Enhancing LRU Replacement via Phantom Associativity

Min Feng Chen Tian Rajiv Gupta
Dept. of CSE, University of California, Riverside
Email: {mfeng, tianc, gupta}@cs.ucr.edu

Abstract

In this paper, we propose a novel cache design, Phantom Associative Cache (PAC), that alleviates cache thrashing in L2 caches by keeping the in-cache data blocks for a longer time period. To realize PAC, we introduce the concept of phantom lines. A phantom line works like a real cache line in the LRU stack but does not hold any data or tag. When a phantom line is selected for replacement, cache bypassing is performed instead of replacement. By using appropriate number of phantom lines, PAC can always keep the data blocks that show stronger locality longer in the cache and bypass the cache for other blocks. We show that PAC can be implemented reasonably in practice. The experimental results show that on average PAC reduces cache misses by 17.95% *for twelve CPU2006 benchmarks with Misses Per Kilo-Instruction (MPKI) larger than 1 and by* 6.61% *for all CPU2006 and PARSEC benchmarks. With the help of compiler hints, PAC can further reduce cache misses by* 22% *for benchmarks that have relatively high MPKI or miss rate.*

1 Introduction

The cache performance is a key factor that affects the execution speed of applications. The LRU replacement policy and its variants (e.g., the pseudo-LRU policy) are currently the industry standard for cache replacement policy and have been widely used for many decades. However, the workloads for L2 caches have relatively low locality since L1 caches filter out all the consecutive accesses to the same data blocks. Thus, data swapping, i.e., thrashing, may occur constantly under the LRU replacement policy, which may then degrade the system performance.

Many techniques have been proposed to improve the cache performance by resolving cache thrashing. Many research works have considered cache bypassing and early eviction. McFarling [8] recognized the common instruction reference patterns where storing an instruction in the cache actually harms performance. He then proposed a technique for reducing direct-mapped cache conflict misses by excluding the harmful instruction. Many works [2, 3, 13, 14, 16] present techniques for bypassing or early eviction by using locality information. The underlying idea is to bypass the data accesses which has low reuse. Another series of publications focus on cache optimization by predicting the last touch of a cache line [5, 6]. By knowing the last-touch references, the cache line can be turned off after the last touch to save energy [4]. Lai et al. [5] proposed to use the dead cache line to store the prefetched data. Some techniques [10, 11, 16] try to keep the cache lines that exhibit temporal locality in the L2 cache. However, these techniques incur high storage overhead and require complicated hardware designs for dynamically detecting the temporal locality in lines residing in the L2 cache. Other techniques [1, 9] have been proposed to randomly select the data accesses for bypassing or LRU insertion. Through random sampling, only part of the working set is brought into the cache, which avoids competition for cache resources. In random sampling, the sampling rate must be set appropriately so that the data blocks in the cache can be kept long enough to reach their next usages. However, due to the nature of randomness, data blocks may be evicted either too early or too late causing the cache resource to be wasted.

In this paper, we propose a cache design, Phantom Associative Cache (PAC). The PAC design proposed in this paper aims to improve the cache performance in the presence of cache thrashing. When cache thrashing occurs, for most data accesses the forward reuse distances are larger than the cache associativity. To avoid cache thrashing, the PAC technique inserts extra phantom cache lines into a normal LRU stack. In the LRU stack, a phantom cache line works in the same scheme as a normal one. However, since the phantom cache lines do not have any associated storage, they cannot hold any data or tag. When a phantom cache line is selected for replacement, we bypass the cache and send the requested data block from memory directly to the processors. By adding phantom cache lines, we postpone data evictions and give *the illusion of increased associativity*. Moreover, by carefully selecting the number of added phantom cache lines, we can always keep the data blocks that show better locality in the cache and bypass the cache for the other data.

To get the full benefit from the PAC technique, we need to design a scheme for selecting an appropriate number of phantom cache lines that should be added into the cache for a given program. A simple way is to add an instruction to the ISA that allows us to set the number of phantom cache lines. The compiler can then insert such an instruction into program segments where cache thrashing may occur based on profiling results. In this paper, we present a different technique - a runtime scheme that dynamically adjusts the number of phantom cache lines according to the cache performance. We select a small number of cache sets and apply different numbers of phantom cache lines to them at runtime. The cache misses taking place in these cache sets are tracked for comparing the cache performance with different numbers of cache lines. At regular execution intervals, we select the number of phantom cache lines that produces fewest cache misses for the whole cache. By using the runtime scheme, the PAC technique can adapt to different types of applications as well as different phases of the same application.

In our experiments, we used all 29 SPEC CPU2006 benchmarks and 13 PARSEC benchmarks. The performance was measured on a simulated SPARC processor. We compared the cache misses achieved by PAC and LRU. PAC reduces cache misses on average by 17.95% for twelve CPU2006 benchmarks with Misses Per Kilo-Instruction (MPKI) larger than 1 and 6.61% for all CPU2006 and PARSEC benchmarks. It reduces the cycles per instruction (CPI) on average by 7.1% for the twelve CPU2006 benchmarks with MPKI larger than 1 and 4.3% for PARSEC benchmarks. The experiments also showed that with the help of compiler hints, PAC can further reduce the cache misses by 22% for benchmarks that have relatively high MPKI or miss rate.

The remainder of the paper is organized as follows. In Section 2, we first give the logic underlying PAC and then present its hardware design. In Section 3, we propose a runtime scheme that dynamically adjusts the associativity of PAC. We present the evaluation of our method in Section 4. Section 5 discusses the related work and section 6 concludes this paper.

2 Phantom Associative Cache

2.1 Phantom Associativity

We define an n:m-way PAC as a cache that contains n real cache lines and m phantom cache lines in each cache set. The total associativity of an n:m-way PAC is $(n + m)$. In a PAC, each real cache line can actually store a data block while each phantom cache line is a dummy one that cannot store any data or tag. When a real cache line is selected for replacement, the requested data block is brought into the cache line so that it can be reused in the future. However, when a phantom cache line is selected for replacement, we bypass the cache and send the requested data block directly to the processor since the phantom line cannot hold any data. A normal n-way cache can be used to simulate an arbitrarily large n:m-way PAC. Although the actual storage size of an n:m-way PAC is equal to that of an n-way normal cache, it creates an illusion that the cache has higher associativity by only bringing part of the working set into the cache.

When we apply the LRU cache replacement policy to PAC, data blocks are kept in the cache for a longer period of time and then they have a greater chance of being reused. The LRU policy with phantom associativity works as follows. A LRU stack is maintained to indicate the access order of both real and phantom cache lines. When a cache miss occurs, the LRU policy selects the bottom of the LRU stack (i.e., the least recently used position) for replacement. The replacement decision (replace or bypass) is made based on whether the selected line is real or phantom as mentioned before. No matter whether the selected line is real or phantom, it is then moved to the top of the LRU stack (i.e., the most recently used position). Note that phantom cache lines cannot be hit since their tags are not kept in the cache.

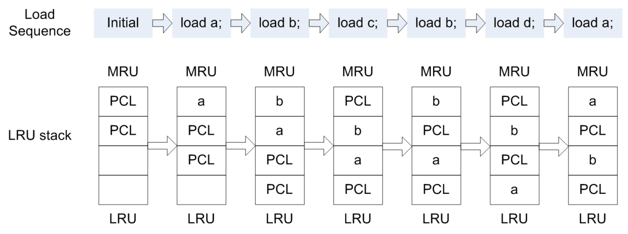


Figure 1. An example of the LRU policy in a 2:2-way PAC, where PCL stands for phantom cache line.

The example in Figure 1 shows how the LRU policy works in a 2:2-way PAC that contains two real cache lines and two phantom cache lines. The data block a, b, c, and d are mapped to the same cache set. Initially, the cache is empty with two phantom cache lines at the top of the LRU stack. The processor first loads a and b. Thus, the two real cache lines at the bottom of the LRU stack are replaced with them and moved to the top of the LRU stack. When c is accessed, the bottom of the LRU stack is a phantom cache line. Therefore, c bypasses the cache and the selected phantom cache line is moved to the top of the LRU stack. The processor then accesses b again. Since b is in the cache, the data access causes a cache hit and the cache line that contains b is moved to the top of the LRU stack. At the fifth data access, since another phantom cache line is available at the bottom of the LRU stack, d is directly sent to the processor without affecting the cache. In the whole procedure, a is always kept in the cache and eventually hit at the last data access because we bypass the cache for c and d. The LRU policy with phantom associativity reduces one miss compared to the LRU policy for a regular 2-way cache.

We use a theoretical model of cyclic references to analyze the LRU policy with phantom associativity on thrashing workloads. Similar model has been used in [12] for modeling conflict misses in caches. Let $(a_1, a_2, \ldots, a_L)$ denote a temporal sequence of memory references mapped into the same cache set, where $a_i (1 \leq i \leq L)$ is the address of the requested data block. Let $(a_1, a_2, \ldots, a_L)^K$ denote a temporal sequence that repeats K times.

We analyze the behavior of the LRU policy with phantom associativity on the access pattern in which $(a_1, a_2, \ldots, a_L)^K$ is followed by $(b_1, b_2, \ldots, b_L)^K$. We assume that the number of real cache lines is smaller than L ($n < L$) and each temporal sequence repeats no less than three times ($K >= 3$). Table 1 compares the hit rate of the LRU policy in an n:m-way PAC and that in a regular n-way cache.

	$(a_1 \ldots a_L)^K$	$(b_1 \ldots b_L)^K$
n-way LRU	0	0
n-way OPT	$\frac{(n-1)(K-1)}{KL}$	$\frac{(n-1)(K-1)}{KL}$
n:m-way LRU (n+$m < L$)	0	0
n:m-way LRU (n+m = L)	$\frac{n(K-1)}{KL}$	$\frac{n(K-1)}{KL}$
n:m-way LRU (n+m = $2L$)	$\frac{n(K-1)}{KL}$	$\frac{n(K-2)}{KL}$
n:m-way LRU (n+m = $3L$)	$\frac{n(K-1)}{KL}$	$\frac{n(K-3)}{KL}$

Table 1. Comparison of theoretical hit rate.

As the cache associativity n is less than L, the LRU policy in a regular n-way cache causes thrashing and cannot achieve any hit for both temporal sequences. The optimal policy brings any n data blocks from the current temporal sequence into the cache in the first iteration and keeps them in the cache until the next temporal sequence. Therefore, it can achieve $(n-1)(K-1)$ hits for both temporal sequences.

In an n:m-way PAC, if the total associativity $(n + m)$ is less than L, the LRU policy still causes thrashing, which is similar to that in a regular n-way cache. However, when the total associativity $(n + m)$ is equal to or larger than the length of the temporal sequence, the LRU policy retains the first n data blocks out of the L data blocks of the first temporal sequence in the cache. The other data blocks in the first temporal sequence are sent to the processor through bypassing since when they are accessed the phantom cache lines are selected for replacement. For the second temporal sequence, if the total associativity of the PAC is exactly equal to L, the LRU policy will bring the first n data blocks of the second temporal sequence into the cache since all the phantom cache lines are consumed in the last iteration of the first temporal sequence. After warm-up, the LRU policy will keep the n data blocks in the cache, similar to the first temporal sequence, and it can achieve $n(K-1)$ hits for the second temporal sequence. When the total associativity of PAC is larger than L, the LRU policy has to use up all the extra phantom cache lines before it can actually bring the data blocks into the cache. Therefore, no cache hits can be obtained for the first few iterations of the second temporal sequence if $(n + m)$ is much larger than L. The smaller the repetition time K, the higher is the degradation in hit ratio due to an overly large phantom associativity. In other words, PAC can achieve optimal cache hits in a cyclic reference model when its total associativity is equal to the length of the temporal sequence. When the total associativity is larger than the length of the temporal sequence, the performance of the PAC gradually worsens with the increase in total associativity.

2.2 Cache Design

In this section, we propose a PAC design that has low space overhead and enables fast phantom associativity adjustment. The extra space overhead for each cache set is $(n \log_2 m)$ bits. As will be shown, to adjust the phantom associativity in this PAC design, we only need to update the value of two registers in each cache set.

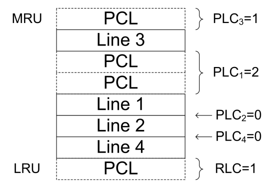


Figure 2. The LRU stack of a 4:4-way PAC.

In our PAC design, the LRU stack of each cache set in a n:m-way PAC only contains n entries that point to the n real cache lines. To maintain the position information of the phantom cache lines, we use $(n - 1)$ counters that count the phantom cache lines between each two consecutive real cache lines in the LRU stack. We also have two counters that store the number of phantom cache lines above the most recently used real cache line and below the least recently used real cache line in the LRU stack. Figure 2 shows an example of a 4:4-way LRU stack, where we have 5 counters (named PLC$_1$~PLC$_4$ and RLC, which will be described later) that count the phantom cache lines in five possible positions. Since we do not distinguish between different phantom cache lines, these $(n + 1)$ counters can tell us all the information we need from the $(n+m)$-entry LRU stack of the straightforward implementation. When a cache miss occurs, we can simply make the replacement decision based on the number of phantom cache lines below the least recently used real cache line in the LRU stack. If there is no phantom cache line below the least recently used real cache line, then the least recently used real cache line is at the LRU position and will be selected for replacement. Otherwise, the LRU position is taken by a phantom cache line and therefore the requested data block will bypass the cache. After each cache access, we update these counters to maintain the state of the LRU stack.

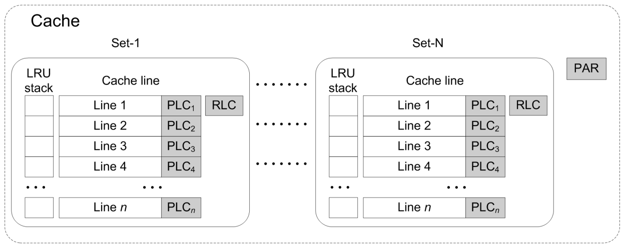


Figure 3. PAC design.

Figure 3 shows our design for PAC. Compared to the traditional cache design, the extra components are shown in the shaded part. We describe these extra components of the PAC in detail below:

Phantom cache line counter (PLC). A PLC is attached to each real cache line. It stores the number of phantom cache lines between this real line and the preceding real line in the LRU stack. Note that for the first real line, its PLC stores the number of phantom lines preceding itself. For a cache that supports up to m-way phantom associativity, a PLC takes $\log_2 m$ bits.

Remaining phantom cache line counter (RLC). The RLC stores the number of phantom cache lines below the least recently used real cache line in the LRU stack. It indicates how many times the cache needs to be bypassed before the next real cache line is selected for replacement. The RLC could be negative after we decrease the phantom associativity at runtime. When a cache miss occurs, the replacement decision is made based on the RLC in the corresponding cache set. The RLC takes $\log_2 m + 1$ bits for a cache that support up to m-way phantom associativity.

Phantom associativity register (PAR). The PAR stores the phantom associativity that the cache currently simulates. The PAR takes $\log_2 m$ bits for a cache that supports up to m-way phantom associativity.

The values of PLCs and RLC need to be maintained at runtime since the cache replacement decision is made based upon them. The phantom associativity of PAC should also be adjustable dynamically since different programs may require different phantom associativities. Figure 4 summarizes the logic for three PAC operations. For each PAC operation, we only need to update at most two counters besides the LRU stack. We describe our modification to the normal LRU cache logic in detail below.

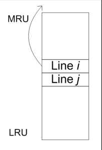
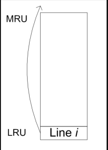
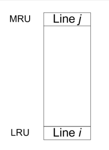
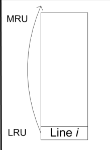
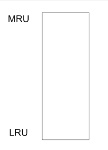
Operation	Cache hit (on line *i*)		Cache miss		PAR change
Situation	Line *i* is in the middle	Line *i* is at the bottom	RLC > 0	RLC <= 0	
LRU stack update					
Counter update	PLC_j=PLC_i+PLC_j PLC_i=0	RLC=RLC+PLC_i PLC_i=0	RLC=RLC-1 PLC_j=PLC_j+1	RLC=RLC+PLC_i PLC_i=0	RLC=RLC+ (PAR_{new}-PAR_{old})

Figure 4. PAC operations.

Modification for a cache hit. Upon every cache hit, the accessed cache line needs to be moved to the top of the LRU stack. Before moving the accessed cache line, we add its PLC value to the PLC of the real cache line just below it in the LRU stack since the two segments of phantom cache lines above or below the accessed cache line will be merged. If the accessed cache line is the least recently used real cache line, its PLC value is added to the RLC since no real cache line is below it in the LRU stack. After being moved to the top of the LRU stack, the PLC of the accessed cache line is reset to 0 because there is no phantom cache line above it in the LRU stack.

Modification for a cache miss. Upon every cache miss, the replacement decision is made based on the RLC in the corresponding cache set. If the RLC is no more than 0, the least recently used real cache line will be selected for replacement since there is no phantom cache line below it in the LRU stack. Otherwise, the requested data will bypass the cache since a phantom cache line is at the LRU position.

If a real cache line is selected for replacement, it will be moved to the top of the LRU stack. Its PLC value is added to the RLC before the PLC is reset to 0. If the requested data block is sent to the processor through cache bypassing, i.e., a phantom cache line is selected for replacement, the RLC will be decreased by 1 and the PLC of the most recently used real cache line will be increased by 1 since a phantom cache line is moved from the bottom to the top of the LRU stack.

Adjusting the phantom associativity. When we adjust the phantom associativity of the cache, we first add the difference between the new and current phantom associativity to the RLC in each cache set. This is equal to adding the new phantom cache lines to the LRU position for increasing phantom associativity or removing the phantom cache lines from the LRU position for decreasing phantom associativity. We then copy the new phantom associativity to the PAR for future phantom associativity adjustment.

2.3 PAC for Pseudo-LRU Caches

Pseudo-LRU is a cache policy that almost always discards one of the least recently used data blocks. It has been used in many modern processors such as Intel Pentium due to its low implementation cost. The space overhead for Pseudo-LRU is only one bit for each cache line. The pseudo-LRU policy works as follows: all cache lines in a cache set are considered as leaf nodes of a binary search tree. Each internal node of the tree has a one-bit flag indicating which way will lead to the pseudo-LRU cache line. To update the tree upon a cache access, the tree is traversed to find the requested cache line and the flags are set so that they point to the direction that is opposite to the direction taken.

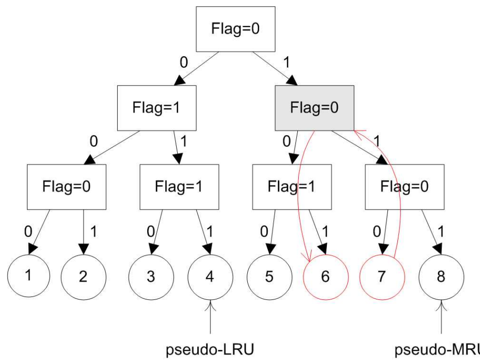


Figure 5. An example of pseudo-LRU tree.

Our PAC technique works with pseudo-LRU caches. As shown in Figure 4, to update the PLCs and RLC for the three PAC operations, we need to know if the accessed line is at the LRU position and which cache line is next to the accessed one in the recency stack. While identifying whether a cache line is at the LRU position is very easy, since the LRU line is indicated by the flags, it is impossible to find a cache line that is next to the accessed one in the recency stack since the pseudo-LRU policy does not maintain a strict recency ordering of the cache lines. However, we can find a cache line that is closer to the LRU position than the accessed one in the pseudo-LRU tree. The procedure is as follows: (1) trace back from the leaf node corresponding to the accessed line to find the first flag that points to the other direction; (2) follow the direction denoted by the flags until reaching a leaf node. The leaf node indicates a cache line that is closer to the LRU position than the accessed one. Figure 5 shows an example of pseudo-LRU tree. In the example, line 4 is at the LRU position as indicated by the flags. To find a cache line that follows line 7 in the pseudo-LRU recency stack, we first trace back from node 7 until reaching the flag in the shaded rectangle and then follow the flags to find node 6. Supposing no hits in the cache set, node 6 has to be selected as victim first to flip the flag in the shaded rectangle so that node 7 can reach the pseudo-LRU position. Therefore node 6 is closer to the LRU position than node 7 in the pseudo-LRU recency stack. Using these information, our PAC technique can work with pseudo-LRU caches without any change.

3 Phantom Associativity Setting

The performance of our PAC technique depends highly on the setting of phantom associativity. Setting large phantom associativity can relieve the cache thrashing for low-locality workloads but increase the cache misses for high-locality workloads, and vice versa. In this section, we present a runtime scheme that dynamically adjusts the phantom associativity of the PAC according to the cache behavior of the running programs.

In PAC, the current phantom associativity is stored in the register PAR. To learn whether the current phantom associativity is optimal, we generate two alternative phantom associativities. One of them is always 0 since we need to make sure our method performs better than a normal LRU cache. For the other associativity, we have a candidate pool of three phantom associativities $\{16, 48, 112\}$, which makes the total associativity $\{32, 64, 128\}$. We select the phantom associativity that is merely bigger than the current PAR value as the second alternative associativity. Set dueling [9] are then used to measure the cache performance with the current and alternative phantom associativities. At fixed execution intervals, we compare the number of cache misses produced by different phantom associativities and select the one that produce fewest cache misses for the cache.

4 Evaluation

We evaluate PAC using Flexus [15], which is a cycle-accurate full-system simulator built on Virtutech Simics [7]. Flexus models the SPARC ISA and allows commercial applications and operating systems to be executed without any modification. The configuration used in our experiments is summarized in Table 2. We run the Solaris 10 operating system on the simulated processor. The SPEC CPU2006 benchmarks and PARSEC benchmarks are used in our experiments. The CPU2006 benchmarks are executed on the single-core model using the reference inputs. The PARSEC benchmarks are executed on the quad-core model using four threads and native inputs. The pre-compiled binaries downloaded from the official website are used.

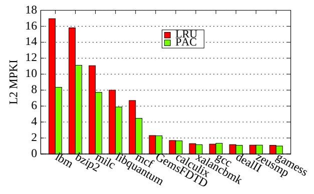
We use the LRU policy as the baseline when we show the results of PAC. Since the PAC we use in the experiments supports up to 112 phantom ways, each cache line (64B) needs a 7-bit PLC and each cache set (64B * 16) needs a 8-

bit RLC as RLC can be negative. The total storage overhead is $7/(64*8) + 8/(64*8*16) = 1.38\%$ of the cache size.

Processors	single-core for CPU2006 benchmarks quad-core for PARSEC benchmarks
Cores	SPARC v9 ISA, 8-stage pipeline, out-of-order execution, 256-entry ROB, 8-wide dispatch, 32-entry store buffer
L1 Caches	Split ID, 64KB 2-way 2-cycle latency, 2 ports 32 MSHRs, 16-entry victim cache
L2 Cache	shared, 4MB 16-way, 64B lines 24-cycle latency, 1 port 32 MSHRs, 16-entry victim cache
Main Memory	4GB total memory 200-cycle access time

Table 2. System parameters.

4.1 Single-Threaded Applications

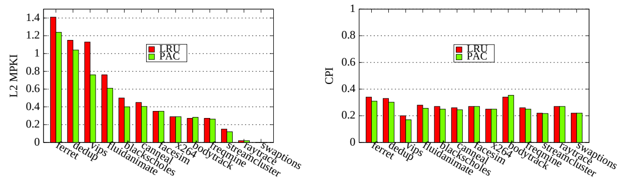


(a) MPKI (b) CPI

Figure 6. Impact on SPEC CPU2006 benchmarks.

Figure 6(a) compares the L2 misses per kilo-instruction (MPKI) achieved by LRU and PAC on SPEC CPU2006 benchmarks. We only show the results for benchmarks with MPKI larger than 1. On average, PAC reduces the L2 cache misses by 17.95% for the benchmarks with MPKI higher than 1. For the benchmarks with MPKI less than 1, PAC only increases the L2 cache misses by 1.2%, which has negligible impact on the system performance ($< 0.5\%$). Figure 6(b) compares the system performance of LRU and PAC in terms of cycles per instructions (CPI). PAC significantly improves the system performance for those benchmarks with higher MPKI since their performance is more sensitive to cache misses. On average, PAC reduces the CPI by 7.1% for benchmarks that have MPKI higher than 1. For the three benchmarks with MPKI higher than 10, PAC improves their performance by more than 10%. On the other hand, the system performance for those benchmarks with lower MPKI is more resilient to changes in number of cache misses. For benchmarks with MPKI less than 1, the average CPI difference between LRU and PAC is within 0.5%.

4.2 Multi-Threaded Applications



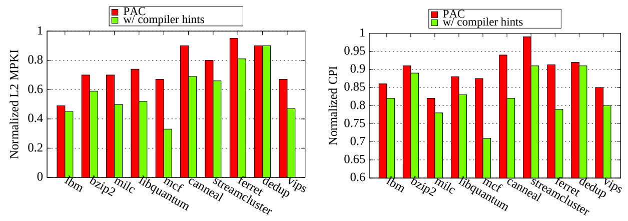
(a) MPKI (b) CPI

Figure 7. Impact on PARSEC benchmarks.

Figure 7(a) shows the L2 MPKI of LRU and PAC on PARSEC benchmarks. Although PARSEC benchmarks generally have lower L2 MPKI than CPU2006 benchmarks, most of them have higher miss rate. For example, for the three benchmarks `blackscholes`, `canneal`, and `streamcluster` that have MPKI less than 1, the miss rates are around 20% which is quite high. Therefore PAC can also reduce their cache misses by alleviating cache thrashing. Compared to LRU, PAC reduces the L2 MPKI for 7 out of 13 benchmarks. On average, PAC reduces the L2 cache misses by 10.1% for PARSEC benchmarks. Figure 7(b) compares the system performance of LRU and PAC on PARSEC benchmarks. On average, PAC reduces the CPI by 4.3% for PARSEC benchmarks compared to LRU.

4.3 Results with Compiler Hints

In this section, we show the performance of PAC if the compiler provides hints for adjusting the phantom associativity. The hints for a program are generated using the following steps: (1) Cache trace is collected by running the program on a training data set; (2) For each main loop in the program, its optimal phantom associativity is found by trying different phantom associativity on the trace; (3) Before each main loop a hint is inserted in the program. During the execution of a program, PAC sets the phantom associativity based on the hints. We conducted this experiment on 10 benchmarks with relatively high MPKI or miss rate.



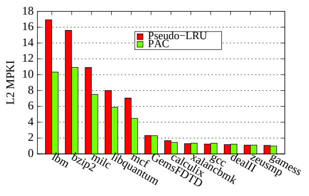
(a) MPKI (b) CPI

Figure 8. Performance with compiler hints.

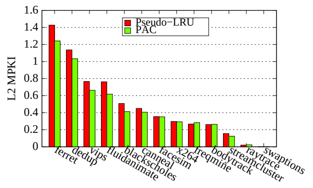
Figure 8 shows the performance of PAC with the help of its compiler. All values are normalized to the LRU results. Since all 10 benchmarks used in this experiment have relatively high MPKI or miss rate, their cache performance has more potential to be improved with greater impact on system performance. Compared to PAC without compiler hints, PAC with compiler hints improves the cache performance for 9 out of 10 benchmarks significantly. On average, PAC with compiler hints reduces the L2 cache misses by 22% for these benchmarks over PAC without compiler hints. Correspondingly, PAC with compiler hints improves the system performance by 7% over PAC without compiler hints.

4.4 Results for Pseudo–LRU Caches

In this experiment, we implemented PAC with the Pseudo–LRU policy as described in Section 2.3. The performance of PAC with Pseudo–LRU is compared with that of regular cache with Pseudo–LRU. We did not conduct this experiments on those CPU2006 benchmarks with MPKI less than 1 since their system performance is hardly impacted by the cache performance.



(a) CPU2006 benchmarks



(b) PARSEC benchmarks

Figure 9. Cache miss reduction with Pseudo–LRU policy.

Figure 9 shows the L2 MPKI with the Pseudo–LRU policy. For CPU2006 benchmark suite, we only show the results for benchmarks with MPKI larger than 1. The performance of Pseudo–LRU is very close to that of LRU. For most CPU2006 and PARSEC benchmarks, the MPKI difference between LRU and Pseudo–LRU is less than 1%. The average MPKI difference between LRU and Pseudo–LRU across all benchmarks is 0.3%. Similarly, PAC achieves similar performance with both LRU and Pseudo–LRU based implementation. The average MPKI difference between PAC+LRU and PAC+Pseudo–LRU across all benchmarks is 1.65%. The most noticable difference occurs on `lbm` benchmark, where PAC reduces cache misses by 51.1% over LRU but only 39.4% over Pseudo–LRU. For most other benchmarks, the improvement achieved by PAC over Pseudo–LRU is slightly larger than that over LRU.

4.5 Comparison with Other Cache Replacement Policies

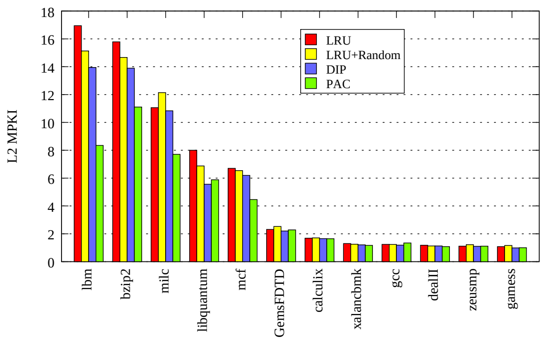
In this section, we compare PAC with other cache replacement policies. Random replacement policy can sometimes alleviate cache thrashing since it does not always evict the LRU line. A possible scheme for improving cache performance is to dynamically select between LRU and Random policy at runtime. Dynamic Insertion Policy (DIP) [9] is another scheme for resolving cache thrashing. It randomly selects LRU or MRU position for inserting new data. We compare PAC with these two policies.

Figure 10 compares the L2 MPKI of LRU, LRU+Random, DIP, and PAC. The CPU2006 benchmarks with MPKI less than 1 are not shown in the figure since their system performances are not significantly impacted by the cache performance. LRU+Random outperforms LRU for most benchmarks with very high MPKI but is mostly inferior to LRU for benchmarks with relatively low MPKI. DIP outperforms both LRU and LRU+Random for most benchmarks. On average, DIP reduces the L2 cache misses by 8.4% for these benchmarks compared to LRU. The overall performance of PAC is best among all four policies. PAC outperforms DIP for 16 out of 25 benchmarks. On average, PAC reduces the cache misses by 14.0% over LRU.

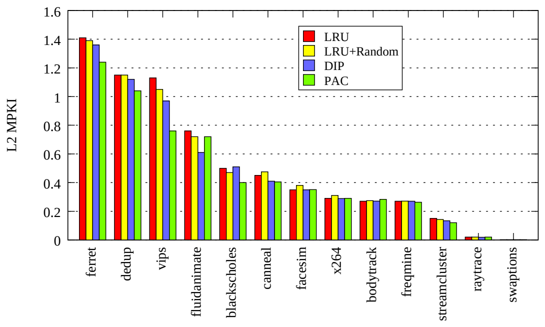
Figure 11 compares the system performance of LRU, LRU+Random, DIP, and PAC. Compared to LRU, DIP improves the system performance by 4.65% for the CPU2006 benchmarks with MPKI larger than 1 and 3.30% for PARSEC benchmarks. For all CPU2006 benchmarks, DIP improves the system performance by 1.95%. PAC outperforms the other replacement policies for most benchmarks shown in the figure. Overall, PAC improves the system performance by 7.06% for the CPU2006 benchmarks with MPKI larger than 1 and 4.34% for PARSEC benchmarks. For the whole CPU2006 benchmark suite, PAC improves the system performance by 3.09% on average.

5 Conclusion

In this paper, we proposed the PAC design for alleviating the cache thrashing in L2 cache. In PAC, the phantom cache lines are introduced to postpone the eviction of the in-cache data blocks. By setting appropriate phantom associativity in PAC, strong-locality data can be kept in the cache for a longer period after being brought into the cache. We also presented a hardware scheme for dynamically adjusting the phantom associativity. According to our experimental results, PAC reduces cache misses on average by 17.95% and improves the system performance by 7.1% for the CPU2006 benchmarks with MPKI larger than 1. On average across all 29 SPEC2006 benchmarks and 13 PAR-

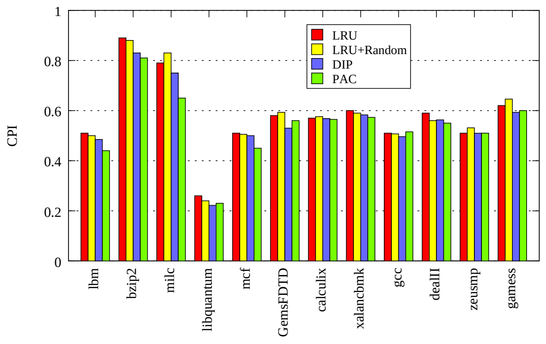


(a) CPU2006 benchmarks

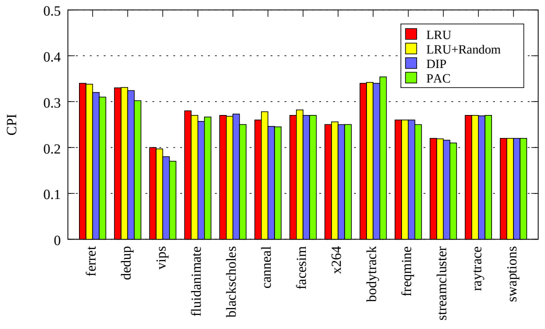


(b) PARSEC benchmarks

Figure 10. Cache miss comparison of cache replacement policies.



(a) CPU2006 benchmarks



(b) PARSEC benchmarks

Figure 11. System performance comparison of cache replacement policies.

SEC benchmarks, PAC reduces L2 cache misses by 6.61%. With the help of compiler hints, PAC can further reduce cache misses by 22% for benchmarks that have relatively high MPKI or miss rate.

References

[1] Y. Etsion and D. G. Feitelson. L1 cache filtering through random selection of memory references. In *PACT*, 2007.

[2] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS*, 1995.

[3] T. L. Johnson. Run-time adaptive cache management. *PhD thesis, University of Illinois, Urbana, IL*, 1998.

[4] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA*, 2001.

[5] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA*, 2001.

[6] W. Lin and S. Reinhardt. Predicting last-touch references under optimal replacement. *Technical Report CSE-TR-447-02, University of Michigan*, 2002.

[7] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[8] S. McFarling. Cache replacement with dynamic exclusion. In *ISCA*, pages 191–200, 1992.

[9] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA*, pages 381–391, 2007.

[10] K. Rajan and G. Ramaswamy. Emulating optimal replacement with a shepherd cache. In *MICRO*, pages 445–454, 2007.

[11] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In *MICRO*, pages 187–198, 2010.

[12] J. E. Smith and J. R. Goodman. Instruction cache replacement policies and organizations. *IEEE Transactions on Computers*, 34(3):234–241, 1985.

[13] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO*, 1995.

[14] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *PACT*, 2002.

[15] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[16] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *HPCA*, 2000.