A Fairness Conscious Cache Replacement Policy for Last Level Cache

Kousik Kumar Dutta
Department of CSE
Indian Institute of Technology Ropar, Punjab, India.
2019csm1018@iitrpr.ac.in

Prathamesh Nitin Tanksale
Department of CSE
Indian Institute of Technology Ropar, Punjab, India.
2019csm1017@iitrpr.ac.in

Shirshendu Das
Department of CSE
Indian Institute of Technology Ropar, Punjab, India.
shirshendu@iitrpr.ac.in

***Abstract*—Multicore systems with shared Last Level Cache (LLC) possess a bigger challenge in allocating the LLC space among multiple applications running in the system. Since all applications use the shared LLC, interference caused by them may evict important blocks of other applications that result in premature eviction and may also lead to thrashing. Replacement policies applied locally to a set distributes the sets in a dynamic way among the applications. However, previous work on replacement techniques focused on the re-reference aspect of a block or application behavior to improve the overall system performance. The paper proposes a novel cache replacement technique *Application Aware Re-reference Interval Prediction (AARIP)* that considers application behavior, re-reference interval, and premature block eviction for replacing a cache block. Experimental evaluation on a four-core system shows that AARIP achieves an overall performance improvement of 7.28%, throughput by 4.9%, and improves overall system fairness by 7.85%, as compared to the traditional SRRIP replacement policy.**

***Index Terms*—Last Level Cache, Replacement Policy, Application Behavior, Re-reference Interval.**

I. Introduction

With the increase in core count, traditional cache replacement policies for LLC are insufficient to scale and meet the increasing demand for various applications. Generally, applications are either cache friendly (Cf) or non-cache friendly (non-Cf). Cf posses a good locality of reference, and hence they effectively utilize the cache space. On the other hand, in non-Cf applications, the working set does not fit into the limited cache space. Therefore, it results in generating frequent cache misses fetching the required block. Streaming applications are an example of non-Cf applications.

In a multicore system, a combination of Cf and non-Cf applications may run on different cores that share the LLC. But the traditional replacement policies have various limitations when both Cf and the non-Cf application run concurrently in the system [1]–[3]. A non-Cf application may cause cache pollution by evicting important blocks of a Cf application. Evicting such blocks from the LLC results in evicting the block from all levels of caches. Hence, re-requesting such blocks is subsequently fetched from the main memory, which is costly. Thus, the application suffers from a long waiting time before resuming its execution. Non-Cf applications may capture more than their required space. Thus, application interference in shared LLC affects system performance. Even if multiple caches are used, LLC's effective utilization is still a major issue that needs to be addressed.

To reduce application level interference at the shared LLC, few issues must be taken care of: 1) Unfair usage of LLC space by different applications. 2) Cache pollution, and 3) Thrashing. Cache pollution and thrashing can be reduced only by fair distribution of LLC among the applications. Thus, Problem 2 and 3 directly depend on Problem 1 for a solution. Cache partitioning [4] and replacement policies [1]–[3], [5], [6] provide different techniques that may solve the unfair usage of LLC space. However, cache partitioning techniques strictly use the same partition across all the LLC sets. Since the access to the LLC sets is not the same, it makes the LLC prone to evict important blocks prematurely from the non-monitored cache sets. Thus, a varying partition that is local to a set may help applications to perform better. Replacement policies, on the other hand, are local to a set. Hence, a novel replacement technique is a better choice that takes care of the varying usage of cache sets by the applications.

The paper proposes a novel replacement technique *AARIP*, that takes care of the application behavior, re-reference interval, and premature eviction of cache blocks from the LLC. It restricts an application from replacing cache blocks of other applications to a certain level, thereby reducing cache pollution. The eviction and insertion policy of AARIP makes it application-aware. AARIP also uses a on-chip data structure *Evicted block Information Storage (EbIS)* that predicts the important blocks that are evicted prematurely. Based on this prediction, the block is inserted into LLC with higher insertion priority, during its next fetch from the main memory.

The paper makes the following contributions:

1) The paper identifies the limitations of existing techniques: SRRIP [1], TADRRIP [1], ABRIP [3] and EAF [7].
2) The paper proposes a novel replacement policy that ensures fair usage of LLC among the applications.
3) We have experimented with real benchmarks from SPEC CPU 2006 [8] benchmarks in a multicore system.

The paper is organized as follows. The next section describes related works in this domain. Section III discusses the motivation behind this work. The proposed replacement technique is discussed in Section IV followed by the experimental analysis in Section V. Finally, we conclude the paper in Section VI.

978-3-9819263-5-4/DATE21/©2021 EDAA

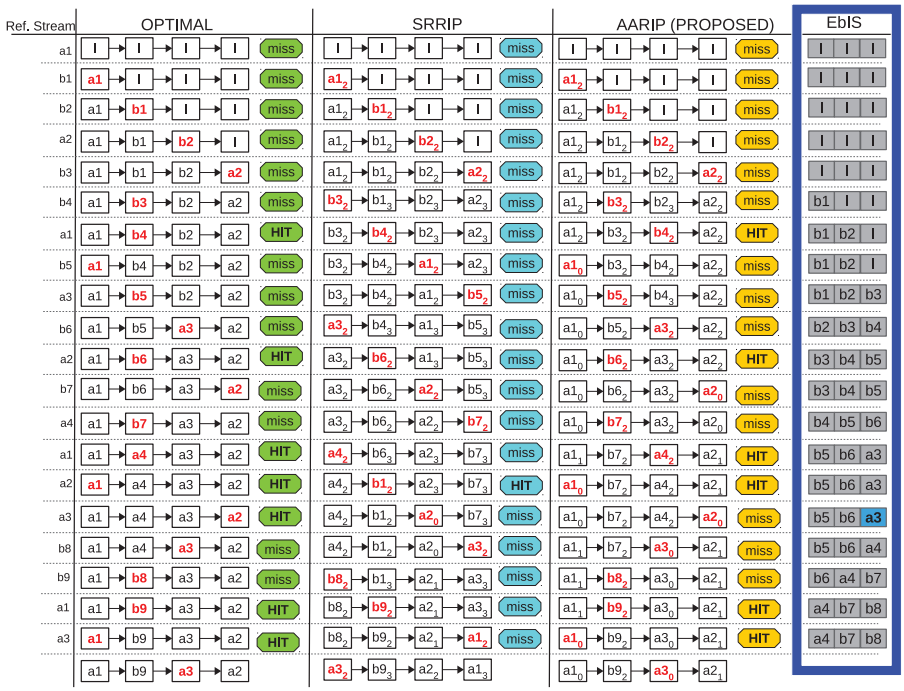


Fig. 1: Behaviour of OPT, AARIP and SRRIP using a Cf (A:{a1 ...a4}) and non-Cf application (B:{b1 ...b9)}. The LLC reference stream is {a1, b1, b2, a2, b3, b4, a1, b5, a3, b6, a2, b7, a4, a1, a2, a3, b8, b9, a1, a3} ai_x, bj_y indicates the RRPV of cache blocks. SRRIP and AARIP here use M=2 to store the RRPV.

II. Related Work

Belady's optimal replacement policy (OPT) [9] performs cache block replacement by making use of the re-reference duration of a block to select a victim such that it is referenced farthest in time. Jaleel et al. proposed Static Re-reference Interval Prediction (SRRIP) [1] that uses Re-reference Prediction Values (RRPV), an M-bit value for each cache block for replacement. A block is initially fetched with an RRPV of $(2^M - 2)$. Upon a cache hit, the RRPV of a block is set to 0 (highest priority). On a miss, if the set is full, SRRIP selects a block of lowest priority, i.e., RRPV = $(2^M - 1)$ as a victim. If multiple such victim block exists, the block with the minimum way number is selected as a victim. Lathigara et al. proposed Application Behavior-aware Re-reference Interval Prediction (ABRIP) [3] that fairly distributes LLC ways among the application using a combination of core RRPV and block RRPV similar to SRRIP. However, ABRIP does not consider the premature block eviction from LLC. Jaleel et al. proposed the Thread-Aware Dynamic Insertion Policy (TADIP) [2] that uses the memory requirements of applications by monitoring a few of the cache sets and selects either of LRU [10] or BIP [5] replacement policy that is implemented uniformly throughout the cache.

To overcome cache pollution and thrashing in SRRIP, a thread-aware dynamic RRIP (TADRRIP [1]) is proposed, which uses Set Dueling Monitors to observe the application behavior and selects an appropriate replacement policy. Seshadri et al. proposed Evicted-Address Filter (EAF) [7], which queues the evicted blocks from LLC. Whenever a cache miss results in an EAF hit, it implies that an important block got evicted prematurely. Such blocks are restored with high priority, thereby making the underlying replacement policy thrashing resistant. However, EAF does not consider the application behavior, and the hardware requirement of EAF is also high.

III. Motivation

Different applications require different amounts of memory in a shared LLC. The underlying replacement policy should carefully promote/evict/inject blocks to caches such that it does not harm other applications present in the system. We explain this using Figure 1. The figure shows the performance of OPT, SRRIP, and AARIP executing a few LLC accesses. In this section, we explain only OPT and SRRIP policy.

For simplicity, consider two applications: *A* and *B* that are running in a dual-core system with a 4 way associative LLC. Application *A* is Cf whereas *B* is a non-Cf application. The memory access from both *A* and *B* is represented as *ai* and *bi*, respectively. The reference stream used in the figure maps in the same set, *S*. Initially, we assume that the cache is empty or invalid. Hence, the first four cache block accesses are cold misses, and all the three techniques behave the same. From fifth block access, b3, the behavior of the technique changes.

OPT: Since the cache is full when b3 is brought in the cache; it needs to select a victim block. OPT replaces b1 with b3 as b1 is not referenced for a longer period. Similarly, for the next access b4, it replaces b3 again and so on. After executing the

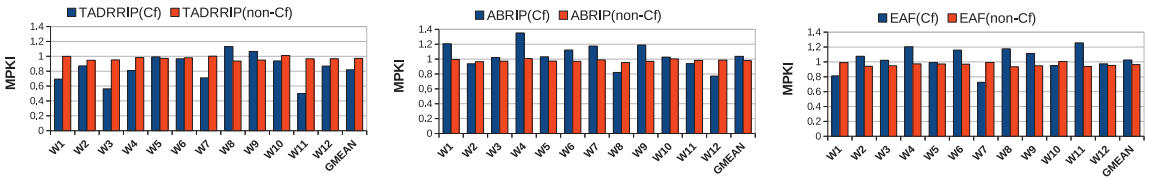


Fig. 2: Misses Per Kilo Instructions (MPKI) observed in different replacement policies normalized to SRRIP.

complete reference stream, OPT achieves a total of 7 cache hits and 13 misses. But OPT is an ideal replacement policy theoretically, and it is not application-aware.

SRRIP: As per the insertion policy of SRRIP, the first four blocks are inserted with RRPV as 2 (2^M-2). In the fifth access, b3 replaces a victim block selected by SRRIP. Thus, the RRPV of all blocks is incremented until a single cache block has a maxRRPV of (2^M-1). In this case, all the four blocks residing in the cache now has an RRPV of 3. Thus, the tie is broken by selecting the first block, i.e., a1. In the next access b4, it replaces b1, as shown in the figure. Now in the subsequent access when a1 is re-referenced, it resulted in a cache miss. Thus, a1 is fetched, and it replaces b2 and so on. Finally, SRRIP achieves a total of 19 cache misses and only one cache hit. From the figure, we can observe that this happened because blocks from application B replaced all A blocks. Since A is a Cf application, its performance is degraded because of the thrashing nature of B.

Figure 2 shows Miss Per Kilo Instruction (MPKI) for TADRRIP, ABRIP, and EAF compared to SRRIP. We experimented on a dual-core system running 12 workloads from SPEC CPU 2006 suite with the cache parameters, as mentioned in Table I. We have also used EAF [7] for experimental analysis, which takes care of cache pollution caused by applications. From the figure, we can observe that in TADRRIP, the MPKI of Cf and non-Cf applications reduces by an average (geometric mean) of 18% and 2.7% compared to SRRIP. We can also observe that in ABRIP and EAF, the MPKI for Cf applications is greater than that of non-Cf applications compared to the SRRIP replacement policy. Though ABRIP and TADRRIP are application-aware replacement policies, only TADRRIP performs better than SRRIP by slightly reducing the overall MPKI.

Though SRRIP uses the re-reference duration of a block, it is not an application-aware replacement policy. This makes SRRIP unable to handle thrashing and cache pollution caused by the variable nature of applications running in the system. Thus, we require a new replacement policy that takes all three concepts, i.e., re-reference duration, application behavior, and premature block eviction into account for selecting an optimal cache block for replacement, thereby improving system performance in terms of WS.

IV. Application-Aware Re-Reference Interval Prediction (AARIP)

AARIP uses (a) re-reference interval prediction, (b) application behavior, and (c) premature block eviction to select an appropriate cache block for replacement. The re-reference interval of a cache block is maintained using *M-bits* per cache block known as the re-reference prediction value (RRPV). The usage of RRPV in AARIP is similar to that of SRRIP. When the RRPV of a block is 0, it indicates that the block will be re-referenced soon, while a higher value of RRPV indicates that the block will be referenced in the distant future. Premature block eviction is taken care of by a data-structure, *Evicted block Information Storage* (EbIS). EbIS reduces cache pollution by capturing the address of the prematurely evicted important blocks. Such blocks, when requested again, are inserted into the LLC with higher insertion priority. This is done by assigning appropriate RRPV values to cache blocks as per applications' priority. It is explained in greater detail in subsection IV-A. The eviction and insertion policy of AARIP is made as application-aware that adjusts with applications' varying nature.

1) Insertion and promotion policy of AARIP: AARIP uses the following insertion policy in a cache. If an empty location (invalid) is present in the cache set, place the new block in that location with RRPV = $2^M - 2$. Otherwise, when the cache is full, find a victim block. A victim block is selected using the eviction policy, as discussed in subsection IV-2. Upon determining a victim block, replace the victim block with the new incoming block and perform the following operation:

a) If the incoming block's information is not present in EbIS, use RRPV = 2^M -2.
b) If information of the incoming block is present in the EbIS, use RRPV $= 0$. This indicates that an important block got evicted from the cache untimely due to some thrashing applications running together in the system. Thus, the blocks are placed at a higher priority.

Whenever access to a cache block results in a hit, the RRPV of the block is promoted as 0.

2) Eviction Policy: The eviction policy of AARIP makes it application-aware. If there is a new incoming block from an application X and the set is full, a victim is selected as follows.

a) If there exists at least one cache block from the same application, X, already residing in the set with maximum RRPV ($maxRRPV = 2^M - 1$), select its block as a victim. This prevents an application from polluting the set by removing another application's block.
b) Otherwise, if there exists a block with *maxRRPV*, but it does not belong to X, select the block as a victim. This helps in increasing the cache requirement for a Cf application. Since non-Cf applications lack temporal locality and every new access is fetched from the main

memory, the number of cache ways allotted to such application is immaterial. Hence allocating one way per cache set or all the ways of the set makes no difference. In case multiple blocks from different applications have *maxRRPV*, we choose the victim application using a round-robin policy. Therefore, the cache sets pre-occupied by a non-Cf application are released using this step and are allotted to an application that is either Cf or has a mixed pattern where the blocks are re-used.

c) If no blocks with *maxRRPV* are present, increment the RRPV of all blocks that belongs to the same application and is currently present in the same set. This rule assures that many cache misses of one application can not increase the re-reference distance of the block that belongs to the other applications.

A. Evicted Block Information Storage (EbIS)

EbIS is an on-chip module, used by AARIP to store the information of a few evicted blocks from the LLC. Each block information such as tag, set and application number (to which it belongs) is stored. Note that the actual data of the block is not stored in EbIS. The applications are numbered from $0 \ldots n$, where n corresponds to the core where it is running. When a block is evicted from the LLC, the information of such blocks are entered into EbIS. Since EbIS is of fixed size, it has to remove an evicted block information when it is full to make room for the new evicted blocks. For this purpose, AARIP initially selects a target application whose block is removed using Equation 1. In the equation, $totalBlocks$ is the total blocks for an application stored in EbIS, and *minBlockDistance* is the minimum distance of a block of the particular application from the front of EbIS. Once the application is selected, the oldest block of the selected application is removed from *EbIS*.

$$target = max(totalBlocks - minBlockDistance) \quad (1)$$

Significance of Equation 1: The application that has the maximum *target* value is the one occupying maximum space in EbIS. The parameter *totalBlocks* indicates how frequently the blocks of an application are evicted. This indirectly measures the locality information on different applications. On the other hand, *minBlockDistance* helps to identify blocks that have been evicted from the LLC long back. If a block is requested and its information is present in the EbIS, it indicates an important block is evicted from the cache. Thus, the equation captures the overall block access pattern and tries to maintain fairness among the applications by placing premature evicted blocks at a higher priority with RRPV $= 0$. As the eviction policy of AARIP takes care of the application behavior, the small size of EbIS is found to be productive enough in detecting premature evicted blocks. The ideal size of EbIS is determined experimentally as 128 entries, as described in Section V-A.

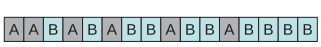


Fig. 3: Representative example explaining the deletion of blocks from EbIS that contains blocks from application A and B.

We explain the deletion of blocks from EbIS using Figure 3. In the figure, EbIS is full of 6 blocks of application *A* (grey colored) and 10 blocks of *B* (green colored) at a particular instance. Now, if a block is evicted from LLC, an existing block must be evicted from EbIS. According to Equation 1, $totalBlocks$ for *A* and *B* are 6 and 10; *minBlockDistance* for *A*, *B* is 0 and 2, respectively. Using equation 1, the target value for *A*, *B* is 6 and 8, respectively. Thus, the first block of *B* is removed from the head of the EbIS and the newly evicted block from LLC is stored at the end.

TABLE I: Simulation parameters

Processor	4, X86 cores OoO superscalar, 3GHz
L1I cache/core (private)	32KB, 8-way, 64B block, 1 cycle
L1D cache/core (private)	32KB, 8-way, 64B block, 4 cycles
L2 cache/core (private))	512KB, 8-way associative, 64B block, 10 cycles
L3 cache (LLC)	4MB, 16-way associative, 64B block, 20 cycles
DRAM configuration	DDR3, 4GB, 64-bits channel.

TABLE II: Workloads generated from SPEC CPU 2006 suite

mixes	benchmarks	mixes	benchmarks
w1	bzip2,gcc,h264ref,gemsFDTD	w2	bzip2,gcc,libquantum,mcf
w3	calculix,bzip2,bwaves,astar	w4	calculix,gamess,astar,gemsFDTD
w5	calculix,gamess,hmmer,bwaves	w6	calculix,gromacs,povray,wrf
w7	gamess,bzip2,perlbench,astar	w8	gamess,gcc,cactusADM,libquantum
w9	h264ref,sjeng,gobmk,lbm	w10	hmmer,gcc,bwaves,gemsFDTD
w11	hmmer,h264ref,gobmk,mcf	w12	hmmer,namd,omnetpp,soplex
w13	leslie3D,milc,povray,soplex	w14	namd,gcc,lbm,libquantum
w15	namd,sjeng,lbm,gemsFDTD	w16	namd,sphinx3,tonto,astar
w17	sjeng,xalancbmk,zeusmp,wrf	w18	sphinx3,tonto,leslie3D,milc
w19	sphinx3,xalancbmk,leslie3D,omnetpp	w20	tonto,zeusmp,milc,soplex

V. Experimental Analysis

We have experimented with AARIP with the machine configuration, as mentioned in Table I. The performance of AARIP is compared with state-of-the-art replacement policies SRRIP [1], TADRRIP [1], ABRIP [3] and EAF [7]. All the techniques are implemented in ChampSim [11]. ChampSim has been used in data prefetching championships [12]. It is enhanced with a multilevel of caches and is effective to compare recent prefetching and cache replacement techniques. For our experiments, the caches are warmed up for 1 million instructions and then execute for the next 200 million to calculate the performance metrics. We use real benchmarks from SPEC2006 [8]. Here we use 20 different combinations as mentioned in TABLE II to evaluate AARIP on a 4-core system. We also evaluate our algorithm in two-core and eight-core systems with a combination of 20 mixes each. We assume that only one application is running on each core. We also evaluate the effect of prefetching on our algorithm.

The performance metrics used to evaluate different techniques are weighted speedup (WS), Throughput, and Harmonic Mean Fairness (HMF). WS indicates the overall system performance in a shared system. It is calculated as $WS = \sum_{i=0}^{P-1} \frac{IPC_i^{together}}{IPC_i^{alone}}$; IPC_i^{alone} denotes instruction per cycle (IPC) of an application when no other applications are running and $IPC_i^{together}$ is IPC achieved when multiple applications are running concurrently in P cores. HMF [13] is used to measure how fairly the resources are distributed among the applications running concurrently in

a multicore system with P cores. It is calculated as $HMF = P/\sum_{i=0}^{P-1} \frac{IPC_i^{alone}}{IPC_i^{together}}$ and throughput is calculated by summing the IPC of all the cores.

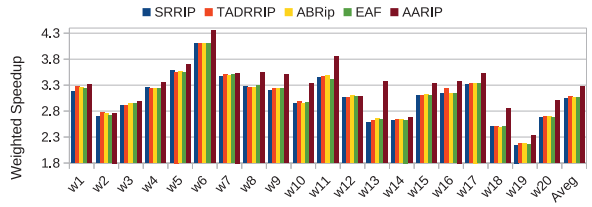


Fig. 4: Weighted Speedup.

A. System Performance: Figure 4 shows the system performance in terms of WS. In the figure, X-axis denotes different workloads and Y-axis contains the average WS obtained for different techniques. From the figure, we can observe that SRRIP, TADRRIP, ABRIP, EAF, and AARIP achieves an average (geometric mean) WS of 3.049, 3.073, 3.069, 3.062, and 3.271, respectively. AARIP achieves the highest WS as compared to the others. On average, the percentage improvement in WS for AARIP is 7.28%, 6.44%, 6.58%, and 6.83% as compared to SRRIP, TADRRIP, ABRIP, and EAF, respectively. As an interesting result, we can observe that the maximum WS obtained by AARIP is for workload W13, i.e., 30.59% over SRRIP. W13 is a mix of *leslie3d-milc-povray-soplex*. Out of this combination *leslie3d, milc,* and *soplex* are non-Cf applications that generate higher misses while *povray* is a Cf application. When the combination of Cf and non-Cf runs concurrently in the system, AARIP fairly distributed the LLC space among these applications, resulting in an increased WS for the workload as compared to SRRIP. This shows that AARIP justifies both Cf and non-Cf applications on efficiently sharing the LLC, as observed in Figure 6.

W12, on the other hand, provides the least improvement of 0.5% WS as compared to SRRIP. W12 is a workload mix of *hmmer-namd-omnetpp-soplex*. Among the benchmarks *hmmer* and *namd* are Cf-applications while the rest two are non-Cf applications. But *omnetpp* comparatively generates fewer misses as compared to *soplex*. But it can be observed from Figure 5 that throughput increases for the workload. This indicates that AARIP balances the LLC space between different types of benchmarks. Thus, the overall improvement by AARIP across all the workloads, which has a wider range of applications running concurrently, makes it a better choice than the others.

B. System Throughput: Figure 5 shows the performance of AARIP as compared to different techniques for the throughput metric. Since the throughput metric implies how fast the core is executing a particular application or a set of applications running concurrently, from the figure, we can observe that AARIP achieves the highest throughput compared with other techniques. On average, the percentage improvement in throughput for AARIP is 4.9%, 5.16%, 5.32%, and 6.19% as compared to SRRIP, TADRRIP, ABRIP, and EAF, respectively.

C. System Fairness: Fairness is calculated to show that

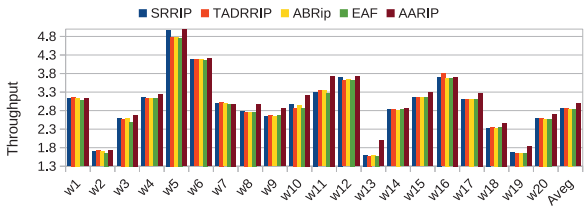


Fig. 5: AARIP performance on shared LLC.

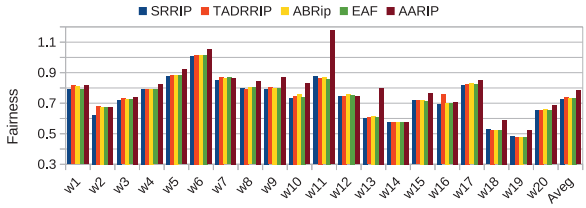


Fig. 6: Harmonic mean fairness obtained in different techniques.

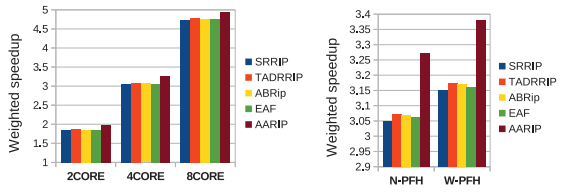
AARIP does not prioritize one application over the other to have an overall improvement in system performance. Figure 6 shows the percentage of fairness improvement of AARIP over SRRIP, TADRRIP, ABRIP, and EAF. From the figure, we can observe that AARIP achieves an average of 7.85%, 6.40%, 6.69%, and 7.15% improvement in fairness over SRRIP, TADRRIP, ABRIP, and EAF, respectively. This shows that AARIP unnecessarily does not prioritize one application over the other to get an overall performance improvement in a shared system. This shows that AARIP aims to maintain the shared cache in a more balanced way among different applications that fairly share LLC's cache space without causing cache pollution and thrashing.

A. Sensitivity Analysis

Multiple cores: We also evaluate AARIP for two-core as well as eight-core system. Figure 7(a) compares the average WS of AARIP with other techniques in the case of systems having 2, 4, and 8 cores. The average is calculated after executing multiple workloads. We get the WS improvement of 7.08%, 6.74%, 7.16%, 7.04% for two-core and 4.48%, 3.3%, 3.57%, 3.91% for eight-core over SRRIP, TADRRIP, ABRip, and EAF respectively.

Prefetching Enabled: Figure 7(b) shows the impact of prefetching in AARIP as well as in other existing techniques. We have used the next-line prefetcher at LLC to analyze the impact. The figure only shows the average WS of multiple workloads. It can be observed that AARIP performs better than other techniques with both prefetching enabled/disabled.

EbIS size vs. cache size: Table III shows the sensitivity of EbIS size concerning different LLC sizes in MBs for a 2-core system. In the table, each row shows that on a particular LLC size, the WS obtained in AARIP for different EbIS sizes. Here EbIS size indicates the number of evicted block entries in the data-structure. For a 2MB shared LLC, the maximum speedup obtained is 1.673 for an EbIS size of 128 entries. Similarly, for a 4*MB* LLC size, the maximum speedup is obtained for EbIS



(a) Multicore analysis (b) Prefetch analysis

Fig. 7: Effect of multiple cores and Prefetching. Here N-PFH means **no prefetch** and W-PFH means **with prefetch**.

TABLE III: Analysis of EbIS size on different cache sizes.

Cache Size	Different EbIS size			
	32	**64**	**128**	**256**
2MB	1.567	1.571	1.673	1.670
4MB	1.748	1.751	1.870	1.871
8MB	1.751	1.782	1.868	1.868

size 256, but the speedup is not significantly higher than EbIS size 128. This shows that with a small number of EbIS entries, i.e., 64 or 128, it can effectively capture the premature eviction of cache blocks from LLC. The EbIS size of 128 is also found suitable for 4-core systems.

TABLE IV: Storage overhead (over SRRIP) analysis considering LLC size as 4MB, 16-way associative.

	AARIP	**SHiP**	**MRP**	**Hawkeye**
2-core	0.44%	0.68%	1.29%	1.55%
4-core	0.46%	0.98%	2.16%	2.52%
8-core	0.48%	1.56%	3.85%	4.45%

B. Other analysis

AARIP assumes one application executing per core, similar to the ones used in TADIP [2], and UCP [4]. Thus, AARIP needs to store the core-id with each LLC block. The core-id of a block can be collected from the sharers' information of each block stored in LLC. The sharers' information is stored as a bit-vector, and it is required for maintaining cache-coherence. In the case of multi-programmed applications, a block belongs to a single core, but in the case of multi-threaded applications, a block can be shared by multiple cores. In that case, the shared block can be considered as a part of all the cores currently sharing it. When multiple applications run on a single-core, we may need some additional bits to uniquely identify which application a block belongs to. This will be explored as a part of future work. However, the significant hardware overhead of other techniques as compared to AARIP makes it a better choice, as discussed next.

We compared AARIP with recent replacement policies like SHiP [14], MRP [15], and Hawkeye [16] in terms of hardware requirements. The paper does not discuss these comparisons in detail as these techniques are not application-aware. It has been observed that the performance of AARIP is slightly less than (≈2%), but AARIP needs significantly less hardware than these techniques. Table IV shows the additional storage required in AARIP, SHiP, MRP, and Hawkeye over SRRIP for a 4MB, 16-way associative LLC. Assuming that the bits we need to identify a block belonging to a core in LLC are already present to support cache coherence, the extra hardware overhead required in AARIP over the traditional SRRIP policy is EbIS only. Similar to SRRIP, AARIP also requires $2 * n$ bits per cache set. As shown in the table, the other three techniques (SHiP, MRP, and Hawkeye) have significantly higher storage overhead than AARIP. Hence, our proposed technique AARIP is lightweight and can be used as an add-on for a system with low hardware requirements.

VI. Conclusion

Application-level interference caused by a combination of Cf and non-Cf applications in shared LLC can hamper system performance altogether. This paper proposes AARIP that considers application behavior and reduces cache pollution and thrashing in the system. AARIP improves the WS of the applications running concurrently in the system. We have used different performance metrics to analyze the overall performance improvement over existing replacement policies. AARIP shows that when other applications are running concurrently in a balanced way, the overall system performance improves. The low hardware overhead of AARIP makes it a practical replacement policy for use in modern multicore systems.

References

[1] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," vol. 38, no. 3, 2010, p. 60–71.
[2] A. Jaleel *et al.*, "Adaptive Insertion Policies for Managing Shared Caches," in *PACT*, 2008, pp. 208–219.
[3] P. Lathigara, S. Balachandran, and V. Singh, "Application Behavior Aware Re-reference Interval Prediction for Shared LLC," in *ICCD*, 2015, pp. 172–179.
[4] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, 2006, pp. 423–432.
[5] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *ISCA*, 2007, p. 381–391.
[6] S. Das and H. K. Kapoor, "Latency Aware Block Replacement for L1 Caches in Chip Multiprocessor," in *ISVLSI*, 2017, pp. 182–187.
[7] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *PACT*, 2012, p. 355–366.
[8] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006, pp. 1–17.
[9] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, p. 78–101, Jun. 1966.
[10] T. R. Puzak, "Analysis of Cache Replacement-Algorithms," Ph.D. dissertation, 1985.
[11] "ChampSim repository." in *https://github.com/ChampSim/ChampSim*.
[12] "3rd Data Prefetching Championship." in *[Online]. Available: https://dpc3.compas.cs.stonybrook.edu/final programs*.
[13] Kun Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors," in *ISPASS*, 2001, pp. 164–171.
[14] C. Wu *et al.*, "SHiP: Signature-based Hit Predictor for High Performance Caching," in *MICRO*, 2011, pp. 430–441.
[15] D. A. Jiménez and E. Teran, "Multiperspective Reuse Prediction," in *MICRO*, 2017, pp. 436–448.
[16] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *ISCA*, 2016, p. 78–89.