ACAM: Application Aware Adaptive Cache Management for Shared LLC

Sujit Kr Mahto(✉) and Newton(✉)

Computer Architecture and Dependable Systems Lab.
Department of Electrical Engineering, Indian Institute of Technology Bombay,
Mumbai 400076, India
{sujitkumar,newton}@ee.iitb.ac.in

Abstract. Modern Chip Multiprocessors (CMPs) are typically multicore systems with shared last level cache (LLC). Effective utilization of the shared cache resource can be a challenge when the demands of competing applications conflict with each other. At times, in order to accommodate new data required by one application, the other application's useful data may get evicted. Such negative interference results into increase in memory miss and degrades system's performance. Hence, a technique is required which optimally manages the LLC even in the presence of such conflicting demands.

Various LLC management techniques have been proposed to efficiently manage shared caches. The state-of-the-art replacement policies like Static Re-reference Interval Prediction (SRRIP) and Application Aware Behavior Re-reference Interval Prediction (ABRip) evict a cache block based on their re-usability in the near future. SRRIP makes the replacement decisions per block basis whereas ABRip also considers the cache behavior of an application to minimize conflicting data demands. Hence, ABRip outperforms SRRIP for workload mixes where one application is cache friendly, and the other one is streaming. However, ABRip does not perform well when the workload mix is Cache friendly-Cache friendly. We propose Application Aware Adaptive Cache Management policy that adapts to both types of workload mixes. The proposed replacement policy reduces LLC misses per kilo instruction *(mpki)* up to 22.74% and 12.7% compared to SRRIP and ABRip respectively on a CMP system running SPEC CPU2006 workloads. Our policy effectively utilizes the shared LLC and outperforms both SRRIP and ABRip with performance gains of up to 10.12% and 9.36% respectively.

Keywords: LLC Replacement Policy · Least Recently Used
Set Dueling

1 Introduction

The performance gap between the memory and microprocessor has been widening for last 4 decades as memory speeds have increased slowly compared to that

B. K. Kaushik et al. (Eds.): VDAT 2017, CCIS 711, pp. 324–336, 2017.
https://doi.org/10.1007/978-981-10-7470-7_32

of the CPU speed [1]. Generally, two or three levels of cache hierarchy are used to bridge this gap. In a multicore system, L1 and L2 are generally used as a private cache, and L3 is used as shared LLC for better resource utilization. As the access latency between LLC and main memory is typically high (100–400 cycles), effective management of LLC is critical to system's performance. One of the approaches to effectively manage the LLC is to optimize its replacement policy. As memory request to shared LLC comes from applications with different memory characteristics, the cache access pattern is highly diverse. Memory request from one application interferes with the other application hence increases conflict misses at LLC.

For shared LLC, the optimal replacement policy should allocate more cache resources to the application that shows more data reuse. In a multicore processor, conventionally Least Recently Used (LRU) is used to manage shared caches. However, LRU policy allocates cache resources to an application on the basis of its cache access rate. It does not consider if the application shows temporal locality and hence gives more ways to an application having high cache access rate [5,9]. However, it is not necessary that an application having higher cache access rate also shows more temporal locality and system benefits from such logical cache partitioning.

Several ideas have been proposed in the literature [5–7,9] to efficiently manage LLC. Recently proposed replacement policies like SRRIP [6] and ABRip [7] associates counters to each cache block to track its position in the LRU recency stack and use them to predict whether the cache block will get re-referenced in next few cache accesses or not. On a cache block replacement, both policies evict those cache blocks which are predicted to get re-referenced in far future. ABRip, apart from the block level counters, also associates counter at each core level to differentiate between the cache reuse behavior of applications running on the multicore system. These core level counters help ABRip in classifying the application as cache friendly (Cf) or non-cache friendly (Str). As ABRip also considers application's past cache reuse behavior while deciding about the block's re-reference interval, it performs well compared to SRRIP when the workload mix is Cf-Str. Non-cache friendly application includes the application with data reuse, but working data sets larger than the cache size (*Thrashing application*) and applications which do not show any data-reuse (*Streaming application*). This classification helps in giving (or associating) more ways (or cache size) to a Cf workload when Cf-Str workload mix is running on the cores. However, ABRip fails to adapt when workload mix is Cf-Cf as shown in [7] because, for such workload mixes, ABRip does unfair cache partitioning thus favoring one of the application at the disadvantage of other.

The cache access pattern to LLC is inherently highly diverse in nature. The workload mixes running on the cores can be of Cf-Cf, Cf-Str, or Str-Str type. As mentioned in the previous paragraph, different policies are efficient for different combination of workloads, e.g., SRRIP performs better for Cf-Cf workload mixes whereas ABRip outperforms SRRIP for Cf-Str type of workload combinations. Hence, a cache replacement policy should be adaptive to different types of cache access pattern and various combination of workloads. Techniques like Set Dueling Monitor (SDM) [5] and Auxiliary Tag Directory (ATD) [9] have been earlier used to dynamically choose the best policy among competing replacement policies.

In this paper, we propose Application Aware Adaptive Cache Management (ACAM) for shared LLC. We dynamically switch between two different eviction policies, i.e., taking eviction decision by differentiating application behavior (ABRip) or by considering cache data re-usability only (SRRIP), to make replacement policy adaptive to the different type of cache access pattern and different combination of workloads running on the cores. We use techniques like SDM and ATD to learn the type of cache access pattern so that we can make best possible eviction decision. We evaluate the proposed policy on a set of SPEC CPU2006 workloads running on dual core systems sharing 4MB of LLC.

The rest of the paper is organized as follows. Section 2 explains the background of our work. Our proposal is discussed in Sect. 3. Section 4 provides details about the experimental methodology adopted. Experimental results are discussed in Sect. 5. Related work is discussed in Sect. 6, and in Sect. 7 we conclude the paper.

2 Background

The commonly used LRU replacement policy inserts a block at Most Recently Used (MRU) position, i.e., at the top of recency stack and evicts the block from LRU position, i.e., from the bottom of the recency stack. It performs well for smaller caches such as L1 and L2 which significantly utilize the temporal and spatial locality present in the application. However, for LLC, it performs poorly because most of the blocks inserted to LLC never get re-referenced due to filtering of the temporal locality by smaller cache as observed by several studies [8]. To mitigate this problem, recently proposed replacement policies like LRU Insertion Policy (LIP) [8] and SRRIP [6] insert new block at LRU and closer to LRU position respectively. Only if the block gets hit, it is promoted to MRU position predicting that the cache block is useful and will get re-referenced again in near future.

To study the efficacy of LRU replacement policy on shared caches, we performed experiments to understand the cache reuse behavior under LRU policy for various SPEC CPU2006 benchmark applications. In our experiment, we studied impact on *mpki* of the applications due to variation in the associativity of LLC keeping number of sets and cache block size constant. Figure 1 shows that as associativity increases, *mpki* of *bzip* and *gcc* benchmark reduces indicating their cache reuse behaviour. Whereas *mpki* of *milc* and *libquantum* benchmark remains almost constant which shows their streaming cache access behavior. Based on this observation, we categorize *bzip* and *gcc* as Cf workloads and *libquantum* and *milc* as Str workloads. LRU replacement policy does not perform well for shared caches as it allocates cache resources based on the rate of demand and does not consider whether an application has temporal locality or not. We observed that *mpki* of *bzip* is 5.27 while running mixes (*bzip*-*libquantum*) on dual core system implementing LRU replacement policy in LLC. From Fig. 1, we inferred that *mpki* of *bzip* is 5.27 when it is given 4 out of 8 ways, i.e., allocating half of the cache resources to application *bzip* and the another half to *libquantum*. However, allocating half of the cache resource to *libquantum* does not give performance benefit as it shows streaming cache access behavior with zero reuse. Other replacement policies like Thread-Aware Dynamic Insertion Policy (TADIP) [5], SRRIP manage LLC more efficiently in such cases compared to the LRU policy.

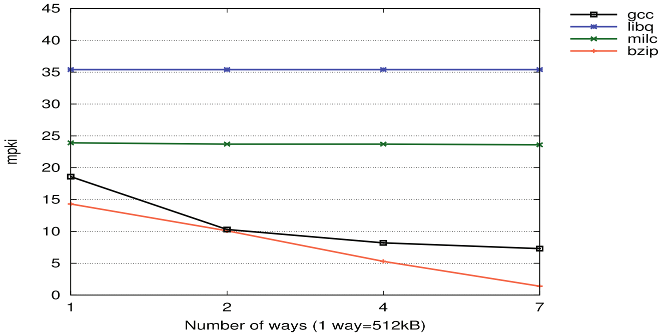

Fig. 1. Memory characteristic of few SPEC CPU2006 benchmarks

SRRIP manages shared caches well as it promotes only re-referenced cache blocks to MRU position, hence filtering out streaming blocks from the blocks having spatial or temporal locality. However, it under-performs compared to ABRip when the workload mix consists of a Str application with high cache access rate and a cache-friendly application with low cache access rate. In such a scenario, before the cache-friendly application's blocks get re-referenced and promoted to MRU, they are replaced by blocks of the high access rate Str application. ABRip uses N more *k-bit* saturating counters per set to differentiate the behavior of applications running on N cores. The value of this counter is used as core level RRPV (Cr). It defines a value of the counter associated with each cache block as block level RRPV (Br). If cache block from any application gets a hit, then its Cr is promoted to 0. This helps in factorizing the behavior of an application. It defines Application Behavior aware RRPV (ABr) as a linear combination of Br and Cr.

$$ABr = Br + \alpha * Cr \tag{1}$$

Higher value of α gives more weightage to Cr than Br. ABRip ($\alpha = 0$) will work same as SRRIP. It evicts cache block which is having ABr value, that is calculated by Eq. (1), greater than or equal to ABr_{max} value. ABr_{max} is defined as Br_{max} plus α times Cr_{max}. As cache block of Cf application will re-referenced more than that of Str application, ABr of Str application's block will reach its maximum value faster compare to Cf application's block which helps in preserving Cf application's block longer in LLC.

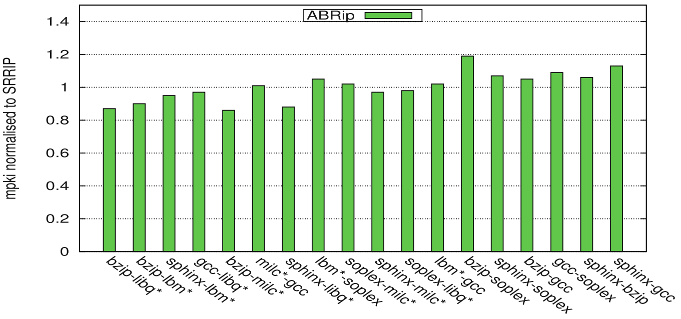

Fig. 2. *mpki* Comparison: Benchmarks with (*) are non-cache friendly application and rest are cache-friendly application

To understand the effectiveness of ABRip and SRRIP on Cf-Str and Cf-Cf workload-mixes, we studied each policies' LLC misses. Figure 2 shows the *mpki* comparison of ABRip ($\alpha = 3$) normalized to SRRIP. The X-axis represents the combination of SPEC CPU2006 benchmarks mentioned in Table 2a. On X-axis, first 12 combinations of benchmarks are the combination of Cf-Str workloads whereas following six combinations of benchmarks are the combination of Cf-Cf workloads. *mpki* of ABRip is higher for Cf-Cf workloads compared to SRRIP because ABRip gives higher priority to an application whose cache blocks get first hit by promoting its Cr value to 0 until Cr value of other application also gets promoted to 0. It protects blocks with $Cr = 0$ from early eviction. However, in Cf-Cf workloads, both applications have temporal locality. So, it hurts the performance of Cf-Cf workloads, whereas, by the same mechanism of protecting blocks with $Cr = 0$, it performs better for Cf-Str workloads. We illustrate such scenario in Fig. 3 for replacement policies ABRip ($\alpha = 2$) and ABRip ($\alpha = 0$) in shared LLC. In policy ABRip ($\alpha = 0$), we give zero weight to Cr and take a decision based on the value of Br only. It works similarly as SRRIP. Here, we consider two applications X and O are running on two cores. The mixed cache access pattern as seen by LLC is (O_1, X_1, X_2, O_1, X_1, X_2, O_3, X_1). Both applications show temporal locality. We assume that the cache blocks (O_1, O_2, O_3, X_2) are initially present in the cache. Request for block O_1 come, and it is a cache hit resulting in the update of the Cr value of application O to 0. Then, request for block X_1 comes and gets a miss. In both policies, block X_2 gets evicted. Then, request for block X_2 comes and gets miss again. Despite that block O_2 has higher Br value and has stayed longer in cache than block X_1, block X_1 get evicted in policy ABRip ($\alpha = 2$) because Cr value of application X is higher than that of application O. Whereas in policy ABRip ($\alpha = 0$), block O_2 get evicted because it does not consider the Cr value of application and make a decision considering Br value only. In policy ABRip ($\alpha = 2$), Cr value of application O is protecting their other blocks from getting evicted and give less number of way to application X. It hurts the performance of application X. Here, we get six hits in ABRip ($\alpha = 0$) whereas in ABRip ($\alpha = 2$) we get only three hits which are shown by blue shaded box in the Fig. 3. Hence, our study identifies and has illustrated the reason for poor performance of ABRip in case of Cf-Cf workload mixes. In our next section, we describe the advanced and adaptive version of ABRip which efficiently manages LLC for all combination of workload-mixes.

ABRip with α=2						Access Pattern	ABRip with α=0					
Core-RRPV		Data Blocks					Core-RRPV		Data Blocks			
O-0	X-3	O_1-0	O_2-2	O_3-1	X_2-2	O_1	O-0	X-3	O_1-0	O_2-2	O_3-1	X_2-2
O-0	X-3	O_1-1	O_2-3	O_3-2	X_1-2	X_1	O-0	X-3	O_1-1	O_2-3	O_3-2	X_1-2
O-0	X-3	O_1-2	O_2-4	O_3-3	X_2-2	X_2	O-0	X-3	O_1-2	X_2-2	O_3-2	X_1-2
O-0	X-3	O_1-0	O_2-4	O_3-3	X_2-2	O_1	O-0	X-3	O_1-0	X_2-2	O_3-2	X_1-2
O-0	X-3	O_1-1	O_2-5	O_3-4	X_1-2	X_1	O-0	X-0	O_1-1	X_2-2	O_3-3	X_1-0
O-0	X-3	O_1-2	O_2-6	O_3-5	X_2-2	X_2	O-0	X-0	O_1-1	X_2-0	O_3-3	X_1-0
O-0	X-3	O_1-2	O_2-6	O_3-0	X_2-2	O_3	O-0	X-0	O_1-1	X_2-0	O_3-0	X_1-0
O-0	X-3	O_1-3	O_2-7	O_3-1	X_1-2	X_1	O-0	X-0	O_1-1	X_2-0	O_3-0	X_1-0

Fig. 3. Access pattern Example: Here, $Br_{max} = 3$, $Cr_{max} = 3$, ABr_{max} for $\alpha = 2$ is 9 and for $\alpha = 0$ is 3. Blue shaded box represents re-referenced block. (Color figure online)

3 Proposed Methodology

3.1 Advanced ABRip

Advanced ABRip (a-ABRip) is an advanced version of ABRip with modifications in its insertion policy. In ABRip, new blocks are inserted at Br equal to $(Br_{max} - 1)$ whereas, in a-ABRip, new blocks are inserted at $(ABr_{max} - 1)$. For most of the time when Cf-Str combinations of workload is running on the cores, the Cr value of Str application is equal to Cr_{max} and that of Cf application is equal to 0. So, the ABr value of newly inserted block of Str application will be equal to $(ABr_{max} - \ + \alpha * Cr_{max})$, i.e., LRU position. Whereas for Cf application, it is inserted with ABr value equal to $(ABr_{max} - \ 1)$ as its $Cr = 0$, i.e., closer to LRU position. This helps in reducing interference from Str application further and reduces *mpki* more for Cf-Str combination of workloads. In SRRIP, it has been shown that inserting blocks closer to LRU position works better compared to other insertion positions. Eviction and promotion policy of a-ABRip is same as that of ABRip.

3.2 Application Aware Adaptive Cache Management (ACAM)

As shown in Fig. 2, for some workload-mixes ABRip has fewer cache misses compared to SRRIP and for other workload-mixes ABRip has more misses. So, different eviction policies perform better for the different combination of workloads. Hence, the replacement policy should be adaptive by dynamically choosing the best performing policy.

The characteristics of a workload-mix, whether it is Cf-Cf or Cf-Str, running on a multicore system can be understood by its mixed cache access pattern as seen by the shared LLC. If the cache access pattern shows that both the applications have temporal locality, then eviction decisions can be taken based on data re-usability, i.e., SRRIP policy. However, if it shows that one of the application has temporal locality while the other application has streaming cache access behavior and causes interference to the first application, then eviction decisions can be taken by differentiating application's behavior, i.e., a-ABRip policy. To make our replacement policy adaptive to different combination of workloads and different type of cache access patterns, we use SDM [5] and ATD [9] to dynamically switch the replacement policy between a-ABRip ($\alpha = 3$) and SRRIP, based on the types of cache access pattern.

SDM. SDM estimates the miss or miss rate of given policy by dedicating few sets to each policy. We use two SDMs on Main Tag Directory (MTD), one for a-ABRip ($\alpha = 3$) and other for SRRIP. After a fixed number of cache accesses (i.e., *phase*), we compare the miss rate of both SDMs and choose the winner policy for remaining sets. Here, we consider 90,000 cache accesses for each *phase*. The winner policy is then implemented on the remaining sets. The winner policy in each *phase* can be different and is chosen dynamically based on the type of its cache access pattern. After every *phase*, we reset the counter values. We use 32 sets for each SDM. In previous work [5,8,10], the authors have compared cache miss of SDMs for comparison. However, in our experiments, we observed that sometimes the number of cache access to one SDM is very high compared to that of other SDM for the same *phase*. So, there is more chance that cache miss of SDM which gets high cache access will be more than that of the other SDM which gets low cache access. For fair comparison, we compare the cache miss rate of SDMs instead of cache miss. To make comparisons more accurately, we have also used ATD where both SDMs see the same cache access pattern as the same sets are dedicated to both competing replacement policies.

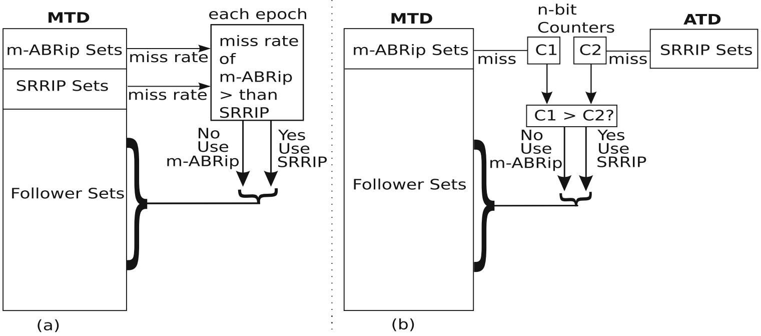

Fig. 4. ACAM Implementations: (a) ACAM-SDM (b) ACAM-ATD

ATD. Here we use one SDM on MTD for a-ABRip policy and we create ATD for SRRIP policy to be used as other SDM. ATD and MTD have the same associativity. We create ATD of those 32 sets that are dedicated to a-ABRip policy in the first SDM on MTD so that both SDMs will see same cache accesses and it will be a fairer comparison. We use saturating counters to count the number of misses of both SDMs. After every *phase*, we compare the counters to decide the winner policy. We implement the winner policy on remaining sets of MTD. On completion of each *phase*, we reset the counter value. Figure 4 shows the implementation of ACAM using SDM and ATD.

4 Experimental Methodology

4.1 Simulation Infrastructure

To evaluate our proposed policy ACAM, we used Sniper [2] multicore x86 simulator. Three level of memory hierarchy was used where L1 and L2 were used as private cache and L3 as shared cache. Table 1 shows the parameter values of baseline processor architecture used. Our cache hierarchy is roughly comparable to the Intel Core i7 [4]. Architecture parameters of the simulated system used here is same as the one used in recent work on shared cache [7].

Table 1. Architecture parameter of simulated system

L1-D Cache	32 KB, 4-Way, LRU, Private, 4-cycles
L1-I Cache	32 KB, 4-Way, LRU, Private, 4-cycles
L2 Cache	256 KB, 8-Way, LRU, Private, 8-cycles
L3 Cache	4 MB, 8-way, Shared, 30-cycles
Main memory latency	175 Cycles
Baseline Processor	x86 Nehalem microarchitecture, 2.67 GHz, 4-wide fetch, 128-entry ROB

4.2 Workloads Combination

On a dual core CMP having shared LCC, we evaluate ACAM replacement policy on 18 different combinations of SPEC CPU2006 benchmarks [3] shown in Table 2a. Out of 18 combinations, 12 are Cf-Str workload combinations, and 6 are Cf-Cf workload combinations. We do not evaluate our policy on Str-Str workload combinations because they do not show any changes in performance irrespective of different replacement policies. We also evaluate our policy on 13 different combinations of SPEC CPU2006 benchmarks shown in Table 2b on 4-core CMPs to understand the scalability of our architecture. We implemented SRRIP with maximum block level RRPV = 15 (i.e., m = 4 bits counter used) and ABRip ($\alpha = 3$) and $ABr_{max} = 60$ (i.e., 6 bit counter used at the block level and 4 bit counter used at core level) and evaluate its performance to compare with that of ACAM replacement policy.

Table 2. Combination of workloads under evaluation

(a) DualCore

Workloads Combination	Type	Workloads Combination	Type
bzip-libq	Cf-Str	sphinx-milc	Cf-Str
bzip-lbm	Cf-Str	soplex-libq	Cf-Str
sphinx-lbm	Cf-Str	lbm-gcc	Str-Cf
gcc-libq	Cf-Str	sphinx-gcc	Cf-Cf
bzip-milc	Cf-Str	bzip-soplex	Cf-Cf
milc-gcc	Str-Cf	sphinx-soplex	Cf-Cf
sphinx-libq	Cf-Str	bzip-gcc	Cf-Cf
lbm-soplex	Str-Cf	gcc-soplex	Cf-Cf
soplex-milc	Cf-Str	sphinx-bzip	Cf-Cf

(b) QuadCore

Mix Name	Workloads Combination	Type
MIX_01	bzip-gcc-milc-libq	Cf-Cf-Str-Str
MIX_02	bzip-gcc-lbm-milc	Cf-Cf-Str-Str
MIX_03	bzip-sphinx-milc-libq	Cf-Cf-Str-Str
MIX_04	bzip-sphinx-libq-lbm	Cf-Cf-Str-Str
MIX_05	bzip-lbm-milc-libq	Cf-Str-Str-Str
MIX_06	gcc-lbm-milc-libq	Cf-Str-Str-Str
MIX_07	soplex-lbm-milc-libq	Cf-Str-Str-Str
MIX_08	sphinx-lbm-milc-libq	Cf-Str-Str-Str
MIX_09	bzip-gcc-soplex-libq	Cf-Cf-Cf-Str
MIX_10	bzip-gcc-soplex-milc	Cf-Cf-Cf-Str
MIX_11	bzip-gcc-sphinx-libq	Cf-Cf-Cf-Str
MIX_12	bzip-gcc-sphinx-lbm	Cf-Cf-Cf-Str
MIX_13	bzip-gcc-soplex-sphinx	Cf-Cf-Cf-Cf

5 Results and Discussion

5.1 Performance Improvement

Figure 5 shows the performance comparison of ABRip, a-ABRip and ACAM using SDM and ATD over baseline SRRIP. a-ABRip improves the performance for most combination of Cf-Str workloads over ABRip by inserting Str application block at LRU position and Cf application block at closer to LRU position. However for Cf-Cf combination of workloads, SRRIP outperforms both policies, i.e., a-ABRip and ABRip. On an average for Cf-Str combination of workloads, a-ABRip outperforms ABRip by 1.22%.

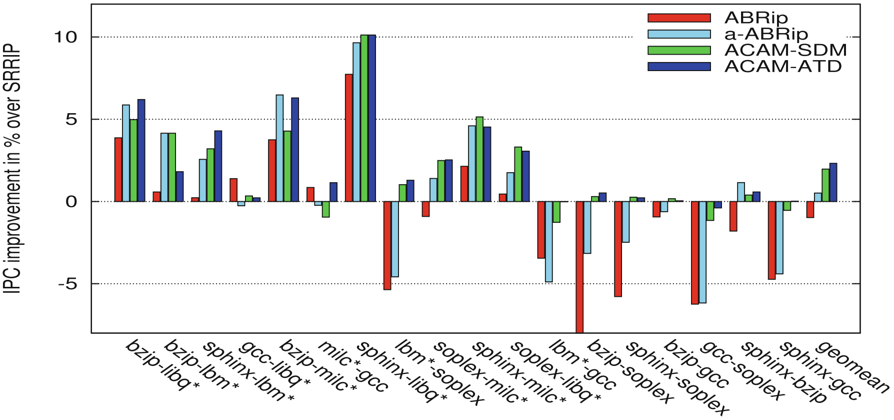

Fig. 5. Performance Comparison: Benchmarks with (*) are non-cache friendly application and rest are cache-friendly application

Using SDM, ACAM outperforms SRRIP and ABRip for most combination of workloads by being adaptive to cache access pattern. It gives maximum performance benefit of 10.12% over SRRIP for workload combination *sphinx-libquantum*. However for some combination of workloads such as *milc-gcc*, *lbm-gcc*, *gcc-soplex* and *sphinx-gcc*, it fails to adapt and lose performance compare to SRRIP. On average, it gives performance improvement of 1.97% and 2.99% over SRRIP and ABRip respectively.

Using ATD, ACAM outperforms SRRIP and ABRip for all combination of workloads except *gcc-soplex* where it loses performance by 0.4% compare to SRRIP. It gives performance gain up to 10.12% for workloads *sphinx-libquantum* compared to SRRIP and 9.36% for workloads *bzip-soplex* over ABRip. On average, it gives performance improvement of 2.32% and 3.33% over SRRIP and ABRip respectively.

Figure 6 shows the performance comparison of ACAM-ATD and ABRip over baseline SRRIP for multi-programmed workloads on 4-core CMPs. The X-axis represents different workload combinations shown in Table 2b. ABRip gives maximum performance benefit of 11.95% for Mix-5. On an average, ABRip gives performance benefit of 6.38% over SRRIP. On top of ABRip, ACAM-ATD gives performance benefit of 4.68% on average. It gains performance by more than 10% compared to SRRIP for 10 out of 13 workload mixes. It gives maximum performance benefit of 19.22% over SRRIP for Mix-2. It outperforms SRRIP policy in all workload mixes and gives performance improvement of 11.35% on average which verifies that our approach is scalable to a large number of cores.

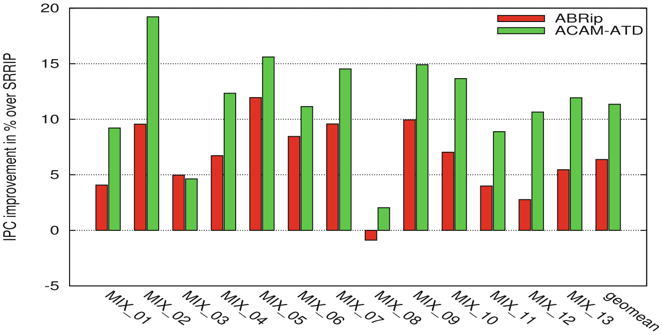

Fig. 6. IPC comparison of multi-programmed workloads: Combination shown in Table 2b

5.2 *mpki* Comparison

Figure 7 shows the *mpki* comparison of ABRip and ACAM-ATD over baseline SRRIP on a dual core CMPs. ACAM using ATD reduces *mpki* up to 22.53% over SRRIP for workloads *bzip-milc*. For Cf-Cf workload mixes, ABRip increases *mpki* by 9.94% over SRRIP. By choosing the best policy based on cache access

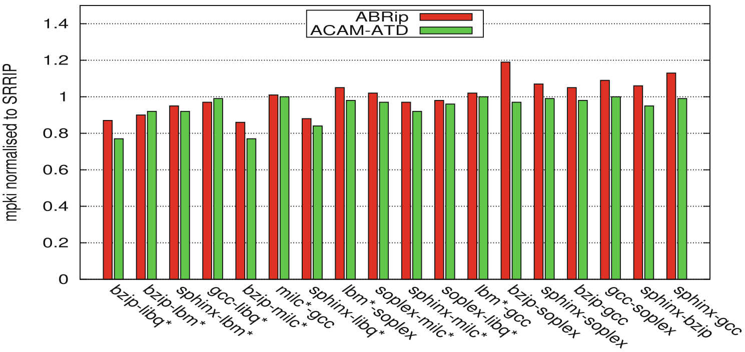

Fig. 7. *mpki* Comparison: Benchmarks with (*) are non-cache friendly application and rest are cache-friendly application

pattern, ACAM-ATD decreases *mpki* by 1.7% for Cf-Cf workload mixes and by 8.28% for Cf-Str workload mixes on an average over SRRIP.

5.3 Hardware Overhead

In order to implement ABRip over SRRIP, additional hardware required is 20 bits per set, i.e., 0.5% on top of SRRIP. a-ABRip does not require extra hardware on top of ABRip. In ACAM using SDM, additional hardware required on top of ABRip is 4 11-bits saturating counters and logic circuit is required to monitor and compare the miss rate to decide the winner policy. Hence hardware overhead to implement ACAM using SDM is very low on top of ABRip. In order to implement ACAM using ATD, hardware required to create ATD contributes to major portion of total hardware overhead over ABRip. Assuming 40 bit physical address space, total hardware require to create ATD is 1.16 KB. In addition to this, 2 11-bits saturating counters and logic circuit is required to compare the cache misses to decide the winner policy. Total hardware overhead to implement ACAM using ATD is 0.026% on top of ABRip.

6 Related Work

The replacement policy of LLC impacts system's performance considerably. Hence, many researchers from academia and industry have significantly contributed with their research work to the improvement in replacement policy managing LLCs. In this section, we summarize prior literature that is relevant to improving LLC's performance.

Jaleel et al. [5] proposed a policy for shared LLC which tried to reduce the interference from non-cache friendly application by inserting its most of cache block at LRU position using Bimodal Insertion Policy (BIP). Using SDM, they implemented BIP policy for non-cache friendly application and LRU policy for cache friendly application. However, LRU policy failed to manage cache blocks efficiently at private LLC even for cache friendly application [6].

Qureshi and Patt [9] logically partitioned the LLC dynamically. They used Utility Monitor (UMON) circuit to track the utility of LLC for each application. Based on each application cache's utility, they partitioned the *ways* in cache-sets among the applications sharing the LLC. They implemented two UMON circuits for dual core CMPs. Auxiliary tag directory (ATD) and counter for UMON circuit were used to study the cache's utility. However, they also used LRU as underlying replacement policy which is not efficient in managing *dead* blocks in LLC.

Wu et al. [10] observed the performance of DRRIP [6] LLC replacement policy in the presence of L2 cache prefetcher. They found that the cache blocks brought into LLC due to the prefetch requests pollute the cache. They tried to reduce such interferences by changing the insertion and promotion policy of prefetched cache blocks. Prefetched cache blocks brought in LLC were inserted at LRU position and were not promoted to MRU even on cache hit.

7 Conclusion

In this paper, we first proposed a-ABRip which helps further in reducing interference to Cf application from Str application when compared to ABRip for Cf-Str combination of workloads. However for Cf-Cf combination of workloads, SRRIP outperforms a-ABRip. To make replacement policy adaptive to different types of cache access pattern and different combination of workloads, we proposed a policy ACAM that switches between a-ABRip ($\alpha = 3$) and SRRIP based on type of cache access pattern. To learn the cache access pattern and decide which eviction policy is better for the next phase of program, we used SDM and ATD. We compared our policy with ABRip and SRRIP for SPEC CPU2006 benchmark. We found our policy ACAM-SDM outperforms ABRip and SRRIP by 2.97% and 1.96% respectively. ACAM-ATD outperforms SRRIP and ABRip by 2.32% and 3.33% respectively on average on 2-core CMPs. On 4-core CMPs, ACAM using ATD outperforms SRRIP and ABRip by 11.35% and 4.68% respectively on average.

References

1. Borkar, S., Chien, A.A.: The future of microprocessors. Commun. ACM **54**(5), 67–77 (2011)
2. Carlson, T.E., Heirman, W., Eeckhout, L.: Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 52. ACM (2011)
3. Henning, J.L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput. Archit. News **34**(4), 1–17 (2006)
4. Intel: Intel Core i7 Processor. http://www.intel.com/products/processor/corei7/specifications.htm

5. Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely Jr., S., Emer, J.: Adaptive insertion policies for managing shared caches. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 208–219. ACM (2008)
6. Jaleel, A., Theobald, K.B., Steely Jr., S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). In: ACM SIGARCH Computer Architecture News, vol. 38, pp. 60–71. ACM (2010)
7. Lathigara, P., Balachandran, S., Singh, V.: Application behavior aware re-reference interval prediction for shared LLC. In: Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD), pp. 172–179. IEEE (2015)
8. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely Jr., S.C., Emer, J.: Adaptive insertion policies for high performance caching. In: ACM SIGARCH Computer Architecture News, vol. 35, pp. 381–391. ACM (2007)
9. Qureshi, M.K., Patt, Y.N.: Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society (2006) 423–432
10. Wu, C.J., Jaleel, A., Martonosi, M., Steely Jr., S.C., Emer, J.: Pacman: Prefetch-aware cache management for high performance caching. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 442–453. ACM (2011)