AB-Aware: Application Behavior Aware Management of Shared Last Level Caches

Suhit Pai, Newton Singh and Virendra Singh
{suhitpai, newton, viren}@ee.iitb.ac.in
Computer Architecture and Dependable Systems Laboratory
Indian Institute of Technology Bombay, India

ABSTRACT

In modern multicore systems, Last-Level Cache (LLC) is usually shared among multiple cores. Though it benefits applications by sharing and utilizing cache resources efficiently; the benefits come at the cost of increased conflict misses due to interference among applications. In shared LLC, conventionally used LRU-based cache replacement policies logically partition the cache on-demand basis. Thus, cache friendly applications sharing LLC with streaming applications, suffer due to high data demands and low reuse of streaming applications. Apart from different data locality behavior, applications also show different memory access behavior while accessing the LLC. Some applications inherently have parallel memory accesses while others have more isolated long-latency accesses. The cost of idle cycles processor spends waiting for off-chip memory accesses is shared by parallel misses. However, misses which occur in isolation hurt the performance most. This adds another dimension to application's behavior. We propose an application behavior aware cache replacement policy to manage shared LLC. The proposed policy simultaneously reduces the negative interference among applications sharing the LLC and the miss-penalty associated with each LLC miss. Evaluation on SPEC CPU2006 benchmarks shows that our replacement policy improves performance on dual-core systems and quad-core system by up to 15.9% and 23.8% respectively over SRRIP for shared LLC. It is worth to note that effectiveness of our policy improves with the increase in the number of cores.

ACM Reference Format:
Suhit Pai, Newton Singh and Virendra Singh. 2018. AB-Aware: Application Behavior Aware Management of Shared Last Level Caches. In *GLSVLSI '18: 2018 Great Lakes Symposium on VLSI, May 23–25, 2018, Chicago, IL, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3194554.3194573

GLSVLSI '18, May 23–25, 2018, Chicago, IL, USA

ACM ISBN 978-1-4503-5724-1/18/05…$15.00
https://doi.org/10.1145/3194554.3194573

1 INTRODUCTION

In the last couple of decades, microprocessors' performance has improved almost thousand-folds mainly due to core micro-architectural innovations and transistor-speed scaling. However, the DRAM access latency has not improved as significantly as the CPU's speed. This disparity in the speeds of DRAM and CPU has resulted in a memory bottleneck known as *Memory Wall*. Hence, modern processors dedicate a large portion (∼40%) of die area to on-chip caches for reducing off-chip memory accesses. Processors use a large on-chip L3 cache as the LLC. It acts like a victim cache for L1 and L2 caches and also helps in maintaining cache-coherency in a multicore system. Each LLC miss stalls the CPU for ∼200 cycles and significantly degrades processor's performance [2].

Sharing LLC enables dynamic allocation of cache space between multiple cores. However, multiple cores utilizing the LLC also compete with each other to make the most of the shared resource. At times, these conflicts result in negative interference among the competing cores. One core may evict the useful data of the other core, resulting in unnecessary off-chip memory accesses to the main memory, thus reducing the system performance. *Hence, effective management of LLC is very critical to the performance of a multicore system.*

Conventionally, LRU-based policies are used to manage LLC. Such policies perform sub-optimally for LLC as most of the temporal and spatial regularity present in the data is filtered-out by the higher level caches (L1 and L2). Caches benefit by utilizing the locality present in the data. Hence, a filtered access pattern makes LRU-based policies inefficient in managing the LLC. The problem further aggravates as such policies allocate cache resources on the basis of access rates of the competing applications. The cache is implicitly partitioned in favor of application having high access rate. However, a highly demanding application may not have high data locality, resulting in a poor resource utilization. Thus, considering application's data-reuse behavior can be beneficial while allocating LLC resources among applications with different access rates. Data-reuse pattern of cache blocks brought by an application can be assimilated to better understand application's behavior and limit the negative interference of high access rate applications.

Apart from different data-reuse behavior, applications also show different memory access pattern. Some applications have multiple memory accesses which can be parallelly serviced, while others have more isolated accesses. The ability to process multiple memory operations concurrently is called Memory Level Parallelism (MLP) [10]. The penalty per miss

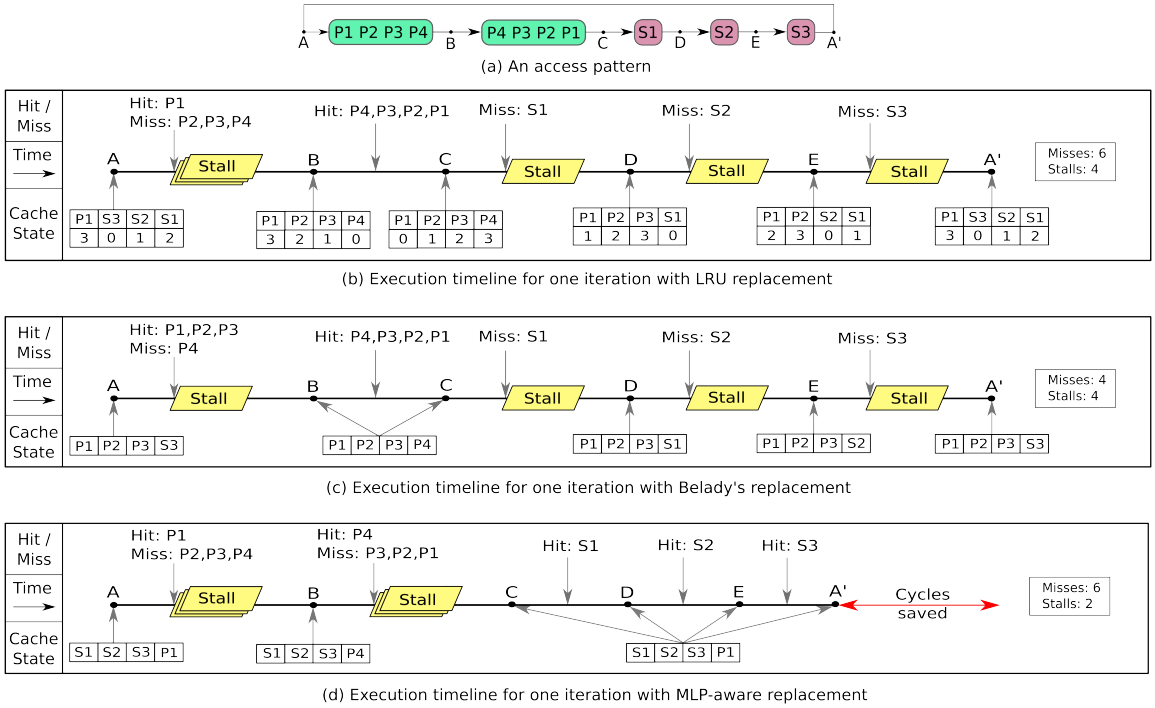


Figure 1: The drawback of MLP-unaware Cache Replacement

incurred while servicing parallel misses is less than that of isolated misses. Most of the LRU-based cache management policies focus only on reducing the number of LLC misses, assuming same cost for each miss. Hence, such policies are oblivious to the observation that servicing N parallel misses may be less costlier than servicing M ($N > M$) isolated misses. MLP information can be useful while managing caches, however, it can complicate the replacement policy. Such a replacement policy is costly for L1 & L2 caches, considering their relatively lower miss penalties as compared to that of LLC. Hence, using both the data-reuse and MLP-behavior of an application can be effective in managing the LLC.

This paper proposes an application behavior aware cache replacement policy for shared LLC. Our policy manages the LLC by considered data-reuse behavior as well as MLP behavior of the applications sharing the LLC. It simultaneously reduces both the number of misses and the cost of misses. Evaluation of SPEC CPU2006 benchmarks on a multicore system sharing LLC shows that our policy outperforms both *state-of-the-art* SRRIP [6] and ABRIP [7] policies.

2 MOTIVATION

Traditional cache replacement algorithms try to reduce the overall number of cache misses assuming same miss penalty for each miss. However, this assumption may not be always true. Consider an example [10] of a loop having 11 memory accesses to LLC as shown in Figure 1(a). A, B, C, D and E are points representing a program's intervals. Blocks *P1*, *P2*, *P3* and *P4* are accessed between interval A & B and B & C. The four accesses of these blocks are close to each other in instruction space and hence, can be serviced in parallel. While *S1*, *S2* and *S3* are the sequential accesses between interval C & A' which result in *isolated misses*. As an example, we consider a fully-associative cache having four cache blocks.

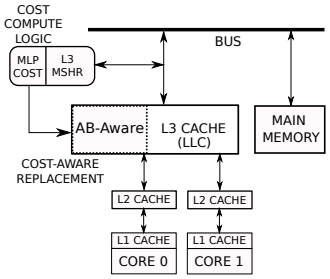
Figure 1(b) shows the behavior of LRU replacement policy for this loop. While managing the cache, LRU replacement policy only considers the recency of a block and is oblivious to the block's MLP behavior. Recency value of each cache block has been shown as cache state (MRU = 0 and LRU = 3). LRU replacement policy results in total 6 misses and 4 stalls. Cache behavior under Belady's OPT is shown in Figure 1(c). Belady's OPT [1] minimizes the number of misses by evicting those blocks which will be referred farthest in future, based on oracle's information. Blocks *P1*, *P2*, *P3* and *P4* are referred more than once as compared to blocks *S1*, *S2* and *S3*. So, in order to minimize the misses, it will try to retain *P* kind of blocks. Belady's OPT replacement is able to reduce parallel misses, making a total of 4 misses and 4 long-latency stalls for every iteration of the loop.

It is interesting to observe, as shown in Figure 1(d), that we can still reduce the number of long-latency stalls from 4 to 2 if our replacement policy is MLP-aware. MLP-aware policy focuses on minimizing isolated misses by not evicting those cache blocks which result in isolated misses. *Thus, the policy prefers retaining S type cache blocks over P type cache blocks.* However, making the replacement policy only MLP-aware will not be sufficient while managing caches, as caches mainly work on the philosophy of utilizing the recency information of data blocks. Several policies have been proposed earlier [5] [6] which consider the recency information of cache blocks to predict their future re-references. Cache blocks which are less likely to be re-referred again or which are going to be re-referred in far future are placed near LRU position and the ones which are likely to be re-referred in the near future are

placed near MRU position. However, LRU replacement policy predicts that each newly inserted block will be re-referenced in the near future and thus, places them at MRU position.

Predicting the future re-references of cache blocks becomes more difficult for a multicore system when its LLC is shared among multiple applications running on different cores. In such a case, the LLC experiences mixed access pattern from applications with diverse behavior. Considering only the recency information of cache blocks and being ignorant of the behavior of the application which brought the cache block can be sub-optimal [7]. LRU-based replacement policies and recently proposed policies [6] [5] manage cache by utilizing the recency behavior of cache block. They do not assimilate the recency information of each block to understand the application's behavior better. Applications' data access behavior was utilized in [7] to classify them as cache-friendly (*CF*) or streaming (*STR*) applications. However, insights about applications' data access behavior and MLP-behavior can be used to classify their temporal-varying phases more specifically as *CF-IM* (cache-friendly isolated misses), *CF-PM* (cache-friendly parallel misses), *STR-IM* (streaming isolated misses), and *STR-PM* (streaming parallel misses). In an application's *IM* phase, it suffers from many isolated misses whereas in *PM* phases misses can be parallelized to reduce effective miss penalty. The priority of retaining cache blocks in the LLC of applications should be *CF-IM* > *CF-PM* > *STR-IM* > *STR-PM*. This more specific classification gives more insight into an application's behavior and can be effectively utilized to manage the LLC.

3 APPLICATION BEHAVIOR AWARE LLC MANAGEMENT



(a)

MLP-based cost value	Quantized value
0-22 cycles	7
23-45 cycles	6
46-68 cycles	5
69-91 cycles	4
92-114 cycles	3
115-137 cycles	2
138-160 cycles	1
160+ cycles	0

(b)

Figure 2: (a) Microarchitecture of AB-Aware policy (b) Quantization of *mlp_cost* to *q_cost*

3.1 Framework for AB-Aware policy

Figure 2(a) shows the microarchitecture of AB-Aware Policy. The LLC consists of a cost-aware replacement engine. It calculates the MLP-based cost for LLC misses according to algorithm 1, as described in section 3.2. The MLP-based cost value is quantized to 3-bits (to limit the storage overhead) and is stored with each cache block. The recency information about a particular block is collected using recency counters at each block. Our proposed policy takes replacement decisions by considering a weighted sum of block's mlp_cost and its recency information.

3.2 Computation of MLP-based cost

The MLP-based cost of each block is computed using the number of LLC accesses to main memory as done in [10]. Whenever a LLC miss occurs, an entry is made in Miss Status Holding Register (MSHR) before sending the request to main memory. The information about the *number of in-flight misses* and *wait-cycles before being serviced* is taken from MSHR. To compute MLP-based cost, a field *mlp_cost* has been added to each entry of MSHR and the entry is set to 0 when a new entry is made in MSHR. The quantized value of the field *mlp_cost* is stored in tag-store entry of each block and is used by replacement algorithm. The *mlp_cost* of a block shows predictability across successive misses to the same block as shown in [10]. Hence, the MLP-behavior of a block captured as its *mlp_cost* shows phasic behavior and can be predicted.

Algorithm 1 Calculation of MLP-based cost for LLC misses

```
1: init_mlp_cost(miss):        /* when miss enters MSHR */
2:     miss.mlp_cost = 0
3: update_mlp_cost():          /* called every cycle */
4: N_i ⇐ Number of outstanding misses from i^th core
5: for each demand miss in MSHR do
6:     mlp_cost += (1/N_i)
7: end for
```

We modified the algorithm proposed by Qureshi *et al.* [10] for MLP-based cost calculation to make it suitable for shared LLCs in a multicore system. *mlp_cost* for all the outstanding misses is incremented by (1/Number of outstanding misses from i^{th} core) per cycle, where i is the core-id of the requesting core. The modification accurately calculates *mlp_cost* considering LLC misses *on per core basis instead of total LLC misses from each core*. Consider an example of a 2-core system, where *core-0* is having one outstanding miss and *core-1* is having four outstanding misses at some point in time. Then, considering $N = 5$ as the total number of outstanding memory misses for *mlp_cost* calculation of all misses would be unfair to *core-0*, as it is facing one isolated miss. Without the modification, the isolated miss of *core-0* will be considered as a parallel miss and thus, the MLP-behavior per core will be wrongly understood.

3.3 Quantization of MLP-based cost

In our simulation environment, an isolated LLC miss takes 175 cycles to get serviced. We quantize *mlp_cost* into 3 bits (8 values), according to intervals shown in Figure 2(b). Quantized cost (*q_cost*) is proportional to the degree of parallelism. Note that *mlp_cost* is inversely quantized, as shown in Figure 2(b), to *q_cost* for suitability to AB-Aware policy. High *q_cost* of a block signifies that the block was brought in cache while servicing other parallel misses. A block brought in cache while servicing isolated miss will have low *q_cost*.

3.4 AB-Aware Policy

The proposed replacement algorithm takes into account both MLP-behavior (captured in the *q_cost*) and application's data-reuse behavior assimilated from recency information of each block (captured in the block's *ABr* value [7]). The victim of the *AB-Aware* policy is selected by a linear function block's recency value and its *q_cost* as

$$ABAr = Br + \alpha * Cr + \lambda * q\_cost \quad (1)$$
$$Victim = max\{Br(i) + \alpha * Cr(i) + \lambda * q\_cost(i)\} \quad (2)$$

In equation 1, *Br* represents that block's recency value, *Cr* captures the application's behavior and *q_cost* represents MLP-based cost of the block. A higher value of α gives more weight to core's behavior over block's recency behavior and similarly, a higher value of λ gives more weight to *q_cost*. We tried different combinations of λ and α (2, 4, 8) and experimentally found that the combination ($\lambda = 4$ & $\alpha = 4$) is optimum for the *AB-Aware* policy. The policy selects the victim block as per equation 2. Thus, the policy will protect cache blocks from the core which were isolated misses and showed high data-reuse behavior. The promotion, eviction and insertion policy of the proposed policy is explained below.

Promotion Policy: Hit to any block predicts that next reference of the block will be in the near future. Hence, the recency counter *Br* of the block is set to 0. The core which brought the block has also shown data-reuse and hence, the *Cr* counter corresponding to the block is also set to 0.

Eviction Policy On a cache replacement, block to be evicted is selected as per equation 2.

Insertion Policy Each incoming block is inserted at $Br = 2^M - 2$ location, which is closer to LRU.

4 EXPERIMENTAL METHODOLOGY

4.1 Simulator Infrastructure

We use Sniper [3] multicore x86 simulator as the evaluation platform. The baseline processor is 2.67 GHz, 4-wide fetch, 128-entry ROB, x86 Nehalem micro-architecture, with three level memory hierarchy as shown in Table 1. L3 miss penalty is 175 cycles. We evaluate 2-core configuration with 4MB, 8-way LLC and 4-core configuration with 8MB, 8-way LLC. MSHR supports up to 32 outstanding misses.

We use weighted speedup to evaluate our results, as it is a commonly used metric to quantify the performance of multi-core systems [11] [12].

4.2 Benchmarks

We use workloads from SPEC CPU2006 benchmark suite [4] for our evaluation. Using Pin-Points [8], 250M instruction traces of representative regions of these workloads were collected. We use four such traces of each application, i.e., 1 Billion instructions per application, for 2-core and 4-core

L1-D Cache	32KB, 4-Way, Private, 4-cycles, LRU
L1-I Cache	32KB, 4-Way, Private, 4-cycles, LRU
L2 Cache	256KB, 8-Way, Private, 8-cycles, LRU
L3 Cache	2MB per core, 8-Way, Shared, 30-cycles

Table 1: Cache hierarchy of the simulated system

Application	*apki*	*mpki*	Characteristics
bzip	15.44	1.07	cache-friendly (CF)
gcc	28.32	7.24	cache-friendly (CF)
soplex	37.05	14.64	cache-friendly (CF)
sphinx	13.44	4.35	cache-friendly (CF)
lbm	31.92	31.57	Streaming (STR)
libquantum	49.45	35.39	Streaming (STR)
mcf	96.12	56.34	Streaming (STR)

Table 2: Workloads under evaluation

configurations. Recent works on shared LLC [11] [6] [7] have followed the similar methodology.

To identify memory characteristics of workloads under evaluation, we run each application with single core configuration under LRU policy. As shown in Table 2, some applications like *bzip*, *sphinx* have low *apki* (accesses per kilo instructions) & low *mpki* (misses per kilo instructions). When these applications are allotted more cache resources, they show commensurate performance improvements; we label them as cache-friendly (CF). However, some applications like *libquantum*, *mcf* have very high *apki*. They have either very large working data set, which does not fit in cache (results into thrashing), or streaming access pattern [5]; we label them as Streaming (STR).

We compare our results with LIN [10] having $\lambda = 4$, SRRIP [6] having 2-bit RRPV and with ABRIP [7] having 2-bit *Cr* RRPV and $\alpha = 4$. We use the same configuration as baseline ABRIP with 3-bit MLP-cost and $\lambda = 4$.

5 RESULTS AND ANALYSIS

5.1 2-core configuration

We evaluate the proposed replacement policy on 18 combinations of workloads under evaluation. In subsequent performance plots, first 12 workload mixes are of *CF-STR* category, and rest 6 are *CF-CF* mixes. We do not consider *STR-STR* workload mixes, as their performance remains unaffected irrespective of policy used.

Figure 3 shows the comparison of weighted speedup achieved by SRRIP, LIN, ABRIP and AB-Aware. *sphinx-gcc* achieve weighted speedup of 1.988, which is almost equal to the ideal value of weighted speedup (2 for dual-core and 4 for quad-core). LIN, which is MLP aware replacement with LRU as the baseline, performs poorly in all the workload mixes. LIN is MLP-aware, however, it is application behavior un-aware. *lbm* benchmark performs poorly with AB-Aware policy due to the poor predictability of the *mlp_cost* for *lbm's* blocks. AB-Aware policy implicitly assumes that the blocks which had high *mlp_cost* at the time they were brought in the cache will continue to have high *mlp_cost* in their successive access. However, blocks belonging to *lbm* showed large variability in the *mlp_cost* of the blocks across successive accesses. Hence, all workload mixes having *lbm* as one the benchmarks perform poorly with AB-Aware policy. While ABRIP worsens weighted speedup compared to SRRIP, we achieve 1.69% improvement on an average for all mixes.

Figure 4 shows *mpki's* of workload mixes for AB-Aware policy normalized to SRRIP. *Workload mixes having bar less than 2 perform better for AB-Aware policy.* A maximum of 69.22% of reduction is achieved by *bzip* while running with

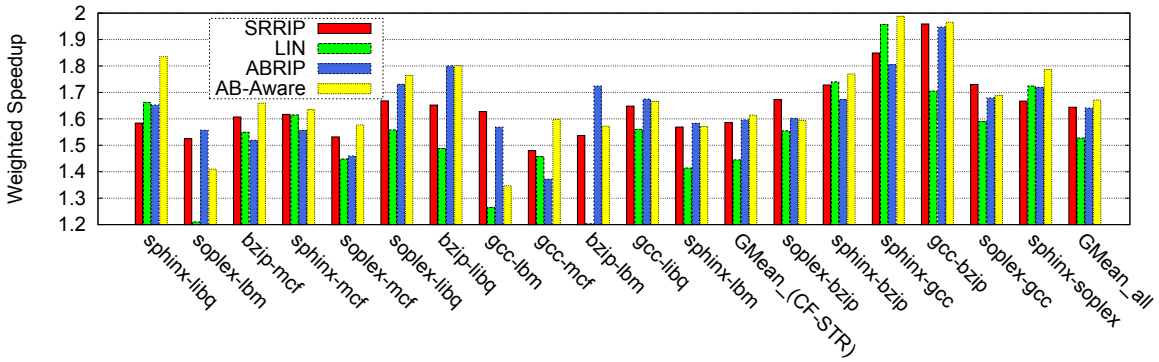


Figure 3: Weighted speedup comparison for 2-core configuration

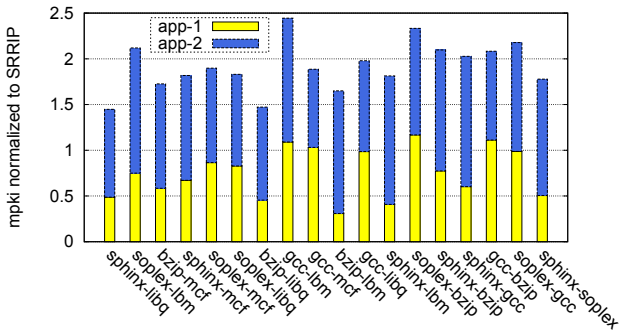


Figure 4: *mpki* reduction of AB-Aware over SRRIP

Workload	Applications	Category
Mix_1	soplex-sphinx-gcc-libq	CF-CF-CF-STR
Mix_2	bzip-gcc-sphinx-lbm	
Mix_3	sphinx-bzip-gcc-mcf	
Mix_4	bzip-soplex-gcc-lbm	
Mix_5	gcc-soplex-sphinx-mcf	
Mix_6	sphinx-soplex-bzip-lbm	
Mix_7	bzip-sphinx-libq-mcf	CF-CF-STR-STR
Mix_8	gcc-sphinx-mcf-libq	
Mix_9	sphinx-bzip-lbm-mcf	
Mix_10	bzip-gcc-libq-lbm	
Mix_11	bzip-gcc-lbm-mcf	
Mix_12	soplex-bzip-libq-lbm	
Mix_13	soplex-libq-mcf-lbm	CF-STR-STR-STR
Mix_14	sphinx-libq-lbm-mcf	
Mix_15	bzip-gcc-sphinx-soplex	CF-CF-CF-CF

Table 3: Workloads under evaluation for quad-core

libq. In CF-STR mixes, *soplex-lbm* and *gcc-lbm* see increased *mpki*, mainly because of misses for *lbm* increase by around 35% due to poor *mlp_cost* predictability as mentioned in the previous paragraph. Even though workload mixes like *sphinx-bzip* and *sphinx-gcc* observe total normalized *mpki* more than 2, they gain in performance over SRRIP as shown in Figure 3 as in the proposed policy *mlp_cost* component sometime increases the total number of misses. However, in such cases, the number of reduced isolated misses overshadow increase in the number of parallel misses.

5.2 4-core configuration

In order to investigate the scalability of our policy, we evaluate our replacement policy on 15 workload mixes (shown in Table 3) with 8 MB LLC shared among four cores. These workload mixes include the following categories: six CF-CF-CF-STR, six CF-CF-STR-STR, two CF-STR-STR-STR and one CF-CF-CF-CF.

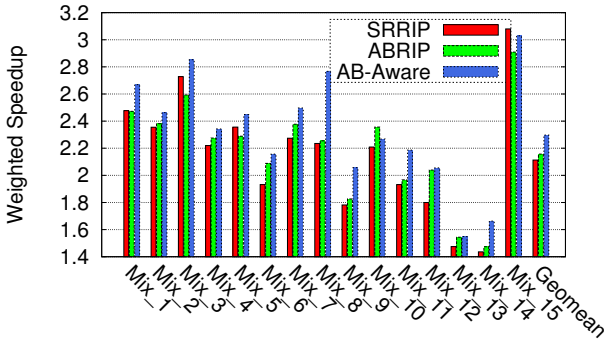


Figure 5: Weighted speedup comparison for 4-core configuration

Figure 5 shows the comparison of weighted speedup achieved by SRRIP, ABRIP and AB-Aware. AB-Aware outperforms SRRIP in all workload mixes, except Mix_15 (CF-CF-CF-CF). For Mix_15, AB-Aware performs poorly as the policy finds it difficult to allocate cache resources among all CF workloads. AB-Aware outperforms ABRIP in 14 out of total 15 workload mixes and also outperforms SRRIP by 8.71% on an average. As we achieve better performance gains for 4-core system over 2-core system, it shows that AB-Aware policy is scalable and will perform better as the number of cores sharing LLC increases.

5.3 Hardware Overhead

Table 4 details the storage overhead incurred by AB-Aware policy for 2-core and 4-core configurations over SRRIP. 2-core configuration with 4 MB shared LLC has 8192 sets and 4-core configuration with 8 MB shared LLC has 16384 sets. We use 2-bit saturating counter *Cr* as core RRPV, which accounts for 2 bits of storage overhead per core per set. MLP cost (3-bit) is stored in tag-store array, which has a overhead of $(3\ bits * 8\ ways) = 24\ bits$ per set. Also, block RRPV *Br* uses additional 2 bits per set per way, which has a overhead of 16 bits per set. AB-Aware has 0.6% overhead (incurred due to MLP bits) over ABRIP for both 2-core and 4-core configurations.

Overhead	2-core	4-core
Core RRPV (2 bits per core per set)	4 KB	16 KB
Block RRPV bits (16 bits per set)	16 KB	32 KB
MLP Cost storage (24 bits per set)	24 KB	48 KB
Net overhead	44 KB	96 KB
% overhead over LLC	1.1%	1.2%

Table 4: Storage overhead of AB-Aware over SRRIP

6 RELATED WORK

Both academic and industrial research communities have made significant contributions in the development of efficient cache management policies. We summarize prior research work in the area of improving LLC performance through efficient management.

Qureshi *et al.* [10] proposed a low-cost run-time mechanism to compute MLP-cost of in-flight misses and uses the MLP-cost to reduce isolated misses. This replacement scheme uses LRU as its baseline policy. Hence, the policy is not efficient for a multicore system when a STR application high data demand is running with a CF application having low data demand. In such a case, STR application will thrash the data of CF from the shared cache. Qureshi *et al.* proposed a Utility-based Cache Partitioning (UCP) [11] scheme to minimize the miss rate of all the competing applications. It uses Utility Monitor (UMON) counters to find the utility of providing extra *ways* to an application. Based on statistics received from each UMON, replacement policy logically partitions the cache to have the lowest number of misses.

Qureshi *et al.* [9] show that on an average more than 60% of the cache lines are not re-referred before getting evicted as LLC receives filtered data locality. Such *zero reuse* cache lines adversely affect the performance of other applications under LRU policy, as these cache lines spend time traversing from MRU to LRU position before eviction, and therefore consume cache resources without any returns in terms of reuse. *LRU Insertion Policy (LIP)* [9] inserts cache blocks directly at LRU position, so they get evicted without spending much time in cache.

Jaleel *et al.* [6] proposed *Static Re-reference Interval Prediction (SRRIP)*, which predicts re-reference intervals of cache blocks. Instead of maintaining recency counters, SRRIP maintains M-bit saturating counter, which they call *Re-Reference Prediction Value (RRPV)*. $RRPV = 0$ denotes near re-reference interval and $RRPV = 2^M - 1$ denotes distant re-reference interval. SRRIP inserts blocks at $RRPV = 2^M - 2$, which is intermediate re-reference value. Upon hit, the block is promoted to $RRPV = 0$ (promotion policy), which is equivalent to MRU position. SRRIP tries to preserve the data-sets of the application showing good locality, however, some workloads have very high access rates compared to others. Such workloads may still interfere with cache blocks of recency-friendly workloads, especially, if they have low access rates. STR applications may produce misses with high rate, and CF blocks may get evicted before getting the first hit. To address this issue, Lathigara *et al.* [7] propose to add one more RRPV counter at core/application level in *Application Behavior-aware Re-reference Interval Prediction (ABRIP)* policy. However, ABRIP doesn't perform well when workload mixes are CF-CF.

7 CONCLUSIONS & FUTURE WORK

LLC is generally shared by multiple applications having diverse nature and different access pattern. Applications also have different MLP-behavior. Some applications have more parallel misses, whereas other applications have many isolated misses. Conventionally used LRU-based cache replacement policies are unaware of such application behaviors. We proposed an application behavior aware replacement policy AB-Aware. It considered both application's MLP-behavior and data-reuse behavior to manage shared LLC. The proposed policy reduced the negative interference among applications by assimilating each block's recency information to understand application's behavior as a whole. It also reduces the miss-penalty associated with each LLC miss. Experimental results showed up to 15.85% performance improvement (1.69% on an average) for 2-core configuration and up to 23.8% IPC improvement (8.71% on an average) for 4-core configuration over SRRIP.

We plan to investigate the benefits of using Auxiliary Tag Directory (ATD) to dynamically choose the best performing policy among SRRIP and AB-Aware as in few workload mixes SRRIP outperforms AB-Aware policy. AB-Aware policy increases the total number of misses for few workload mixes, we plan to study the energy cost for these increased misses and increased activity in MSHR due to updates in every cycle.

REFERENCES

[1] L. A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Syst. J.* 5, 2 (June 1966), 78–101.

[2] Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Commun. ACM* 54, 5 (May 2011), 67–77.

[3] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM TACO* (2014).

[4] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.

[5] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. 2008. Adaptive Insertion Policies for Managing Shared Caches. In *PACT-17*. 208–219.

[6] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *ISCA-37*. 60–71.

[7] P. Lathigara, S. Balachandran, and V. Singh. 2015. Application Behavior Aware Re-reference Interval Prediction for Shared LLC. In *ICCD-33*. 172–179.

[8] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. 2004. Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation. In *Micro-37*. 81–92.

[9] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *ISCA-34*. 381–391.

[10] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. In *ISCA-33*. 167–178.

[11] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Micro-39*. 423–432.

[12] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. PACMan: Prefetch-aware Cache Management for High Performance Caching. In *Micro-44*.