache without prefetching. Fig.2 presents the retention benefit distribution for some representative programs. The left bars in the histogram represent the retention benefit distribution of fetch and load misses, while the rightmost bar represents the percentage of store misses, whose retention benefits are always 1. There is no writeback miss because we model an inclusive cache hierarchy.

For programs with high retention benefits, such as *milc* and *libquantum*, most of their misses are fetch and load misses and occur in isolation. For programs with middle retention benefits, such as *zeusmp* and *omnetpp*, some of misses have retention benefits equal to memory access latency, while the others have small retention benefits. For programs with low retention benefits, most of their misses occur in parallel and their memory latency is hidden by that of other concurrent misses. We observe that for programs with middle or low retention benefits, the disparity in retention benefits of different misses is large, and thus it is more important for the cache replacement policy to take into account the variation of retention benefits for these programs.

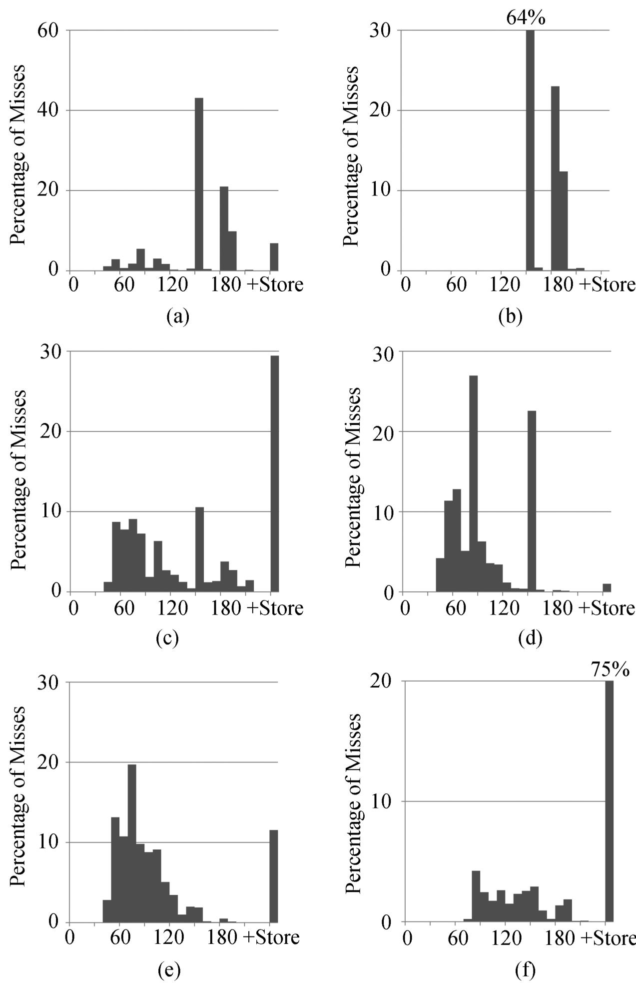


Fig.2. Retention benefit distribution of LRU for representative programs in SPEC CPU2006. (a) *milc*. (b) *libquantum*. (c) *zeusmp*. (d) *omnetpp*. (e) *mcf*. (f) *lbm*. (a) and (b) are with high retention benefits. (c) and (d) are with middle retention benefits. (e) and (f) are with low retention benefits.

Moreover, in multi-core systems where last-level caches are shared, the variation of retention benefits exists not only between different requests from the same core, but also between requests from different cores. For instance, when *libquantum* and *lbm* execute in parallel, since the misses of *libquantum* have larger retention benefits, more cache space should be allocated to it. In doing so, although the miss count of *lbm* increases, its performance loss is limited since extra misses have small retention benefits.

Our results above motivate the need for a retention benefit based cache replacement policy, which can deal with the retention benefit variation of different misses.

4 Retention Benefit Based Replacement

Our goal is to calculate the aggregate retention benefit for each cache block, and then design a cache replacement policy that focuses on reserving blocks with larger aggregate retention benefits to improve performance. To that end, we propose retention benefit based replacement (RBR).

4.1 Aggregate Retention Benefit Computation

To enable retention benefit based cache replacement, it is necessary to calculate and keep track of aggregate retention benefits for all cache blocks. Therefore, RBR associates each cache block with an M-bit retention benefit value (RBV) to record its aggregate retention benefit. The RBV of 0 indicates the minimal aggregate retention benefit, while the RBV of $2^M - 1$ indicates the maximum aggregate retention benefit.

On each cache miss and hit, the computed retention benefit using the method in Section 3 is accumulated to the RBV of the corresponding block. If a block has a large RBV, it indicates that the processor will stall for more cycles if that block is not resident in the cache, and thus the cache should retain blocks with larger RBVs preferentially.

As introduced in Section 3, the value of retention benefit can be hundreds of cycles, and keeping accurate aggregate retention benefits requires large storage overhead and is also not necessary. In the real implementation, the retention benefit is quantized to a number between 0 and the maximum promotion number *MAX_PRO*. Table 1 shows the retention benefit quantization when $MAX_PRO = 3$. Then, the quantized promotion number is accumulated to the RBV. After accumulation, if the RBV is larger than the maximum value of RBV ($2^M - 1$), the RBV is saturated to $2^M - 1$.

Table 1. Retention Benefit Quantization When $MAX_PRO = 3$

Retention Benefit (Cycles)	Quantized Promotion Number
0	0
1∼90	1
91∼180	2
180+	3

When the width of RBV is fixed, *MAX_PRO* should be large enough to distinguish accesses with different retention benefits. On the other hand, *MAX_PRO* cannot be too large because it will limit the ability of RBV to record aggregate retention benefits for more cache accesses. We study how to select the width of RBV and the value of *MAX_PRO* in Section 6.

4.2 Static RBR

To improve cache performance by retaining blocks with large retention benefits preferentially, we propose static RBR (SRBR) to select blocks with the smallest RBVs as the victim.

On a cache miss, SRBR selects the block with the minimum RBV in the corresponding cache set as the victim block. If there are multiple blocks with the same minimal RBVs, SRBR breaks the tie by always searching for the victim from a fixed position (from the leftmost block in this paper). If the minimal RBV is larger than 0, SRBR decreases the RBVs of all blocks in the corresponding cache set by the minimal RBV. In doing so, SRBR can remove stale blocks from the cache, which have accumulated large RBVs in the past, but have not been accessed recently. Thus, decreasing RBVs of all block in a cache set allows SRBR to adapt to changes in the working set.

After the victim selection, SRBR inserts the new block into the cache and updates its RBV. SRBR uses the method mentioned in Subsection 4.1 to calculate and quantize the retention benefit of the new incoming block. Then, the RBV of the incoming block is initiated to the quantized promotion number.

On a hit to a block, SRBR updates the RBV of the hit block. We also use the same method to calculate and quantize the retention benefit. At last, the quantized promotion number is accumulated to the RBV of the hit block. Fig.3 shows an example of the working process of SRBR.

Step	Next Access	Quantized Promotion Number	RBV ↓				
1	B	4	A 0	B 3	C 1	D 4	Hit
2	E	2	A 0	B 7	C 1	D 4	Miss, Replace A
3	F	3	E 2	B 7	C 1	D 4	Miss, Replace C
			E 1	B 6	F 3	D 3	

Fig.3. Example of SRBR working process.

Specifically, when $MAX_PRO = 1$, i.e., all accesses are considered to have the same retention benefits, SRBR downgrades to an LFU-like (Least Frequently Used) policy, since the RBV of a block records its accessed time in such situations. When the width of RBV $RBV_width = 1$ and $MAX_PRO = 1$, SRBR is identical to NRU (Not Recently Used), which is a pseudo LRU policy widely used in modern processors[54-55].

4.3 Dynamic RBR

The limitation of SRBR is that it cannot deal with thrashing access patterns. The following cache access sequence shows a typical thrashing access pattern:

$$(a_1, a_2, a_3, \ldots, a_n)^I.$$

It denotes that an access sequence from block a_1 to block a_n repeats I times ($n >$ cache size), and we assume that $a_1, \ldots, a_n$ have the same retention benefits. In such scenarios, the working set is larger than the cache size, and SRBR will cause cache thrashing and result in no cache hits.

To avoid cache thrashing, we propose Bimodal RBR (BRBR) to address thrashing access pattern. On cache misses, BRBR sets the RBVs of most incoming blocks to the minimal RBV value 0, and infrequently sets the RBVs of incoming blocks to the quantized retention benefit value as SRBR. By setting the RBVs of most incoming blocks to 0, BRBR can prevent new incoming blocks from evicting blocks resident in the cache to keep a part of working set in the cache steadily, and thus the retained part of working set can receive hits on accesses. Besides, infrequently setting the RBVs of incoming blocks as SRBR makes BRBR adaptive to the working set changes. BRBR is similar to the Bimodal Insertion Policy (BIP) component of DIP[3].

Our experiments show that only setting the RBVs of 1/64 incoming blocks as SRBR is enough. To implement it, the last-level cache only needs to add a 6-bit counter which is added by 1 on each miss. When the counter overflows, BRBR sets the RBV of the incoming block as SRBR. Otherwise, the RBV is set to 0.

On the other hand, BRBR can hurt the performance for non-thrashing access patterns. In order to improve performance for all kinds of access patterns, we propose dynamic RBR (DRBR) to dynamically determine which policy suits best for the current cache access pattern, SRBR or BRBR. DRBR uses set dueling[3] to compare the performance of SRBR and BRBR.

Set dueling is widely used to compare the performance of two competitive replacement policies[3,6]. It permanently dedicates a few fixed number of sets (called leader sets) to each policy and compares the miss number of two groups of leader sets. Then, set dueling uses a saturating policy selection (PSEL) counter to record which group of leader sets incurs less misses, and thus selects its policy as the winning policy. The winning policy is applied to the remaining sets (called follower sets) of the cache as their replacement policy.

Instead of deciding the winning policy by comparing their total miss number, DRBR extends set dueling to decide the winning policy by comparing the total retention benefits of misses of their leader sets with a PSEL counter. PSEL is initiated to 0. On a miss in the leader sets of SRBR, PSEL is decreased by the quantized retention benefit of the miss. While on a miss in the leader sets of BRBR, PSEL is increased by the quantized retention benefit of the miss. If PSEL $\geqslant 0$, it indicates that the misses of BRBR leader sets have larger aggregate retention benefits and thus stall the processor for more cycles. As a result, SRBR is chosen as the winning policy and applied to all remaining sets in the cache. Otherwise, if PSEL < 0, it indicates SRBR

incurs more performance loss and BRBR is chosen as the winning policy. In our experiments, we use a 12-bit PSEL.

4.4 Thread-Aware Dynamic RBR

In multi-core processors, last-level caches are usually shared by all cores. Since concurrently executing programs can show different cache access patterns and thus have different favorite replacement policies between SRBR and BRBR, we extend DRBR to Thread-Aware DRBR (TADRBR), which is similar to the extension from DIP[3] to Thread-Aware DIP (TADIP)[38]. TADRBR chooses the best suited replacement policy for each core. TADRBR dedicates two groups of leader sets for each core to determine which policy the core should use in the presence of cache requests of other cores.

Like TADIP, TADRBR requires one PSEL per core to be thread-aware. Besides, to compute retention benefits for requests of each core, we assign a $C_{\rm load_miss}$ for each independent core. On a cache access from a specific core, the $C_{\rm load_miss}$ of that core is used to compute the retention benefit. The hardware overhead of TADRBR is negligible.

4.5 Design Issues for Inclusive Caches

In our experiments, we evaluate the performance of various techniques in an inclusive L2 cache. In an inclusive cache hierarchy, when a cache block is evicted from the L2 cache, the same block must be evicted by L1 caches to satisfy the inclusion property, and these evicted L1 cache blocks as a result of inclusion are called inclusion victims[56]. Inclusion victims may show good locality in L1 caches, and thus evicting them early can hurt performance[56].

To eliminate inclusion victims, the L2 cache replacement policy should avoid replacing blocks resident in L1 caches. In the typical implementation of a two-level inclusive cache hierarchy, each L2 cache block is associated with per-core tracking bits that denote which cores are caching the block. Therefore, we can use the tracking bits of a block to decide whether it is retained by any L1 cache. However, the information provided by tracking bits is inaccurate because L1 caches silently drop evicted clean blocks. To address this problem, on the replacement of a clean block in the L1 cache, it sends an explicit eviction notification to the L2 cache, and thus the L2 cache can update the tracking bits of the corresponding block[57].

Upon replacement, the L2 cache preferentially selects the victim among blocks which are not resident in L1 caches. If there are no such blocks, the replacement policy will select the victim block from all blocks in the cache set. This method is similar to a recently proposed inclusive cache replacement policy[58]. To make a fair comparison, we evaluate the performance of all techniques using this extension in our experiments. Our experiments show that the performance of all techniques can improve slightly with this extension, and the increasing traffic due to explicit eviction notifications is very small.

5 Experimental Methodology

5.1 Simulator

The simulator we used is gem5[59]. The microarchitecture parameters of the simulator are shown in Table 2, and the configuration of the processor is similar to the Intel Nehalem①. The simulator models a two-level inclusive cache hierarchy using the MESI coherence protocol. The L1 caches are private to each core, while the L2 cache is shared by all cores. In single-core configuration, the L2 cache is 16-way 2 MB. In multi-core configuration, the L2 cache is 4MB for 4-core configuration and 8MB for 8-core configuration. The simulator also models a hardware stream prefetcher for each core, and prefetched blocks are inserted into both the L1 and the L2 caches.

Table 2. Parameters of the Simulator

Parameter	Configuration
Processor	4-wide, 128-entry ROB, 48-entry load queue, 32-entry store queue
L1 ICache	32 KB, 64 B block size, 4-way, 3-cycle hit latency, PLRU
L1 DCache	32 KB, 64 B block size, 4-way, 3-cycle hit latency, PLRU, 32-entry MSHR
L2 cache	2 MB/4 MB/8 MB, 64 B block size, 16-way, 16-cycle hit latency, 64-entry MSHR
Memory	1 channel, 2 dimms, 2 ranks per dimm, 8 banks per rank, bank conflicts modeled, 150-cycle minimal access latency

5.2 Benchmarks

We use SPEC CPU2006 benchmarks[60] with the first reference inputs to do evaluation. SimPoint[61] is used to obtain a single representative 200 million instructions for each benchmark. Among 29 SPEC CPU2006 benchmarks, seven benchmarks cannot be addressed by our simulation infrastructure. For the remaining ones, *gamess*, *namd*, *povray*, *calculix*, *h*264*ref*, and *wrf* are not evaluated because their working sets are very small and their MPKI (misses per kilo instruc-

①Intel® Core™ i7 processor. http://www.intel.com/products/processor/corei7/, Sept. 2014.

tions) is less than 0.1 in a 2 MB L2 cache under LRU. The rest of 16 benchmarks are used in our experiments.

For the performance evaluation in multi-core environment, we choose several benchmarks out of the 16 selected SPEC CPU2006 benchmarks at random to combine into a multi-core workload. Totally, we create 20 4-core workloads and eight 8-core workloads. We quantify the performance in multi-core environment with the following three widely used metrics:

$$\text{weighted speedup} = \sum_{i=1}^{n} \frac{IPC_i}{SingleIPC_i},$$

$$\text{throughput} = \sum_{i=1}^{n} IPC_i,$$

$$\text{fair speedup} = n / \sum_{i=1}^{n} \frac{SingleIPC_i}{IPC_i},$$

where n is the number of cores, IPC_i represents the number of instructions per cycle of program i, and $SingleIPC_i$ represents the number of instructions per cycle of i when it runs alone.

6 Results and Analysis

6.1 Single-Core Workloads

At first, we evaluate the performance of SRBR in the absence of prefetching. Fig.4 studies the sensitivity of SRBR to the width of RBV (RBV_width) and the maximum promotion number MAX_PRO. The x-axis represents the value of MAX_PRO under different widths of RBV, and the y-axis represents the geometric mean speedup of 16 single-core workloads. The speedup is computed by dividing the IPC of SRBR by that of LRU. When $MAX_PRO = 1$, all accesses are considered to have the same retention benefits, and thus SRBR has no performance improvement. When MAX_PRO is close to its maximum value ($2^{RBV_width} - 1$), it limits the ability of RBV to record the aggregate retention benefit for more cache accesses. The results show that when MAX_PRO equals 3, SRBR achieves the best performance improvement. They also show that using 3-bit RBVs is enough. Therefore, RBV_width is set to 3 and MAX_PRO is set to 3 in the following experiments. We also observe that even using 2-bit RBVs and MAX_PRO of 3 can achieve a geometric mean speedup of 3.8%.

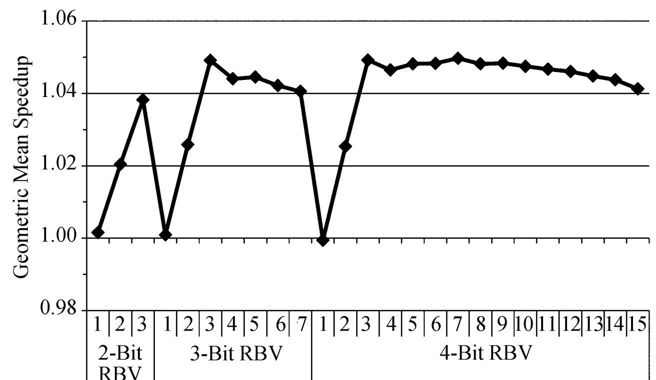


Fig.4. Sensitivity to the RBV width and the MAX_PRO value for RBR.

Then, we compare the performance of SRBR and DRBR with other state-of-the-art techniques in the absence of prefetching. Besides SRBR and DRBR, we investigate the performance of other techniques including MLP-aware replacement (MLP)[16], DIP[3], SRRIP[6], and DRRIP[6]. Since RBR uses 3-bit RBVs, we also use SRRIP and DRRIP with 3-bit RRPVs in our experiments to make their comparison fair, and our experiments show that their 3-bit versions perform slightly better than the 2-bit versions.

Figs. 5~6 show MPKI and IPC both normalized to LRU for various techniques respectively. The geometric

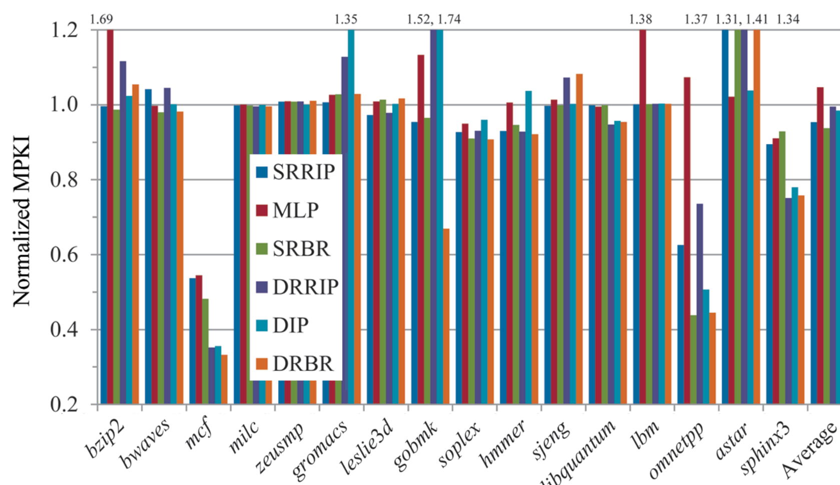


Fig.5. Normalized MPKI for various techniques in the absence of prefetching.

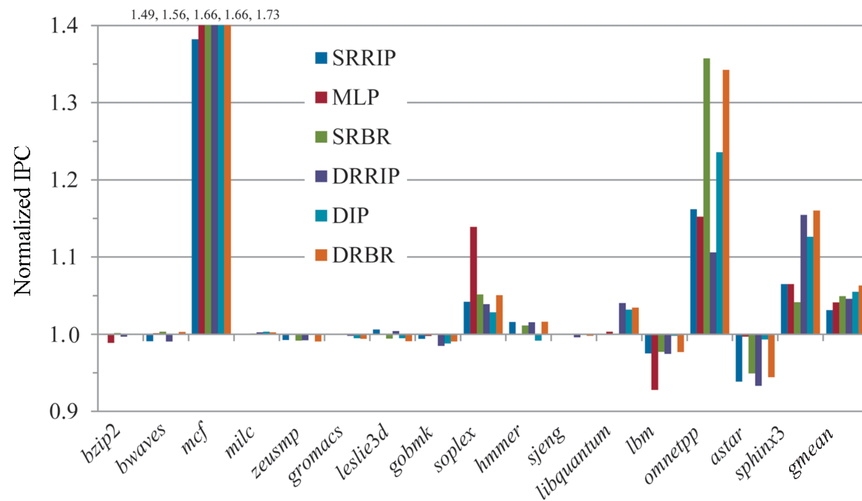


Fig.6. Normalized speedup for various techniques in the absence of prefetching.

mean speedup of SRBR and DRBR is 4.9% and 6.3% respectively, while it is 3.1% for SRRIP, 4.1% for MLP, 4.6% for DRRIP, and 5.5% for DIP. Among these benchmarks, *mcf* and *sphinx*3 show thrashing access patterns. Therefore, DRBR chooses BRBR dynamically and outperforms SRBR for these two benchmarks. For the remaining benchmarks, SRBR is mainly chosen by DRBR. Our results show that SRBR outperforms other static cache replacement policies including LRU, SRRIP, and MLP, while DRBR outperforms all other static and dynamic policies. We observe that although DRBR increases the MPKI of *bzip*2, it can improve the IPC of *bzip*2 slightly since the increasing misses of DRBR have small retention benefits. For *lbm* and *astar*, their retention benefits are unpredictable and only LRU performs well for them. Thus, most other policies degrade their performance compared with LRU. DIP does not degrade their performance since it can dynamically switch to LRU. However, the performance loss of RBR for these programs is limited, and RBR requires less storage compared with LRU as we will show. Thus, we conclude RBR also suits for LRU friendly programs.

To illustrate why RBR can improve cache performance, Fig.7 compares the retention benefit distribution of cache misses under LRU, DIP, and DRBR for *omnetpp*. The retention benefit distribution of DIP and DRBR is both normalized to the miss number of LRU. Compared with LRU, both of them can reduce the aggregate miss count significantly. While DIP reduces more misses with small retention benefits compared with DRBR, DRBR can reduce more misses with large retention benefits. Hence, DRBR can achieve more performance gain compared with DIP. The retention benefit distribution of SRBR shows similar results.

We also study the performance of SRBR and DRBR in the presence of prefetching. Fig.8 shows the speedup when prefetching is enabled. The speedup is normalized to that of LRU in the presence of prefetching. SRBR and DRBR reduce average MPKI by 4.9% and 7.6% and achieve a geometric mean (*gmean*) speedup of 5.6% and 6.7% respectively, and DRBR also outperforms other techniques. Compared with LRU without prefetching, DRBR outperforms it by 22.2%, while LRU with prefetching outperforms it by 14.5%. These experiments show that SRBR and DRBR can also improve cache performance in the presence of prefetching.

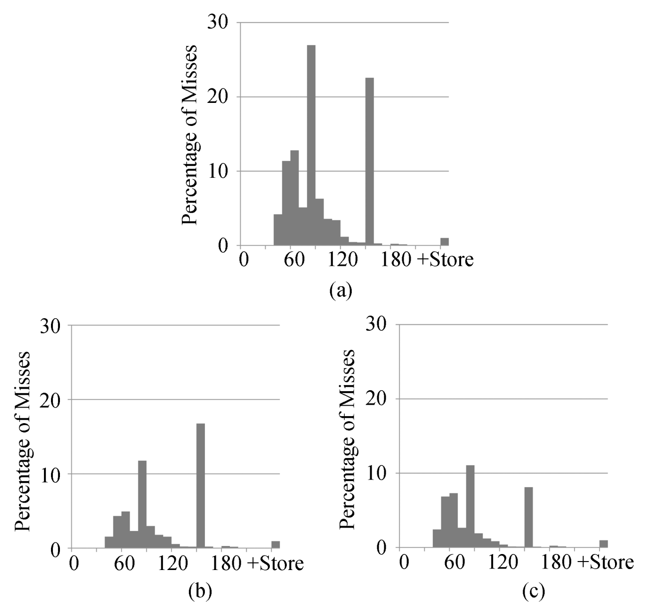


Fig.7. Retention benefit distribution of LRU, DIP, and DRBR for *omnetpp*. (a) LRU. (c) DIP. (c) DRBR.

6.2 Multi-Core Workloads

In multi-core environment, TADRBR has more performance potential since it can deal with the variation of retention benefits between different concurrent programs. On multi-core workloads, we compare the performance of TADRBR with other state-of-the-art shared cache management policies including TADIP[38], TADRRIP, UCP[2], and PIPP[5]. TADIP and TADRRIP are the thread-aware versions of DIP and DRRIP respectively.

Fig.9 presents the weighted speedup normalized to LRU for various techniques on 20 4-core workloads in the absence of prefetching, where mix_i represents the i-th multi-core workload. Compared with LRU, TADRBR achieves a geometric mean weighted speedup of 5.3%, while it is 2.0% for PIPP, 3.1% for UCP, 3.9% for TADRRIP, and 4.0% for TADIP. TADRBR only degrades the performance of three workloads, and their performance loss is all less than 0.3%.

The results on the metrics of throughput and fair speedup are similar to those on weighted speedup. TADRBR can achieve a normalized throughput improvement of 5.5% on geometric mean, and the geometric mean fair speedup of TADRBR is 8.0%. TADRBR also outperforms other state-of-the-art techniques on the metrics of throughput and fair speedup. The performance improvement of TADRBR on fair speedup is the most, since TADRBR rarely causes performance degradation for individual program in a multi-core workload.

Fig.10 shows the normalized weighted speedup on 4-core workloads when prefetching is enabled. Compared with LRU with prefetching, the weighted speedup improvement is 5.9% for TADRBR, while it is 4.2% for TADIP which performs the best among the rest of techniques. TADRBR outperforms the other techniques significantly. The results on throughput and fair speedup are similar.

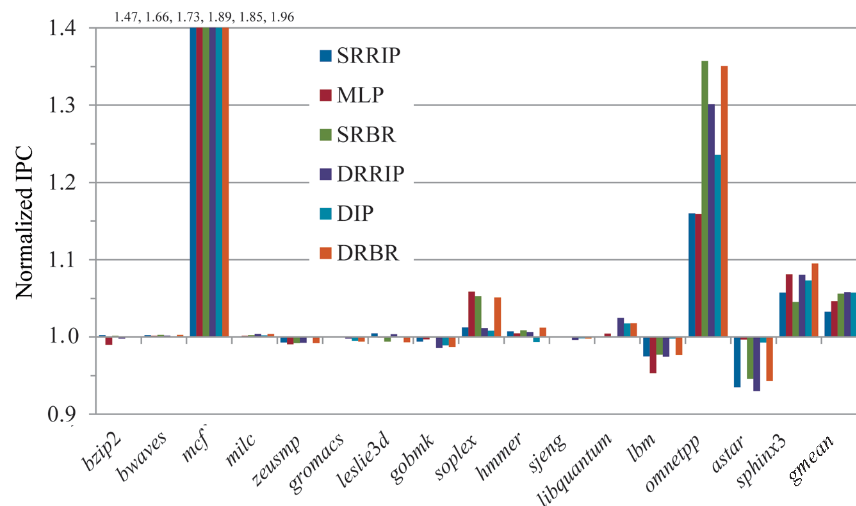


Fig.8. Normalized speedup for various techniques in the presence of prefetching.

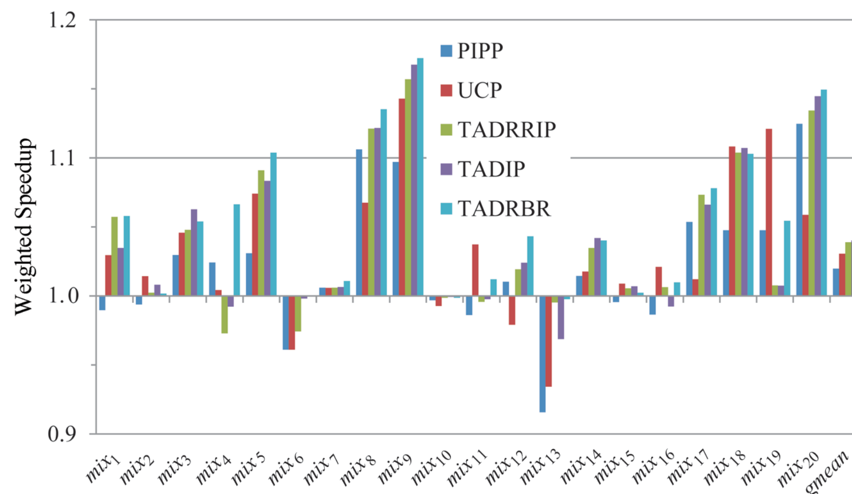


Fig.9. Normalized weighted speedup for 4-core workloads in the absence of prefetching.

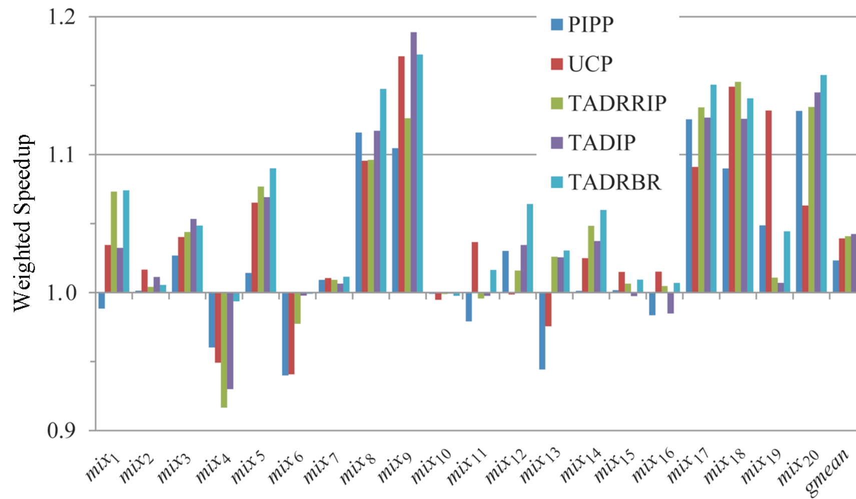


Fig.10. Normalized weighted speedup for 4-core workloads in the presence of prefetching.

To study the scalability of TADRBR with the number of cores, we also evaluate the performance of TADRBR on 8-core workloads. Fig.11 presents the normalized weighted speedup for 8-core workloads in the absence of prefetching. Compared with LRU, the weighted speedup improvement of TADRBR is 4.5%, which is roughly 1.5 times of the improvement of other techniques in which TADIP performs the best and outperforms LRU by 3.0%. With the increment of core number, the retention benefit difference from different cores also increases, and TADRBR thus has significantly better performance.

In summary, due to the variation of retention benefits between different cores, TADRBR is effective in multi-core environment.

6.3 Storage Overhead

Table 3 shows the storage overhead of various techniques for the 4 MB L2 cache used in the 4-core configu-

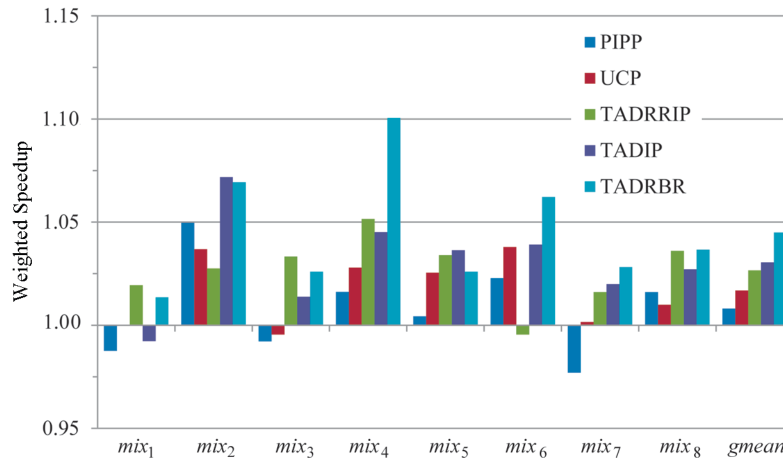


Fig.11. Normalized weighted speedup for 8-core workloads in the absence of prefetching.

Table 3. Storage Overhead of Various Techniques for the 16-Way 4 MB L2 Cache in 4-Core Configuration

	Weighted Speedup	Weighted Speedup with Prefetching	Storage per Block (Bit)	Extra Storage	Total (KB)
LRU	1.000	1.180	4	0	32.00
PIPP	1.020	1.201	4	5 KB	37.00
UCP	1.031	1.224	6	5 KB	53.00
TADRRIP	1.039	1.221	3	40 bits	24.00
TADIP	1.040	1.223	4	40 bits	32.00
TADRBR	1.053	1.241	3	0.12 KB	24.12

Note: the weighted speedup is normalized to that of LRU without prefetching.

ration. The storage overhead of TADRBR comes from the counters for retention benefit computation, per-block RBV, and PSELs. To compute the retention benefit, each core requires a 9-bit $C_{\text{load_miss}}$, and each MSHR entry requires a 14-bit RB. Each cache block needs three bits to record its RBV. To be thread-aware, each core needs a 12-bit PSEL. It totally consumes $(9 \times 4 + 14 \times 64 + 3 \times 65\,536 + 12 \times 4)$ bits $= 24.12$ KB of extra storage to implement TADRBR, which is roughly 0.6% of the total storage of a 4 MB LLC. Compared with other recent proposals except TADRRIP, TADRBR has a significant lower storage overhead. The storage overhead of TADRRIP is similar to that of TADRBR. The storage overhead of DRBR in single-core configuration is also very low.

7 Conclusions

Compared to the aggregate cache miss count, the aggregate cache miss penalty is more related to the system performance. In modern systems, memory access latency is variable, and processors adopt techniques such as non-blocking caches and prefetching to tolerate memory access latency. As a result, cache miss penalty can change dramatically, which motivates the need for the cache replacement policy to be aware of the variation of miss penalty. This paper addresses this problem by making the following contributions.

1) We proposed the notion of retention benefit to represent the reduction of processor stall cycles when a block is reserved by the cache, and we also proposed a simple method for its computation. The retention benefit can evaluate not only the performance loss on cache misses, but also the performance gain due to cache hits.

2) We proposed Static Retention Benefit Based Replacement (SRBR), which selects the block with the minimum aggregate retention benefit as the victim. In doing so, SRBR retains blocks with larger aggregate retention benefits in the cache.

3) We proposed dynamic retention benefit based replacement (DRBR). DRBR uses set dueling to dynamically select the best suited policy between SRBR and BRBR (bimodal RBR), which is designed to deal with thrashing access patterns. DRBR can also be thread-aware with only several additional counters.

Our evaluation shows that RBR improves cache performance for both single-core and multi-core workloads, no matter whether prefetching is enabled or not.

To the best of our knowledge, RBR is the first pure penalty-based cache replacement policy without depending on any other information. RBR is applied in a desktop-like system in this paper. In future systems, cache miss penalty can change more dramatically, and thus it is even more important to deal with the variation of miss penalty. To apply RBR in such systems is part of our future work. Besides, since memory aware policies can reduce memory access latency as discussed in Subsection 2.3 and RBR is aware of the changes of memory access latency, they can cooperate with each other to improve performance further. Thus, to apply RBR with memory aware policies is also part of our future work.

References

[1] Lai A C, Fide C, Falsafi B. Dead-block prediction & dead-block correlating prefetchers. In *Proc. the 28th ISCA*, Jun. 2001, pp.144-154.

[2] Qureshi M K, Patt Y N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. the 39th MICRO*, Dec. 2006, pp.423-432.

[3] Qureshi M K, Jaleel A, Patt Y N, Steely Jr S C, Emer J. Adaptive insertion policies for high performance caching. In *Proc. the 34th ISCA*, Jun. 2007, pp.381-391.

[4] Kharbutli M, Solihin D. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 2008, 57(4): 433-447.

[5] Xie Y, Loh G H. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. the 36th ISCA*, Jun. 2009, pp.174-183.

[6] Jaleel A, Theobald K B, Steely Jr S C, Emer J. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. the 37th ISCA*, Jun. 2010, pp.60-71.

[7] Khan S M, Tian Y, Jimenez D A. Sampling dead block prediction for last-level caches. In *Proc. the 43rd MICRO*, Dec. 2010, pp.175-186.

[8] Wu C J, Jaleel A, Hasenplaugh W, Martonosi M, Steely Jr S C, Emer J. SHiP: Signature-based hit predictor for high performance caching. In *Proc. the 44th MICRO*, Dec. 2011, pp.430-441.

[9] Xie Y. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design & Test of Computers*, 2011, 28(1): 44-51.

[10] Loh G H, Hill M D. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proc. the 44th MICRO*, Dec. 2011, pp.454-464.

[11] Kroft D. Lockup-free instruction fetch/prefetch cache organization. In *Proc. the 8th ISCA*, May 1981, pp.81-87.

[12] Vanderwiel S P, Lilja D J. Data prefetch mechanisms. *ACM Comput. Surv.*, 2000, 32(2): 174-199.

[13] Jeong J, Dubois M. Optimal replacements in caches with two miss costs. In *Proc. the 11th SPAA*, Jun. 1999, pp.155-164.

[14] Jeong J, Stenström P, Dubois M. Simple penalty-sensitive replacement policies for caches. In *Proc. the 3rd CF*, May 2006, pp.341-352.

[15] Ju R D C, Lebeck A R, Wilkerson C. Locality vs. criticality. In *Proc. the 28th ISCA*, Jun. 2001, pp.132-143.

[16] Qureshi M K, Lynch D N, Mutlu O, Patt Y N. A case for MLP-aware cache replacement. In *Proc. the 33rd ISCA*, Jun. 2006, pp.167-178.

[17] Sheikh R, Kharbutli M. Improving cache performance by combining cost-sensitivity and locality principles in cache replacement algorithms. In *Proc. the 28th ICCD*, Oct. 2010, pp.76-83.

[18] Kharbutli M, Sheikh R. LACS: A locality-aware cost-sensitive cache replacement algorithm. *IEEE Transactions on Computers*, 2013, 63(8): 1975-1987.

[19] Chaudhuri M. Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches. In *Proc. the 42nd MICRO*, Dec. 2009, pp.401-412.

[20] Keramidas G, Petoumenos P, Kaxiras S. Cache replacement based on reuse-distance prediction. In *Proc. the 25th ICCD*, Oct. 2007, pp.245-250.

[21] Wu C J, Jaleel A, Martonosi M, Steely Jr S C, Emer J. PAC-Man: Prefetch-aware cache management for high performance caching. In *Proc. the 44th MICRO*, Dec. 2011, pp.442-453.

[22] Duong N, Zhao D, Kim T, Cammarota R, Valero M, Veidenbaum A V. Improving cache management policies using dynamic reuse distances. In *Proc. the 45th MICRO*, Dec. 2012, pp.389-400.

[23] Hu Z, Kaxiras S, Martonosi M. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proc. the 29th ISCA*, May 2002, pp.209-220.

[24] Liu H, Ferdman M, Huh J, Burger D. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proc. the 41st MICRO*, Nov. 2008, pp.222-233.

[25] Jalminger J, Stenstrom P. A novel approach to cache block reuse predictions. In *Proc. the 2003 ICPP*, Oct. 2003, pp.294-302.

[26] Johnson T L, Connors D A, Merten M C, Hwu W M W. Run-time cache bypassing. *IEEE Transactions on Computers*, 1999, 48(12): 1338-1354.

[27] Rivers J A, Davidson E S. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proc. the 1996 ICPP*, Aug. 1996, Vol. 1, pp.154-163.

[28] Rivers J A, Tam E S, Tyson G S, Davidson E S, Farrens M. Utilizing reuse information in data cache management. In *Proc. the 12th ICS*, Jul. 1998, pp.449-456.

[29] John L K, Subramanian A. Design and performance evaluation of a cache assist to implement selective caching. In *Proc. the 1997 ICCD*, Oct. 1997, pp.510-518.

[30] Walsh S J, Board J A. Pollution control caching. In *Proc. the 1995 ICCD*, Oct. 1995, pp.300-306.

[31] Chi C H, Dietz H. Improving cache performance by selective cache bypass. In *Proc. the 22nd HICSS*, Jan. 1989, Vol. 1, pp.277-285.

[32] González, A, Aliagas C, Valero M. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. the 9th ICS*, Jul. 1995, pp.338-347.

[33] Tyson G, Farrens M, Matthews J, Pleszkun A R. A modified approach to data cache management. In *Proc. the 28th MICRO*, Dec. 1995, pp.93-103.

[34] Xiang L, Chen T, Shi Q, Hu W. Less reused filter: Improving L2 cache performance via filtering less reused lines. In *Proc. the 23rd ICS*, Jun. 2009, pp.68-79.

[35] Gao H, Wilkerson C. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *Proc. the 1st JWAC*, Jun. 2010.

[36] Gaur J, Chaudhuri M, Subramoney S. Bypass and insertion algorithms for exclusive last-level caches. In *Proc. the 38th ISCA*, Jun. 2011, pp.81-92.

[37] Li L, Tong D, Xie Z, Lu J, Cheng X. Optimal bypass monitor for high performance last-level caches. In *Proc. the 21st PACT*, Sept. 2012, pp.315-324.

[38] Jaleel A, Hasenplaugh W, Qureshi M, Sebot J, Steely Jr S, Emer J. Adaptive insertion policies for managing shared caches. In *Proc. the 17th PACT*, Oct. 2008, pp.208-219.

[39] Manikantan R, Rajan K, Govindarajan R. NUcache: An efficient multicore cache organization based on next-use distance. In *Proc. the 17th HPCA*, Feb. 2011, pp.243-253.

[40] Sanchez D, Kozyrakis C. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proc. the 38th ISCA*, Jun. 2011, pp.57-68.

[41] Manikantan R, Rajan K, Govindarajan R. Probabilistic shared cache management (PriSM). In *Proc. the 39th ISCA*, Jun. 2012, pp.428-439.

[42] Hsu L R, Reinhardt S K, Iyer R, Makineni S. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *Proc. the 15th PACT*, Sept. 2006, pp.13-22.

[43] Iyer R. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proc. the 18th ICS*, Jun. 2004, pp.257-266.

[44] Kim S, Chandra D, Solihin Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. the 13th PACT*, Sept. 2004, pp.111-122.

[45] Jeong J, Dubois M. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 2006, 55(4): 353-365.

[46] Jeong J, Dubois M. Cost-sensitive cache replacement algorithms. In *Proc. the 9th HPCA*, Feb. 2003, pp.327-337.

[47] Moreto M, Cazorla F, Ramirez A, Valero M. MLP-aware dynamic cache partitioning. In *Proc. the 3rd HiPEAC*, Jan. 2008, pp.337-352.

[48] Kaseridis D, Iqbal M, John L. Cache friendliness-aware management of shared last-level caches for high performance multicore systems. *IEEE Transactions on Computers*, 2014, 63(4): 874-887.

[49] Lee H H S, Tyson G S, Farrens M K. Eager writeback — A technique for improving bandwidth utilization. In *Proc. the 33rd MICRO*, Dec. 2000, pp.11-21.

[50] Lee C J, Narasiman V, Ebrahimi E, Mutlu O, Patt Y N. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical Report, TR-HPS-2010-002, High Performance Systems Group, Department of Electrical and Computer Engineering, The University of Texas at Austin & Department of Electrical and Computer Engineering, Carnegie Mellon University, April 2010.

[51] Stuecheli J, Kaseridis D, Daly D, Hunter H C, John L K. The virtual write queue: Coordinating DRAM and last-level cache policies. In *Proc. the 37th ISCA*, Jun. 2010, pp.72-82.

[52] Wang Z, Khan S M, Jiménez D A. Improving writeback efficiency with decoupled last-write prediction. In *Proc. the 39th ISCA*, Jun. 2012, pp.309-320.

[53] Lee C J, Ebrahimi E, Narasiman V, Mutlu O , Patt Y N. DRAM-aware last-level cache replacement. Technical Report, TR-HPS-2010-007, High Performance Systems Group, Department of Electrical and Computer Engineering, The University of Texas at Austin & Department of Electrical and Computer Engineering, Carnegie Mellon University, Dec. 2010.

[54] HP. Inside the Intel® Itanium® 2 processor. HP Technical White Paper, July 2002. http://www.dig64.org/about/Itanium2_white_paper_public.pdf, Oct. 2014.

[55] Oracle. UltraSPARC T2 supplement to the UltraSPARC architecture 2007. Draft D1.4.3, Sept. 2007. http://www.oracle.com/technetwork/systems/opensparc/t2-14-ust2-uasuppl-draft-hp-ext-1537761.html, Oct. 2014.

[56] Jaleel A, Borch E, Bhandaru M, Steely Jr S C, Emer J. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Proc. the 43rd MICRO*, Dec. 2010, pp.151-162.

[57] Martin M M K, Hill M D, Sorin D J. Why on-chip cache coherence is here to stay. *Commun. ACM*, 2012, 55(7): 78-89.

[58] Albericio J, Ibáñez P, Viñals V, Llabería J M. Exploiting reuse locality on inclusive shared last-level caches. *ACM Trans. Archit. Code Optim.*, 2013, 9(4): Article No. 38.

[59] Binkert N, Beckmann B, Black G, Reinhardt S K, Saidi A, Basu A, Hestness J, Hower D R, Krishna T, Sardashti S, Sen

R, Sewell K, Shoaib M, Vaish N, Hill M D, Wood D A. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011, 39(2): 1-7.

[60] Henning J L. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006, 34(4): 1-17.

[61] Perelman E, Hamerly G, Biesbrouck M V, Sherwood T, Calder B. Using SimPoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 2003, 31(1): 318-319.

Ling-Da Li received his B.E. degree in computer science from Harbin Institute of Technology in 2008. He is now a Ph.D. candidate in computer architecture of Peking University. His research interests include cache system, processor architecture, and multi-core system. He is a student member of CCF and ACM.

Jun-Lin Lu received his Ph.D. degree in computer science from Peking University. He is now an assistant professor in Peking University. His research interests include computer architecture, HW/SW co-design and the communication architecture of system-on-chip.

Xu Cheng is a professor and Ph.D. advisor in Peking University. He is the director of Microprocessor Research and Development Center and a member of Advisory Committee for State Informatization. His research interests include high performance microprocessor, system-on-chip, embedded system, instruction-level parallelism, HW/SW co-design and compiler optimization. He is also a member of CCF.