DAAIP: Deadblock Aware Adaptive Insertion Policy for High Performance Caching

Newton, Sujit Kr Mahto, Suhit Pai and Virendra Singh
{newton, sujitkumar, suhitpai, viren}@ee.iitb.ac.in

Computer Architecture and Dependable Systems Laboratory
Department of Electrical Engineering
Indian Institute of Technology Bombay, India

***Abstract*—The commonly used LRU replacement policy for management of shared last-level cache (LLC) is not efficient as the policy is sharing-oblivious. LRU policy is suitable for applications which show a high-degree of data locality i.e. applications which are cache-friendly. However, applications with working data set greater than the available cache size or poor temporal locality perform poorly with LRU as most of the cache lines inserted by them simply traverse from MRU to LRU position without being re-referenced. Such applications are streaming in their cache behaviour and have very less data reuse. LRU policy makes inefficient use of shared caches for application mixes which are a combination of cache-friendly and streaming applications as the policy treats each cache line independently and doesn't learn from application's past cache reuse behaviour.**

We show that simple adaptive changes to the insertion policy can significantly improve system's performance. We propose Deadblock Aware Adaptive Insertion Policy (DAAIP) which dynamically adapts to the changing cache behaviour of applications sharing the LLC. DAAIP protects the data of application having high temporal locality from high access rate thrashing/streaming applications. Our proposed mechanism monitors each application at-runtime using cost-effective hardware circuits. The information collected is used to dynamically modify the insertion policy and implicitly partition the cache in favour of application showing more data locality. Our evaluation, with 39 multiprogrammed workloads, shows that DAAIP improves performance of dual-core system by up to 21% and on an average 5.8% over LLC caches managed by SRRIP replacement policy. We show that DAAIP also outperforms state-of-the-art cache replacement policy ABRip by 4.6% on system throughput metric.

I. Introduction

For the past 40 years, there has been an exponential growth in the performance of the microprocessors. However, the disparity in the improvement speeds of DRAM and CPU has resulted in a memory bottleneck known as *Memory Wall* [1]. Hence, processors dedicate up to 40% of die area on large on-chip caches to minimize off-chip memory accesses. Large last-level cache (LLC) occupy a significant proportion of these on-chip caches. Modern multicore processors have a LLC which is shared by concurrently executing multiple applications. As many applications simultaneously run on the multicore system, the pressure on the memory system to efficiently meet their diverse data demands increases. Many applications sharing the LLC also compete with each other to make the most of the shared resource. At times, this competition results into conflicts. In order to accommodate new data brought by one application, other application's useful data which is required later, may get evicted. Such negative interference increase costly (100-400 cycles) memory accesses to off-chip physical memory and reduces system's performance. Hence, efficiently managing the LLC becomes very significant in the pursuit of high performance energy-efficient computing goals.

Conventionally, LRU replacement policy or its variants have been used to manage on-chip caches. The LRU policy allocates caches resources on the basis of *access rate* of the application. Hence, it implicitly partitions the cache in the favour of application having high data demand. However, an highly demanding application with high access rate may not use the allocated cache resources efficiently as it may not have high degree of temporal data locality. In cases where a higher access rate application with less data-reuse like *libquantum* competes with a high data-reuse application like *bzip2*, LRU policy allocates more cache to *libquantum* compared to *bzip2*. Such an allocation fails to protect data blocks of *bzip2* application from being thrashed by blocks of *libquantum* application [2].

Generally, LLCs have large associativity to reduce conflict misses among multiple applications sharing it. However, recent studies have shown that in the case of large associative caches, there is a noticeable performance gap between the traditional LRU policy and optimality [3]. Difficulties for the LRU policy in managing shared caches also aggravates as it only observes filtered temporal and spatial locality from lower-level L1 and L2 caches. Due to this filtering, most of the lines installed in the LLC remain unused. We show that with the conventional LRU policy, around 85% of the cache lines inserted in LLC remain unused between their insertion and eviction. Thus, most of the inserted lines in LLC occupy cache space without contributing in reducing cache misses.

We show that simple changes in the insertion policy which are adaptive to the changes in the behaviour of the application can significantly improve the performance for multi-programmed workloads while requiring negligible hardware overhead. We propose the *Deadblock Aware Adaptive Insertion Policy (DAAIP)* which installs the incoming lines either at LRU position or position closer to LRU. The lines are promoted to MRU position only on getting re-referenced. DAAIP prevents thrashing of re-referenced data blocks of cache-friendly (*Cf*) applications from streaming (*Str*) applications having no data-

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.60

IEEE computer society

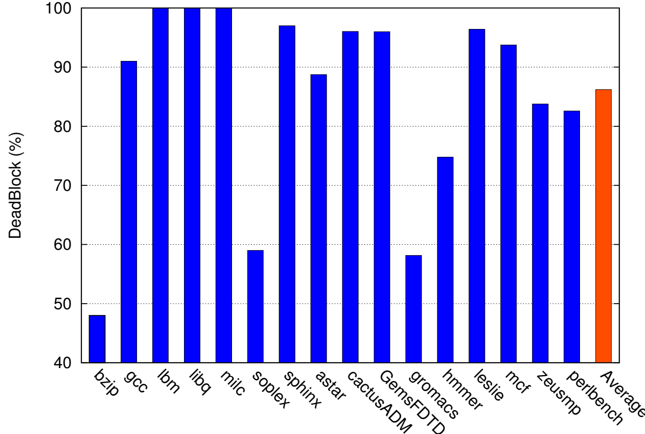

Fig. 1. Zero Reuse Lines for 2MB 16-way L3 cache

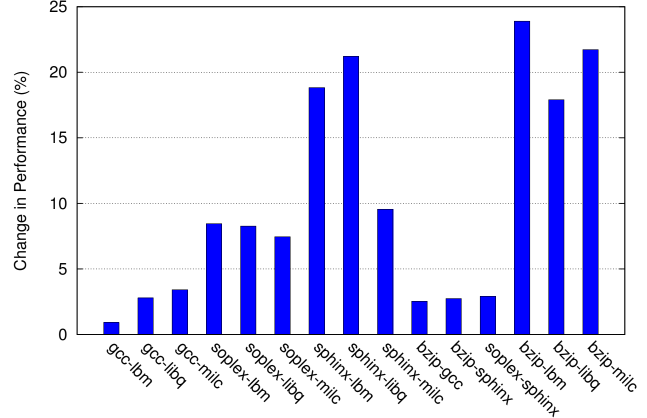

Fig. 2. Performance Improvement of Oracle Vs SRRIP

reuse. This increases the life time of *Cf* blocks in the LLC which in-turn increases their chances of getting re-referenced again and thus reduce off-chip memory accesses. DAAIP achieves performance gains of up to 21% with significant reductions in *misses per kilo instructions* (MPKI) by up to 22% on the evaluated SPEC CPU2006 benchmarks [4] compared to SRRIP [5] policy. DAAIP also outperforms ABRip [2] by 4.6% on an average and improves performance for both *Cf-Str* and *Cf-Cf* type of workload mixes.

The major contribution of our work are following:

1) We show that many incoming lines to the LLC remain unused. Interesting, *all* lines of few application with large working data-sets remain unused.
2) We show that the percentage of incoming unused lines remain almost constant for few subsequent phases for a prediction model to work.
3) We propose a simple yet effective technique which uses the information about unused lines installed by an application to prevent thrashing of *Cf* blocks from *Str* blocks in the shared LLC.

The rest of the paper is organized as follows. Section II explains the motivation behind our work. Our proposal is discussed in Section III. Section IV provides details about the experimental methodology adopted. Experimental results are discussed in Section V. Related work is discussed in Section VI, and in Section VII we conclude the paper.

II. Motivation

A significant proportion of on-chip area in dedicated to LLC as (1) it is shared by multiple cores and (2) an LLC miss will stall the core for hundreds of cycles. Therefore, efficiently managing the LLC is critical to the performance of the processor and hence, it is the focus of our study. As the memory access stream visible to LLC has filtered temporal locality, a significant percentage of LLC cache blocks remain unused. We refer to such blocks as *deadblocks*. Deadblocks fail to get any re-reference while they traverse from MRU position to the LRU position in the LRU-recency stack. Figure 1 shows the percentage of deadblocks inserted into LLC by few SPEC CPU2006 benchmarks. It shows that the percentage of deadblocks inserted by some applications like *bzip2, soplex, gromacs* is less compared to that of applications like *libquantum, lbm, milc*. As per our experiment on an average around 85% of cache blocks installed in LLC are deadblocks. Thus, the traditional LRU policy makes inefficient use of LLC as these unused lines do not contribute to cache hits. Deadblocks are not re-referenced during their stay in the cache as either they do not have any temporal locality or their re-reference distance is greater than the cache size, making them eviction candidates before their possible future re-reference.

Recent studies in the cache replacement policies have suggested that the *LRU chain* can be considered as a *Re-Reference Interval Prediction (RRIP) chain* [2] [5]. RRIP represent the sequence in which the blocks are predicted to be re-referenced, with block at the head of RRIP chain most likely to get re-referenced and block at the tail least likely. As per such policies *Cf* blocks should have *near-immediate re-reference* prediction and *Str* blocks should have *distant re-reference* prediction. However, most of the RRIP-based polices like SRRIP, ABRip are not able to differentiate between *Cf* and *Str* blocks within the same application and hence, unfortunately make *long re-reference* predictions for all the installed blocks of a workload. Moreover, most of these replacement policies make independent predictions for all the incoming cache blocks without considering the past behaviour of the application. ABRip tries to be application aware by factoring in application's behaviour, however, it is also application-aware on per set basis due to independent core-level RRIP counters per set. Hence, behaviour of application as a whole is not considered by such RRIP-based policies.

Dual-core systems running multiprogrammed workloads can have workload combinations of cache-friendly and streaming (*Cf-Str*) workloads in nature. In such cases RRIP-based policies like SRRIP predict same re-reference intervals for blocks from both the workloads. While *long re-reference* interval for blocks from *Cf* application is correct, same prediction for blocks from *Str* application is incorrect and has a negative interference on the blocks of *Cf* application.

The problem worsens when a *Str* application having high data access rate (measured in *accesses per kilo instructions* (APKI)) also has high MPKI. SRRIP, ABRip policies have no mechanisms to take feedback about their prediction accuracy from the evicted cache blocks. DAAIP takes feedback from the evicted cache blocks to decide if the evicted block was re-used or not between its insertion and eviction. Using this re-usage feedback from each evicted block, DAAIP adjusts its predictions to match the behaviour in workload in their different phases. DAAIP also learns about the past history of the workloads from the evicted blocks to make different predictions about re-reference interval for incoming blocks of the different workloads.

Figure 2 shows that by using *oracle's* information about the re-reference interval of cache block even crudely for each phase, and making same predictions for all the blocks inserted in that phase, we can get significant performance benefits of up to 24%. Hence, DAAIP tries to phase-wise learn the cache re-use behaviour of the application from its evicted blocks to make adjust its insertion policy and makes different re-reference interval predictions for blocks from *Cf* and *Str* workloads. Also, since insertion policy comes in effect only during the cache misses, hence, changes to the insertion policy has no effect on the access time of the cache. The reason for the re-reference interval predictions to change in-between different phases of the workload is that the real world workloads suffer frequent *scans* in their cache access pattern [5]. During *scan* phases of the workload, bursts of data references are made whose re-reference is in the *distant* future. Hence, a insertion policy which adaptively changes its insertion decisions based on the phase of the program will be beneficial.

III. Deadblock Aware Adaptive Insertion Policy

A. Framework for DAAIP

Figure 3(a) shows the microarchitecture of Deadblock-aware cache replacement policy. The added structure are shaded. Figure 3(b) shows circuit which decides whether a evicted block is dead or not. The Deadblock Aware Replacement Engine (DARE) is triggered by the cache to take replacement decision of every cache miss. The counters to count inserted blocks *InsertedBlock Counter (IC)* and deadblocks *Deadblock Counter (DC)* are further explained in next section.

B. Tracking Deadblocks in the Cache

To identify if a block was never re-referenced until its eviction from the cache, we add a *reuse* bit to each cache block. The *reuse* bit is zero when a new block is installed in the cache and is set whenever the block is re-referenced and remains set thereafter. We also add $log_2(C)$ bits, C being the number of cores sharing the LLC, to the tag-store entry of each block to identify the core which brought the block to the cache. For a dual core system, only 1 bit is required. To record the total number of blocks and deadblocks brought by each core we add *2K n*-bit counters. For a dual core system (*K=2*), we have used four 16 (*n=16*) bit counters. One 16-bit counter per core, *InsertedBlock Counter (IC)*, records the total number of block installed by that core and other 16-bit counter per core, *Deadblock Counter (DC)*, records the total number of deadblocks installed per core.

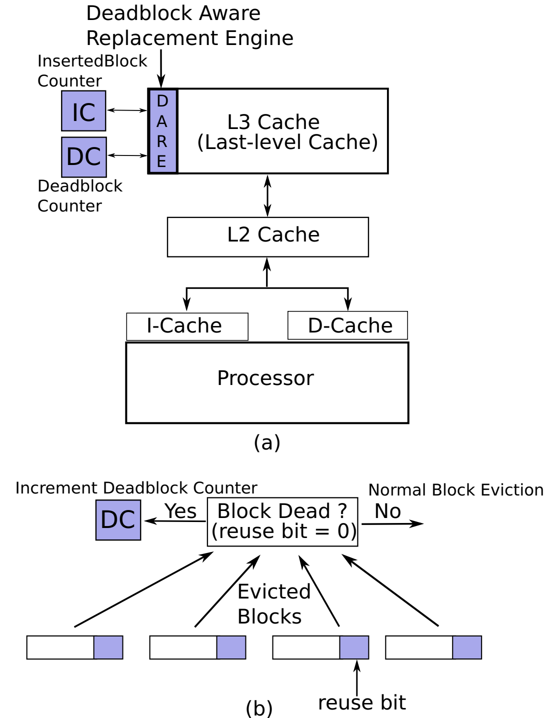

Fig. 3. (a) Microarchitecture of DAAIP. (b) An example of organization of one set of 4-way cache.

Whenever a core requests for a new block and the request is a LLC miss, the block is brought from main memory and is installed into the LLC. The *InsertedBlock Counter* corresponding to that core is also incremented. Similarly, whenever a block is evicted from the cache, its *reuse* bit is checked to identify if the block was re-referenced while the block was in cache. If the *reuse* bit is zero, indicating the block remained unused after first reference, *Deadblock Counter* corresponding to the core who brought it to the cache is incremented.

C. Changes in Insertion Policy

The insertion policy in DAAIP is adaptively changed to reflect the changing behaviour of the application. As shown in Figure 4 and Figure 5, during an application's execution it goes through *phases* (elaborated later) in which the percentage of deadblocks inserted by the application varies. Figure 4 shows the variation in the number of deadblocks inserted by the cache-friendly application *bzip2* benchmarks across its various *phases*. Since, *bzip2* has data locality, percentage of dead-blocks inserted by it are less compared to the percentage of deadblocks inserted by the streaming application *milc* (mostly 100%). The figures also show the behaviour remains almost constant among few subsequent phases and then changes. DAAIP takes insight from this unique behaviour and evaluates

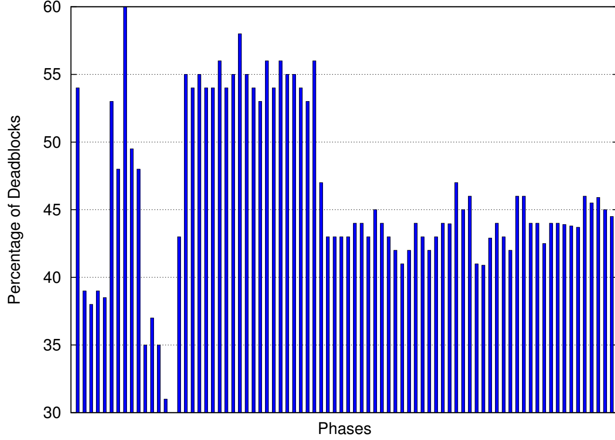

Fig. 4. *bzip2*: Phase-wise changes in the percentage of deadblocks inserted under LRU policy. Each bar corresponds to one phase of the benchmark.

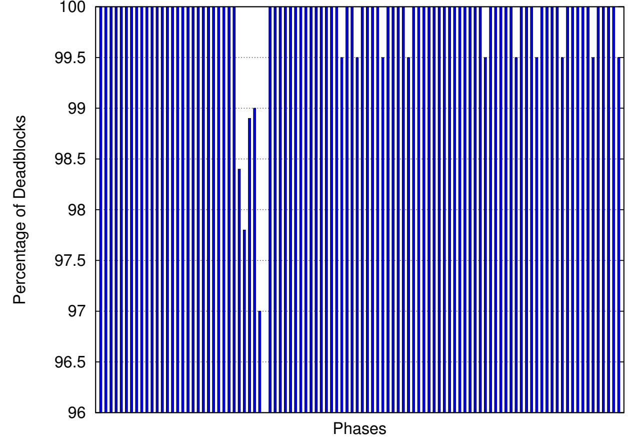

Fig. 5. *milc*: Phase-wise changes in the percentage of deadblocks inserted under LRU policy. Each bar corresponds to one phase of the benchmark.

the percentage of deadblocks for each *phase*. If the percentage exceeds a certain threshold *T*, we label the particular phase as *less-lively-phase*. As per our study, in a *less-lively-phase*, newly requested blocks to LLC should be inserted at LRU position so as to minimize the time spent by deadblocks in the cache. This also helps in filtering out deadblocks from re-used *live* blocks. When the percentage of Deadblocks is within the threshold *T*, we label the phase as *lively-phase*. In *lively-phase*, all new blocks are inserted at positions closer to LRU i.e. (LRU - 1). In *lively-phase* new blocks are still installed at positions closer to LRU as percentage of deadblocks is still very high as shown in figure 1. Hence, as per the changing *liveliness* of blocks in each phase of the application, the insertion policy changes. In our study, each *phase* for an applications is defined as 2^{16} (as our *InsertedBlock Counter* is of 16 bits) cache misses to the LLC. After the end of each phase, the percentage of deadblocks is separately computed for each application and it is compared with the threshold *T* to find out the insertion position of the new blocks of the particular application for next phase. Before beginning of each phase for an application, both of its counters are halved. This allows DAAIP to retain a part of past behaviour while also giving importance to recent information.

We also studied the impact of considering *number of cycles* as a metric to measure each phase. However, as the number of new blocks installed by each application in the cache depends on its *MPKI*, we found that in cases when the workload mix is of a low MPKI and high MPKI application like *bzip2* and *libquantum*, *bzip2* inserts very few new blocks compared to *libquantum* and thus, it was too early to decide on the insertion policy of *bzip2* as it has not yet inserted sufficient new blocks to represent the behaviour of phase. Hence, DAAIP considers *number of misses* to LLC as a metric to measure each phase.

Algorithm 1 Adaptive Insertion Policy for DAAIP

1: ***ForEachCacheMiss***:
2: **if** $NewInsertionCtr$ = $PhaseLen$ **then**
3: *Compute percentageOfDeadblocks*
4: **if** *percentageOfDeadblocks* > *Threshold* **then**
5: *Insert new blocks at LRU*
6: **else**
7: *Insert new blocks at (LRU - 1)*
8: $NewInsertionCtr = NewInsertionCtr/2$
9: $DeadBlockCtr = DeadBlockCtr/2$
10: **else**
11: *NewInsertionCtr++*
12: **if** *BlockEvictedIsDead* **then**
13: *DeadBlockCtr++*

IV. Experimental Methodology

A. Simulation Infrastructure

We evaluate our proposed replacement policy DAAIP using the simulation framework provided by Sniper [6] multi-core x86 simulator. In our study, we have used three-level cache hierarchy which is based on the Intel Core i7 system [7]. The private L1 and L2 caches are managed by LRU replacement policy and the shared LLC is managed by our proposed replacement policy. In our study, we have allocated 2MB cache size for each core. Table I shows the architecture parameters of the baseline configuration used in our experiments, and are similar to those used in the recent work on shared caches [2].

TABLE I
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	2MB per-core, 16-way, Shared, 30 cycles
Main Memory Latency	175 Cycles
Baseline Processor	x86 Nehalem microarchitecture, 2.67 GHz, 4-wide fetch,128-entry ROB

B. Workload Construction

We evaluate DAAIP on 39 different combinations of benchmarks from SPEC CPU2006 benchmark suite [4]. Benchmark

TABLE II
APPLICATIONS AND THEIR CACHE BEHAVIOUR

Application	apki	mpki	Characteristic
bzip	15.50	5.11	Cache Friendly (Cf)
soplex	38.30	19.15	Cache Friendly (Cf)
gcc	28.38	7.93	Cache Friendly (Cf)
sphinx	13.45	11.78	Cache Friendly (Cf)
libquantum	49.45	35.39	Streaming (Str)
lbm	31.92	31.57	Streaming (Str)
milc	26.43	23.68	Streaming (Str)
cactusADM	4.89	5.20	Cache Friendly (Cf)
GemsFDTD	20.52	28.34	Cache Friendly (Cf)
hmmer	1.31	3.07	Cache Friendly (Cf)
leslie	21.67	23.80	Cache Friendly (Cf)
perlbench	0.94	1.34	Cache Friendly (Cf)
zeusmp	3.70	5.42	Cache Friendly (Cf)

with their cache behaviours is shown in Table II. The workload mixes are either *Cf-Cf* or *Cf-Str*. We do not evaluate our policy on *Str-Str* workload mixes as streaming workloads do not show any significant change in their misses irrespective of replacement policies used.

The SPEC CPU2006 references traces were collected using Pin-Points [8] which is based on simpoints [9] for the reference input sets. These workloads were run for 1 billion instructions, that contains 4 traces of 250 million instructions each. During simulation of multi-programmed workload on the dual-core system, our simulation model automatically restarts the trace which completes early. This ensures that the shared cache always receives data requests from both of the running programs throughout the simulation. Recent work on shared caches [2] [5] have also followed similar methodology for multi-programmed workloads.

V. RESULTS AND ANALYSIS

A. Performance on Individual MPKI and IPC Metrics

We compare the results of DAAIP with two state-of-the-art adaptive replacement policies: ABRip [2] and SRRIP [10]. Both the policies predict the re-reference interval of a LLC's block based on its temporal or spatial locality. ABRip also considers application's behaviour, hence it performs better for *Cf-Str* type of workload mixes. Figure 6 shows the relative change in the MPKI for individual application comprising the workload mix. In most of the workload mixes, it can be seen that the MPKI of one application in the mix decreases at the cost of small increase in the MPKI of other application. As shown in the figure, for most of the workload mixes, the gains in the MPKI reduction of one benchmark is very high compared to the small increase in the MPKI of other application. For workload mixes like *bzip-lbm*, *bzip-libq*, *bzip-leslie*, *bzip-gemsFDTD* the cache-friendly application *bzip2* experiences significant MPKI reductions of up to 80% with only a small increase in the MPKI of streaming applications like *libquantum, lbm*. Only in few workload mixes like *gcc-soplex*, *zeusmp-gemsFDTD*, we see increase in the MPKI of both the workloads. In *Cf-Cf* type of workload mixes like *bzip-cactusADM, gromacs-cactusADM, hmmer-cactusADM* both the application benefit due to the efficient management of deadblocks in the LLC by filtering out the deadblocks from

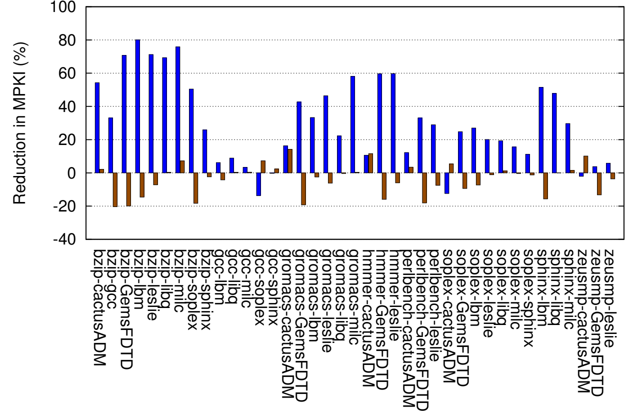

Fig. 6. Relative change in the MPKI of individual benchmarks normalized to SRRIP. Cache-friendly applications show significant reduction in their MPKIs.

cache-reuse blocks. The figure also shows that the DAAIP replacement policy benefits both *Cf-Cf* and *Cf-Str* type of workload mixes unlike ABRip policy which degrades the performance in case of *Cf-Cf* workload mixes.

Due to significant reductions in the MPKI of individual applications of the workload mixes, there is also a corresponding improvement in the IPC of the individual applications. Figure 7 compares the relative changes in the performance of each individual benchmark with SRRIP. Benchmarks which observed significant reductions in MPKI also gained in their performances as is shown by the performance gains of cache-friendly benchmarks like *bzip2, soplex, gcc, gromacs, sphinx, hmmer*. Cache-friendly benchmarks like *bzip2* and *sphinx* show performance benefits of up to 39% and 41% respectively when other benchmark in the workload mix is highly streaming application like *lbm*.

One of the reasons of different levels of performance gains for different benchmarks like *bzip2, gcc, sphinx, soplex* can be understood from the speedup of these benchmarks when LLC size of the core solely running these applications is increased from 2MB to 4MB. As shown by figure 11 when LLC sized is doubled, benchmarks like *sphinx, bzip2, soplex* show maximum speedup indicating better cache usage whereas benchmarks like *milc, libq* are not able to make use of increased cache size due to less data locality. So, in essence DAAIP successfully allocates more cache resources to the application which shows more data locality.

We have not compared DAAIP with DRRIP [10] as DRRIP uses two *Set Dueling Monitors (SDMs)* to dynamically selects the best policy between LRU, BIP and SRRIP. Using two SDMs increase the hardware and related energy costs. Moreover, DAAIP can replace SRRIP if a replacement policy gives less importance to hardware costs.

B. Performance on Throughput Metric

Figure 8 compares the performance benefit of the proposed DAAIP scheme to the SRRIP policy. The bar labeled *GeoMean* represents the geometric mean of the individual performance gains of all the 39 workload mixes. In both type of the

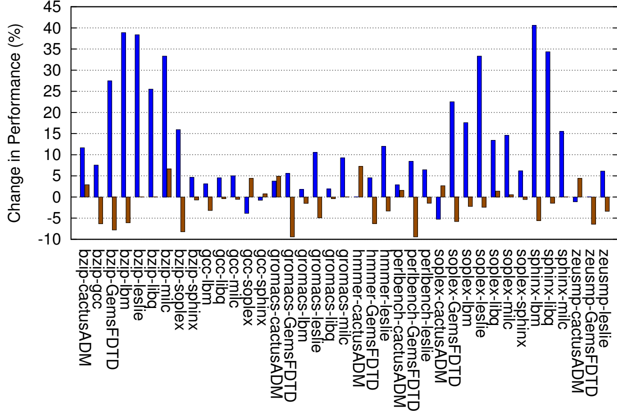

Fig. 7. Relative change in the IPC of individual benchmarks normalized to SRRIP. Cache-friendly applications show significant performance gains.

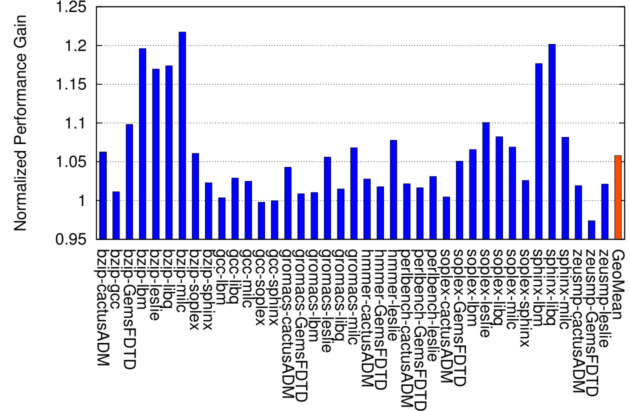

Fig. 8. Relative changes in the IPC gains of workload mixes normalized to SRRIP. *Cf-Str* workload mixes show performance gains.

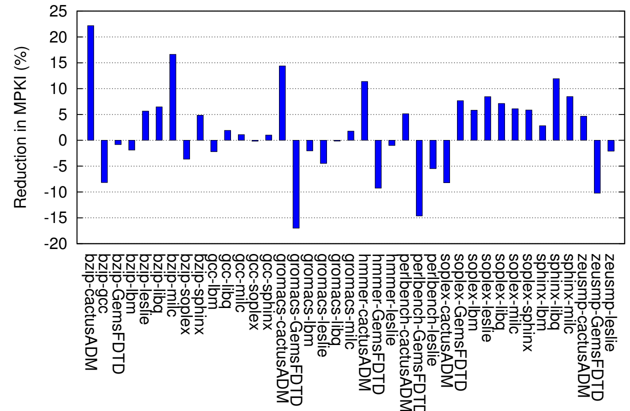

Fig. 9. Relative changes in the MPKI of workload mixes normalized to SRRIP. *Cf-Str* workload mixes show significant reduction in MPKI.

workload mixes i.e. *Cf-Cf, Cf-Str* DAAIP outperforms SRRIP giving an average performance gain of 5.8% across 39 workload mixes studied. *Cf-Str* type of workloads mixes gain more than *Cf-Cf* benchmarks as for *Cf-Cf* type of workloads mixes DAAIP's insertion policy is the same as that of SRRIP for most of the *phases*. Figure 9 shows the changes in MPKI of the workload mixes compared to SRRIP. Workload mixes like *bzip-lbm, bzip-gemsFDTD, groamcs-gemsFDTD, perlbench-gemsFDTD* show net IPC improvements even-though there is net MPKI increase as in these workload mixes the total number of misses to LLC is dominated by the less cache-friendly benchmarks like *lbm, gemsFDTD*. The IPC degradation due to increased MPKI for these less cache-friendly benchmarks in less than the IPC improvement of more cache-friendly application as is shown in figure 6. Figure 6 shows that for workloads mixes *bzip-lbm, bzip-gemsFDTD, groamcs-gemsFDTD, perlbench-gemsFDTD*, the MPKI reduction in more cache-friendly application *bzip2, gromacs, perlbench* is more than the MPKI increase in less cache-friendly application *lbm, gemsFDTD*.

Figure 10 compares the performance of DAAIP to ABRip. ABRip studied *only* 18 workload mixes, hence we consider same 18 workload mixes for our study. Figure 10 shows that for all the workloads mixes except *gcc-milc*, DAAIP outperforms application behaviour ABRip. Since, DAAIP tries to learn behaviour of the application as a whole, it uses the information of block re-use pattern in one cache set to predict the re-reference interval for cache blocks in other set, unlike ABRip where behaviour of application in one set is maintained independent of behaviour in other.

C. Effect of Varying the Threshold and Counters' Size

We also studied the impacts of changing the threshold T and size of the n-bit counters: *InsertedBlock Counter* and *Deadblock Counter* on the performance of DAAIP. We studied the performance impacts of the changes for few workload mixes as shown in Figure 12. The threshold values of 85% and 95% were studied. Similarly, the n-bit counters' size were changed to 10-bits and 14-bits. Figure 12 shows that increasing the threshold value T to 95% and decreasing the counter values to 10-bits or 14-bits have a negative impact on the performance gain on a average. The threshold value of 85% also did not give any significant performance benefit. Hence, the threshold T = 90% and the size of the counters was chosen as 16-bits.

D. Hardware Overhead of DAAIP

Table III detailed the hardware overhead of implementing DAAIP over SRRIP. In DAAIP, 1-bit *reuse* flag per cache block is added to identify if the block was a deadblock. In our dual-core model, 4MB LLC with 64B block has 62500 cache blocks. Hence, 62500 extra bits are required. In addition to this, a bit to the tag-store entry per block has been added to identify the core which brought the block to the cache. Four 16-bit counters to count total new block inserted per core and deadblocks per core have been added. Hence, the total extra hardware required to implement DAAIP is approximately 16KB which is less than 0.005% on top of 4MB LLC. Hardware overhead can be further reduced by considering only 64 or 32 sampled sets [11] to understand the behaviour of the application.

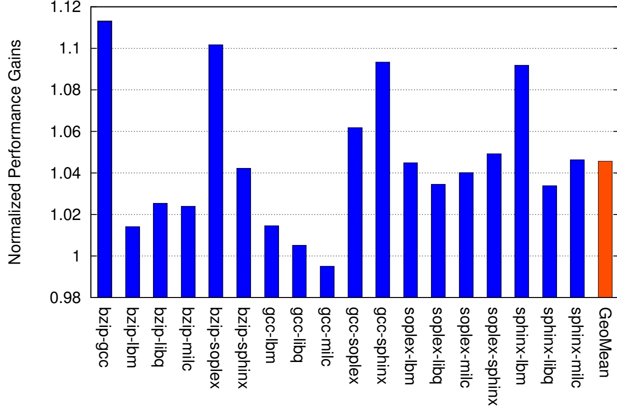

Fig. 10. Performance gains of DAAIP relative to ABRip

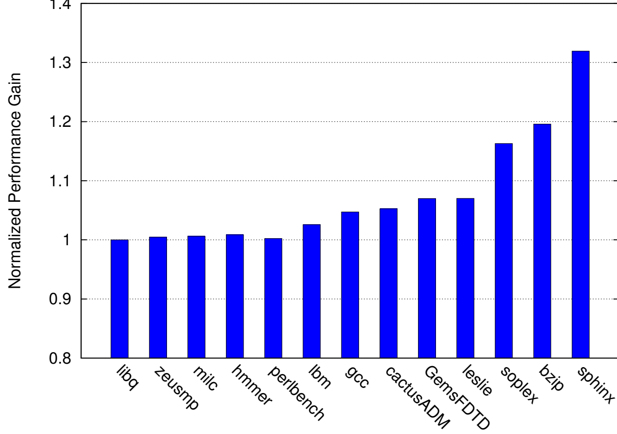

Fig. 11. Relative changes in the performance of the benchmarks when its LLC size is increased from 2MB to 4MB.

TABLE III
HARDWARE OVERHEAD OF DAAIP OVER SRRIP

1 bit per tag-store entry to identify the core	1 b
1 bit per block to track if it is dead at the time of eviction	1 b
Number of cache blocks in 4MB LLC with 64 B blocksize	62500
Overhead per block (62500*2) bits	15625 B
4 Counters each of size 16 bits	8 B
Total Hardware Overhead	**15633 B**
Percentage increase in LLC area due to DAAIP	**0.0039** %

VI. RELATED WORK

Both academic and industrial research communities have made significant contributions [2] [10] [12] [13] [11] [14] [15] in the development of efficient cache management policies. We summarize prior research work in the area of improving LLC performance through efficient management.

An efficient method to implicitly partition the cache among applications on the basis of their utility has been proposed in [11]. Hardware circuit *Utility monitors (UMONs)* and separate ATDs have been used to separately monitor number of misses for all the *ways* in a set-associative cache for each application. *UMONs* informs their partitioning algorithm about utility of extra *ways* for each applications and thus cache is partitioned among competing applications so as to reduce the total number of misses observed by the shared LLC. However, their LLC

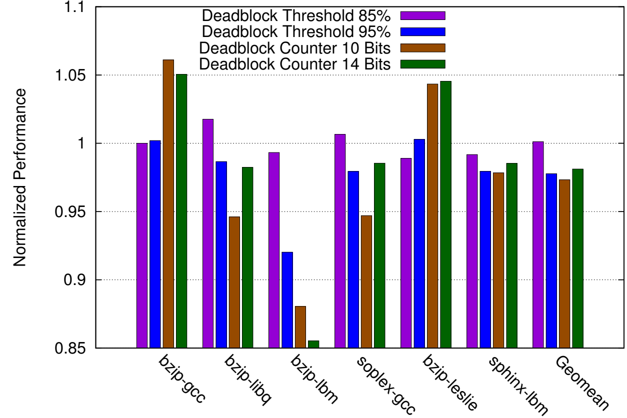

Fig. 12. Performance variation in the workload mixes with change in Threshold value T and size of n-bit Counters.

was managed by LRU policy which is not very efficient in managing deadblocks in a cache. The problem was studied in [12] and a technique to pseudo-partition the cache was proposed by simply managing the insertion and promotion policies of the LRU replacement policy.

Similar investigations to efficiently manage deadblocks in the cache by changing the insertion position of a new cache line in the LRU-recency stack has been done [2] [5] [10]. They have argued that the LRU recency chain can also be interpreted as RRIP chain. The newly brought cache lines are inserted at position closer to LRU [5] or at LRU [10] in the RRIP chain so that cache pollution due to deadblocks can be eliminated. However, their replacement policies did not consider the behaviour of the applications sharing the LLC in eviction decisions for a block. In [2] authors have proposed an *Application Aware RRIP (ABRip)* where RRIP counters were also used at application level to efficiently manage LLC when the application mix sharing the LLC is of *Cf-Str* type. However, for Cf-Cf workload mixes ABRip degraded the performance.

In [16] authors have studied the benefits of removing unused words from a shared L2 cache line for memory-intensive applications having low spatial locality. They have used a novel cache organization, *Distill Cache*, which has both *Line-Organized Cache (LOC)* and *Word-Organized Cache (WOC)* to filter out unused words. However, their architecture has an storage overhead for storing extra tags for lines in WOC. Also, they have not studied performance of *Distill Cache* in a multicore system where the interactions among applications sharing the LLC is more competitive.

In [17] authors have logically partitioned the cache into *Shepherd Cache (SC)* (emulates OPT) and *Main Cache (MC)*. Their study is based on the observation that a smaller 256KB cache managed by OPT performs better than a 512KB LRU managed shared cache. In their cache organization SC guides the replacement in MC. Each new cache line is buffered into SC and is allowed to gather information of the optimal cache line it should replace in MC. Their logical partitioning devoted 25% of cache to SC and remaining to MC.

While *ABRip* [2] provides good performance gain in case of cache-friendly streaming workloads mixes, *ABRip* limits performance in case of cache-friendly cache-friendly workload mixes as *ABRip* can favour one of the cache-friendly workloads during cache allocation. Such biasing results in disappropriate cache allocation to one workload and increases cache-misses for another workload.

A unique technique to consider the cost of the misses in order to reduce the number of isolated misses was presented in [14]. The focus of the study was to classify LLC misses into isolated and parallel misses, as parallel misses have less cost per miss due to overlapping memory accesses supported by non-blocking caches. The authors have proposed a model which combines both recency of the block and its MLP (Memory-level parallelism) cost while considering the replacement candidate. However, their work was only focused on single workloads and multiprogrammed workloads were not studied. Multiprogrammed workloads present a unique case as there is also negative interference among applications sharing the LLC.

As the memory access stream visible to LLC has filtered temporal locality, the authors in [18] have studies the performance gains on making LLC partially aware of the temporal locality visible to the L2 cache. They have proposed a Temporal Locality Aware (TLA) cache where L2 passes re-usage hints about its lines to the replacement policies managing LLC, so that the policy can update recency state of such lines in LLC.

VII. Conclusion and Future Work

Last-level caches are shared by multiple workloads having diversified nature and different access pattern. Cache replacement policies managing shared caches should be aware about such different behaviour while taking replacement decisions. Traditional design for shared caches use LRU replacement policy which is sharing-oblivious and logically partitions the cache among competing applications on their demand basis.

In this paper, we have introduced Deadblock Aware Adaptive Insertion Policy (DAAIP) for shared caches which takes feedback from evicted cache blocks to identify the cache-reuse behaviour of the application. We have shown that a significant percentage of the cache lines inserted by applications in the LRU managed LLC remain unused. Also, the percentage of deadblocks inserted per phase remains fairly constant in between subsequent phases. Our proposed policy makes use of both these insights in its adaptive insertion decisions. It is able to differentiates between cache blocks of cache-friendly and streaming applications, and makes different re-reference interval prediction decisions for them. Our study has shown the usefulness of being aware of the re-use information of installed cache blocks. DAAIP being deadblock aware efficiently makes different insertion decisions for cache-friendly and streaming workloads in a multiprogrammed workload mix. By such simple yet effective changes in the insertion policy, DAAIP is able to out perform SRRIP by up to 22% and 5.8% on an average, while requiring less than 0.005% hardware overhead. We also show that DAAIP outperforms *state-of-the-art* cache replacement policy ABRip by 4.6% on system throughput metric.

In future, we plan to investigate the benefits of using ATDs to dynamically switch between best performing policy, as for few workload mixes, DAAIP looses on performance compared to SRRIP. We also plan to study the scalability of DAAIP for quad-core systems.

References

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[2] P. Lathigara, S. Balachandran, and V. Singh, "Application behavior aware re-reference interval prediction for shared LLC," in *33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 2015.

[3] W. A. Wong and J.-L. Baer, "Modified LRU policies for improving second-level cache behavior," in *Proceedings of 6th International Symposium High-Performance Computer Architecture*. IEEE, 2000.

[4] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[5] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 52.

[7] Intel Core i7 Processor, http://www.intel.com/products/processor/ corei7/specifications.htm.

[8] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2004, pp. 81–92.

[9] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *Micro, IEEE*, vol. 23, no. 6, pp. 84–93, 2003.

[10] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 381–391.

[11] M. K. Qureshi and Y. N. Patt, "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, pp. 423–432.

[12] Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 174–183.

[13] J. T. Robinson and M. V. Devarakonda, *Data cache management using frequency-based replacement*. ACM, 1990, vol. 18, no. 1.

[14] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 167–178, 2006.

[15] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[16] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *Proceedings of 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 250–259.

[17] K. Rajan and G. Ramaswamy, "Emulating optimal replacement with a shepherd cache," in *Proceedings of 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2007, pp. 445–454.

[18] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr, and J. Emer, "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010, pp. 151–162.