An Application-Aware Cache Replacement Policy for Last-Level Caches

Tripti S. Warrier, B. Anupama, and Madhu Mutyam

PACE Laboratory, Computer Science and Engineering Department,
Indian Institute of Technology Madras, Chennai, India-600036
{tripti,anupama,madhu}@cse.iitm.ac.in

Abstract. Current day multicore processors employ multi-level cache hierarchy with one or two levels of private caches and a shared last-level cache (LLC). Efficient cache replacement policies at LLC are essential for reducing the off-chip memory traffic as well as contention for memory bandwidth. Cache replacement techniques for unicore LLCs may not be efficient for multicore LLCs as multicore LLCs can be shared by applications with varying access behavior, running simultaneously. One application may dominate another by flooding of cache requests and evicting the useful data of the other application.

This paper proposes a new cache replacement policy for shared LLC called *Application-aware Cache Replacement* (ACR). ACR policy prevents victimizing low-access rate application by a high-access rate application. It dynamically keeps track of maximum life-time of cache lines in shared LLC for each concurrent application and helps in efficient utilization of the cache space. Experimental evaluation of ACR technique for 2-core and 4-core systems using SPEC CPU 2000 and 2006 benchmark suites shows significant speed-up improvement over the *least recently used* and *thread-aware dynamic re-reference interval prediction* techniques.

1 Introduction

Modern multi-core processors support multiple levels of cache to improve performance. Most often the LLC in such systems is shared among concurrent applications. Implementing an efficient LLC management policy is essential for reduction in off-chip memory traffic and bandwidth since it has a direct impact on power consumption of the system. One of the key features involved in cache management is the replacement policy. An ideal replacement policy will victimize cache lines that are accessed farthest in future and retain the data with high temporal locality [1]. But all practical cache replacement policies take victim selection decision by *predicting* the cache line that is going to be re-referenced farthest in future. The effectiveness of such replacement policies depends on the prediction accuracy.

Least recently used (LRU) policy is one of the most commonly used cache replacement techniques. LRU policy predicts near re-reference for a cache line accessed recently and distant re-reference for one without reference. There are several drawbacks with the LRU technique: 1) it looses the access history if it encounters a burst of references of length more than the associativity; 2) it can victimize a frequently accessed cache line over a less-frequently but recently accessed cache line; 3) it performs badly for working sets larger than the cache size; and 4) it may not be effective for multicore LLC as applications with varying access patterns share the LLC. Since the performance gap between the theoretical optimal [1] and LRU technique is large, several cache replacement techniques have been proposed for unicore LLCs [2, 3] to improve the cache efficiency.

In multicore processors, concurrent execution of applications demands significant memory bandwidth. This is provided by multi-level cache hierarchy with one or two levels of private caches and a shared LLC. As LLC can be shared by parallely running applications with varying access behavior, the replacement techniques proposed for unicore LLC may not be effective for multicore LLC. If such applications conflict with each other, system-wide performance can be significantly degraded. One application may dominate another by flooding cache requests and evicting the useful data of the other application. Performance of most of the cache replacement techniques proposed for multicore LLCs [4, 5, 6, 7, 8] depends on the data access patterns of specific workloads.

This work takes a different approach for replacement of a cache line from LLC by exploiting the non-uniform access rates and access behaviors of applications. It proposes an *Application-aware Cache Replacement* (ACR) policy. Apart from being access rate aware, ACR technique dynamically adapts the eviction process to the varying access patterns of applications. ACR technique is compared with LRU policy and state-of-the-art *thread-aware dynamic RRIP* (TA-DRRIP) [5] policy using SPEC CPU 2000 and 2006 benchmarks. ACR policy achieves (geometric mean) speed-up of 8.62% and 5.08% over LRU and TA-DRRIP policies, respectively, for 4-core workloads. The major contribution of the work is that the proposed replacement technique works well for workloads with both LRU friendly and scan access patterns as opposed to TA-DRRIP, which is not the best replacement technique for LRU friendly workloads.

2 Related Work

Several cache eviction policies have been proposed in the literature for both unicore and multicore systems. Discussion in this section is restricted to techniques that are relevant to proposed techniques.

Counter-based replacement technique [3] for unicore LLC predicts access interval using a counter for each cache line. All counters in a set are incremented on an access and a cache line whose counter value exceeds a given threshold is selected as victim.

The use of reuse information during victim selection for unicore LLC has been exploited in [2]. PC-based prediction method [2] predicts the reuse distance and uses the predicted values for cache eviction. On a cache miss, if the predicted reuse distance of the memory reference is higher than the reuse distance seen by all cache lines in the set, the requested data is directly sent to the processor without storing it in the cache. Otherwise, a cache line with highest reuse distance is replaced with the requested data.

To avoid keeping one time accessed cache lines for longer time, Bimodal Insertion Policy (BIP) [6] inserts most of the cache lines at the LRU position and place the others at MRU. But some applications are benefited if the cache lines are inserted at LRU position. In order to work with this varying behavior, dynamic insertion policy (DIP) [6] chooses either LRU or BIP policies at run-time. When it comes to multicore LLC, DIP technique is extended with thread-awareness [4], wherein each thread selects between LRU or BIP policies at run-time.

Static and dynamic cache replacement techniques based on re-reference interval prediction (RRIP) are proposed in [5]. Static RRIP (SRRIP) is scan-resistant, but not thrash-resistant. Thrashing is avoided by adopting an approach similar to BIP in bimodal RRIP (BRRIP). Both thrashing and non-thrashing access patterns are handled in dynamic RRIP (DRRIP), which selects between BRRIP or SRRIP for a given application using set-dueling monitors (SDMs)[6]. DRRIP policy does not have recency information. It inserts cache lines with low priority and changes it priority to highest only on a hit. During victim selection, it always searches from left and selects any cache line with lowest priority. If there are no suitable candidates, it keeps on changing the priority of all the cache lines till it finds a cache line with lowest priority. In case there are multiple cache lines with lowest priority, the search from left might not give the best victim candidate as the low priority of the chosen victim could be either due to its insertion or insertions of other cache lines. Hence, DRRIP policy does not always work well with LRU friendly applications. The work is extended to handle multi-programmed workloads in thread-aware dynamic RRIP (TA-DRRIP). With the help of two SDMs per application, TA-DRRIP dynamically selects either SRRIP or BRRIP in the presence of other application.

The promotion/insertion pseudo partitioning (PIPP) technique [8] has different priority positions for insertion of cache lines that belong to different applications. On a hit, accessed cache line is promoted by one position up in the priority chain. During promotion of cache lines, the applications with low priority position for insertion face stiff competition from those with high priority position for insertion. Hence, identifying suitable application-specific priority positions is critical for achieving good performance in PIPP technique.

Adaptive timekeeping replacement [7] uses the cache decay concept in cache line level for managing shared LLC. Operating system assigns three levels of priorities to the application and hardware assigns decay intervals accordingly. When a cache line is not accessed within the decay interval, it becomes a potential victim block. The main drawback with the technique is that it cannot distinguish between two or more applications having the same priority values.

Thrasher caging [9] identifies thrashing application that degrades the performance of multicore processor. The thrasher detection is based on the absolute number of misses from the cores. Once an application is detected as a thrasher application, reduced number of cache ways will be allocated.

Table 1. LLC statistics for SPEC CPU 2000 and 2006 benchmarks[1]

SPEC benchmark	LLC statistics		SPEC benchmark	LLC statistics	
	APKI	Miss Rate (%)		APKI	Miss Rate (%)
164.gzip	1.22	17.08	429.mcf	64.47	90.91
168.wupwise	3.01	99.13	435.gromacs	1.72	19.59
171.swim	22.89	99.98	437.leslie3d	9.15	82.64
172.mgrid	12.32	64.95	444.namd	0.68	98.68
173.applu	20.16	99.92	450.soplex	2.94	35.67
175.vpr	11.78	27.48	454.calculix	0.91	62.92
177.mesa	0.72	91.53	456.hmmer	2.14	71.36
178.galgel	14.09	43.91	458.sjeng	0.37	79.98
179.art	129.64	78.81	459.GemsFDTD	0.006	70.98
186.crafty	0.58	9.65	462.libquantum	6.72	99.64
193.fma3d	0.00051	100	464.h264ref	0.88	10.41
300.twolf	15.24	32.37	470.lbm	32.07	99.99
401.bzip2	5.18	43.57			

3 Motivation

In a multi-core scenario, multiple applications compete with each other for space in LLC. The access rates and behavior of these applications are different from one another and their accesses to LLC are filtered by caches closer to the processors.

Access Rates of Applications: Table 1 shows the *accesses per kilo instructions* (*APKI*) at LLC of different SPEC CPU 2000 and 2006 benchmark suites [10] with 3-level cache hierarchy in a single core environment. In a shared LLC with LRU replacement policy, high access rate application can dominate low access rate application. Figure 1 shows the cache line occupancy of a particular cache set for an application (*hmmer*) when it is concurrently executing with a lower access rate application (*calculix)* and a higher access rate application (*libquantum).* The average number of cache lines in the cache for *hmmer* reduces from 8.5 to 5.7 when the co-executing application is *libquantum* instead of *calculix.* This corresponds to a performance loss of 11.2% in IPC for *hmmer* due to *libquantum.*

Figure 2 gives a typical access in a 2-core system at time T1 (=3848587115 simulation cycle) with *hmmer-libquantum* during which *libquantum* flushes the application cache lines of *hmmer.* It shows that the number of cache lines in a particular set from *hmmer* (*libquantum*) is changed from 7 (8) to 2 (14) during an interval of 16 accesses to the set. This is because the LRU replacement policy is unaware of the difference in accesses across the applications. It selects the LRU candidate from an eviction chain that is common to both the applications. Due to the variation in access rate, the cache lines of low access rate application are pushed to the LRU position of priority chain and will be flushed out by the cache lines that belong to high access rate application. In such case, it is better to prevent an application from evicting a cache line of another application [11].

[1] Refer Section 5 for simulation setup.

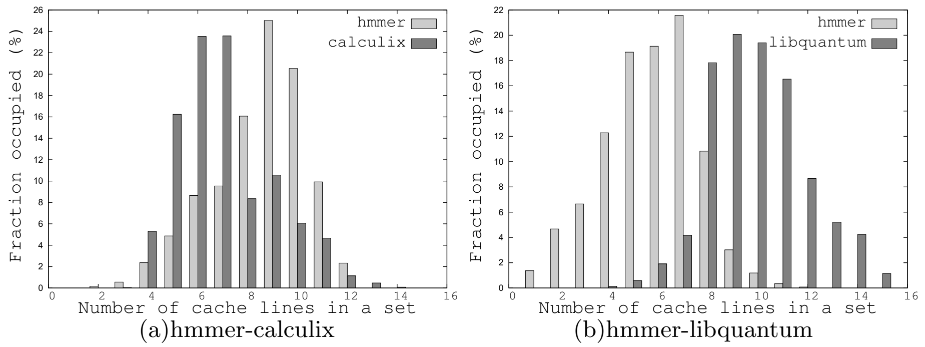


Fig. 1. The distribution of cache lines for set 56 in LLC when *hmmer* is executing with lower (*calculix*) and higher (*libquantum*) APKI applications

Hence, access rate aware eviction policy can improve the overall performance of the system.

Access Pattern of Applications: Access rate of an application does not give any insight on the temporal locality of the application. Relying just on application-wise access rates for cache replacement may sometimes degrade the performance of the system. As can be seen from Table 1, benchmarks such as *mesa* and *n*amd have low APKI and very high miss rates. When such an application is co-scheduled with high access rate application, the access count based strategies can penalize the latter. This is of disadvantage because high access rate application is penalized without considering the temporal locality of individual applications. Hence, the replacement policy should also be aware of the access recency behavior of the overall system along with access rate.

Each application has different reuse patterns, which can change during various stages of the execution. Figure 3 shows the hit-behavior of SPEC CPU 2000 and 2006 benchmark suites in a single core environment with 1MB 16-way set associative LLC. The hit behavior is measured in terms of *hit-gap* of the cache lines. The *hit-hap* of a cache line is defined as the number of accesses to the corresponding set between consecutive accesses to the cache line. The graph provides percentage of hits covered for different *hit-gap* values. Most of the applications

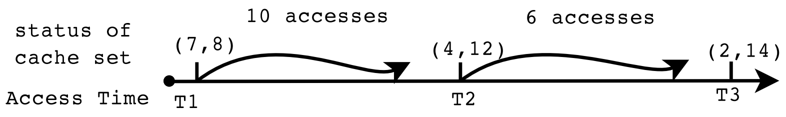


Fig. 2. Access sequence for *hmmer-libquantum* for set-56. The cache occupancy status is denoted by (a,b), where a is number of cache lines that belong to *hmmer* and b is number of cache lines that belong to *libquantum*.

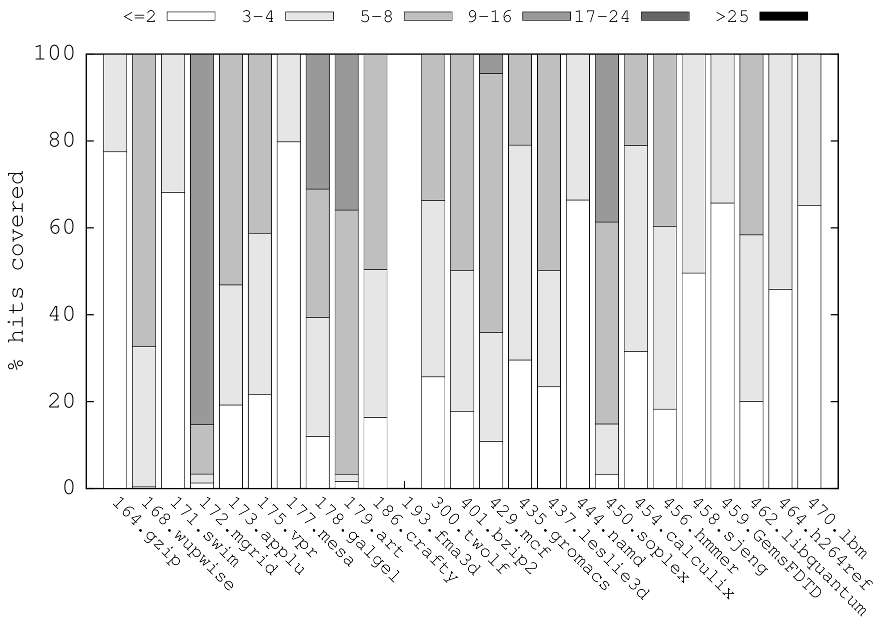


Fig. 3. *hit-gap* for single core SPEC benchmarks

cover their hits by a *hit-gap* of 16 and *fma3d* does not have hits at all. Even when an application gets the entire share of LLC, the *hit-gap* is not increasing beyond a particular value. For example, applications such as *gzip*, *swim*, *mesa*, *namd*, *sjeng*, *GemsFDTD*, *h264ref* and *lbm* cover most of their hits by a *hit-gap* of 4. If a cache line of one of these applications is present in the cache for more than 4 accesses to the set, it is highly likely that the cache line will not be referenced again. This maximum value of *hit-gap* will give the maximum life-time required by any cache line of the application in the cache. The maximum *hit-gap* when tracked dynamically can be used during cache line eviction. The use of maximum *hit-gap* will ensure that all cache lines of an application are present in the cache for only allotted time. Hence, the use of maximum *hit-gap* information can facilitate better utilization of available cache space.

In conclusion, variation in access counts, access recency, and *hit-gaps* across different applications motivates an alternate cache replacement policy. The next section proposes such policy.

4 ACR: An Application-Aware Cache Replacement Policy

The paper proposes an application aware cache replacement (ACR) policy for shared LLCs. To make the policy access rate aware, separate local eviction priority chains are maintained for different cores. The length of each chain is dynamically changed at run-time to make *ACR* policy access pattern aware.

ACR technique updates the eviction priority of only those cache lines that belong to the referencing application and maintains separate eviction priority chains for each application. It is seen in Section 3 that for low-access and low-hit rate applications, separate replacement chains that are aware of the access rates of applications is not sufficent and can sometime degrade the performance. Keeping a tab on the order of access recency among concurrent applications for individual sets along with access counts is of use here. Further in case of victim selection if multiple applications have victims with same eviction priority, the access recency information can be used to guarantee performance similar to LRU policy.

The study of *hit-gaps* of cache lines for different applications in Section 3 also shows that each application has different maximum residency period for its cache lines. This maximum residency period for each application can be used to limit the length of individual eviction priority chain as the cache lines are unlikely to be reference after this period. ACR policy changes the length of the individual eviction priority chain based on the application-wise maximum *hit-gap*. ACR policy dynamically tracks the maximum *hit-gaps* of an application for each interval and uses it as the predicted life-time or *predicted-hit-gap* (PHG) for the next interval. We consider the maximum hit-gap observed for the prediction to avoid any additional miss penalties due to insufficient prediction of cache life-time. Special care is taken in the absence of hit for an application in an interval, as the reason for no hits could be an error in the *predicted hit-gap*.

Implementation: ACR technique uses the following registers/counters for implementation:

- n-bit saturating counter called *hit-gap* counter (HG) for each cache line to keep track of individual access counts.
- $N*logN$ bits per cache set to maintain application-wise access recency order (LRU chain), where N is the number of cores in the system.
- Two n-bit application-wise counters, *predicted-hit-gap* (PHG) and *shadow predicted-hit-gap* ($sPHG$). PHG value is the predicted life time for current interval and $sPHG$ value is the learned maximum *hit-gap* during the current interval (to be used as the predicted life-time for the next interval).
- 1-bit $hitFlag$ per application, which is set on a hit for the application.

Access Rate Awareness: On every access to a set, HG counters of all cache lines in the set that belong to the accessed application are incremented. The counter value of a cache line at any given time is an estimate of its life-time in the cache after its last access. This time is measured in terms of the number of accesses to the cache set. A cache line with the highest counter value is the oldest cache line without access in the set. If an access is a hit, the counter of the corresponding cache line is reset. The value of the counter at the time of hit is the *hit-gap* of the cache line and so is called the *hit-gap* (HG) counter. Whenever a new cache line is inserted, its HG value is set to 0. Each access to the cache also updates the application wise LRU chain corresponding to that set.

Access Pattern Awareness: For dynamically tracking the application life-time (maximum *hit-gap*) the total number of cache sets s is divided into p *monitor*

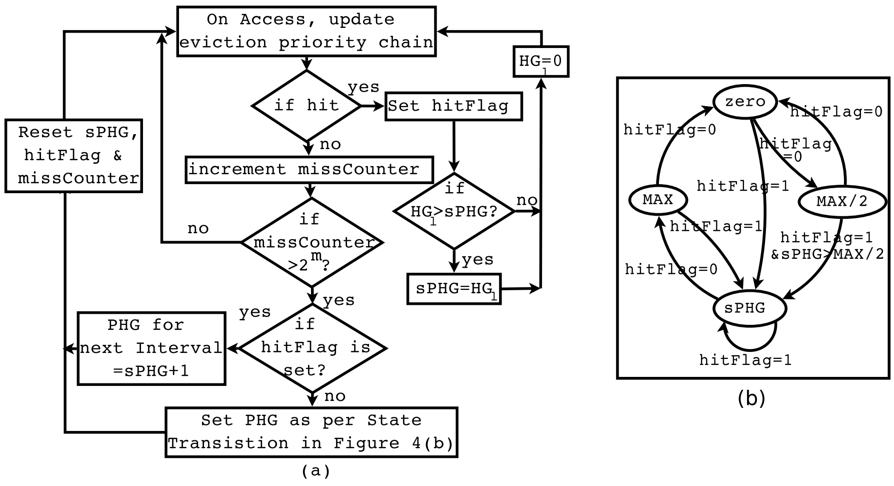


Fig. 4. Illustration of ACR policy (a) Flowchart illustrating dynamic update of PHG. (b) Computation of PHG at the end of an interval.

sets and $(s - p)$ *normal sets*. The predicted life-time for all monitor sets is fixed at $MAX = 2^n - 1$ to ensure the maximum possibility of *hit-gap* for cache lines in monitor sets. The hit gaps of the monitor and normal sets are tracked and stored as $sPHG$ for each interval of 2^m misses (implemented using an m-bit *missCounter*) for LLC and the PHG value is updated at the end of each interval (Figure 4(a)). At the start of each interval $sPHG$ and $hitFlag$ are reset. During each hit to a cache line l of an application i, if $HG_l > sPHG_i$, then $sPHG_i$ is set to HG_l. At the end of the interval, $sPHG_i$ stores the maximum *hit-gap* of the application. If $hitFlag_i$ is set, PHG_i for the next interval is set to $sPHG_i + 1$. If the $hitFlag_i$ is reset, the application had no hits during the interval. This could be because of insufficient PHG, which is tackled using the FSM shown in Figure 4(b). In case of no hit, the FSM gives the application maximum time in cache for the next interval, i.e., $PHG = MAX$. If the status of the $hitFlag$ continuous to remain reset, it means the application has thrashing behavior, and so $sPHG$ is reset. To make sure such application is given opportunity to remain in the cache if its behavior changes, PHG alternates between 0 and $\frac{MAX}{2}$. Note that if at any point the application encounters a hit, the value of $sPHG$ is used for PHG for the next interval.

Victim Selection: As maximum time in the cache for a cache line l of an application i is when $HG_l = PHG_i$, the cache line with minimum predicted life-time will be one with $\min(PHG_i - HG_l)$. Hence on a miss, the victim in the absence of invalid cache line is the cache line with $\min(PHG_i - HG_l)$. The search for victim cache line starts from LRU core so that the cache line that has minimum life in the cache from a core that is least recently used is evicted.

Note that in the absence of *monitor sets* in the PHG computation, the chances of the value of $sPHG$ being larger than PHG would be very less as all cache

Table 2. Architectural parameters of the simulated system

IL1 caches	32KB, 64B, 4-way, 1 cycles, 1W and 2R ports, LRU
DL1 caches	32KB, 64B, 4-way, 1 cycles, 1W and 2R ports, LRU
L2 cache	256KB, 64B, 8-way, 10 cycles, private, LRU
LLC	1MB per-core, 64B, 16-way, 35 cycle, shared, non-inclusive
Main memory	200 cycles

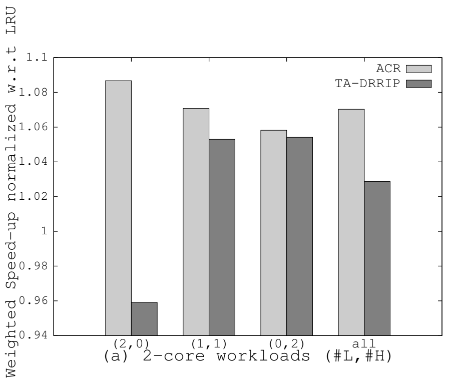
lines with $HG = PHG$ are evicted. Any hit in the monitor or normal cache set can change the value of $sPHG$ as long as the *hit-gap* of the current hit is greater than the present value of $sPHG$. Also, the victim selection procedure is not in the critical path and hence the search involved in the algorithm does not affect the system performance.

Whenever a cache encounters scan access pattern, the corresponding cache lines become victim candidates faster by virtue of application based HG modification. Such scan patterns will not have hits due to which cache lines of such applications will be predicted to have smaller life-time in the cache. Hence, ACR policy is scan resistant irrespective of the length of the scan chain unlike SRRIP. ACR policy is aware of access recency and so can perform well for LRU friendly applications. Hence in contrast to TA-DRRIP, ACR policy can perform well for applications that have access patterns that can be LRU friendly or scan.

5 Evaluation

Our technique is evaluated with 3-level cache hierarchy having private L1 and L2 caches with shared LLC. Table 2 gives other details of the system configuration used in experimentation. Workloads using set of 26 SPEC CPU 2000 and 2006 benchmark suites [10] (refer to Table 1), compiled for ALPHA ISA, are executed on GEM5 simulator [12]. All the benchmarks are executed using reference inputs. Applications are fast-forwarded for 900 million instructions and then warmed up for next 100 million instructions. The statistics are recorded for the next 1 billion instructions for each application. As ACR technique is access rate aware, we consider workloads with mix of low (L) and high (H) APKI values. Applications with $APKI > 6$ are categorized as H and others are categorized as L. The workload mixes are categorized based on the number of low and high APKI applications. For 2- and 4-core systems, we have 3 $((\#L, \#H) = \{(2,0),(1,1),(0,2)\})$ and 5 $((\#L, \#H) = \{(4,0),(3,1),(2,2),(1,3),(0,4)\})$ categories of workloads, respectively. About 20 mixes are considered for each class of workloads in both 2- and 4-core systems.

Results and Analysis: ACR policy is compared with LRU and TA-DRRIP [5] techniques. We consider 4-bit HG counter, 12-bit *missCounter* and 128 monitor sets for ACR policy. TA-DRRIP [5] is implemented with $2N$ SDMs, where N is the number of cores, with 32 sets each to learn the insertion decision of each



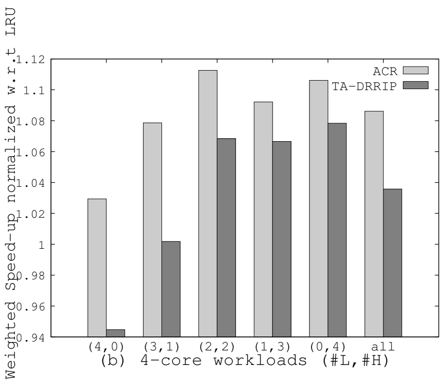


Fig. 5. Performance comparison of ACR policy with LRU and TA-DRRIP policies

application. A 10-bit dedicated PSEL counter is used to decide the core-wise insertion policy.

Effect on System Performance: Performance of multiple applications that execute concurrently is evaluated using weighted speedup. Weighted speedup gives the improvement in execution time compared to the baseline configuration. Figure 5(a) shows the performance improvement of ACR policy for 2-core system compared to LRU and TA-DRRIP policies in various categories of workloads. ACR policy achieves speed-up of 7.02% and 4.16% (geometric mean) with respect to LRU and TA-DRRIP policies, respectively.

Similar behavior for 4-core systems can be observed in Figure 5(b). The geometric mean of performance improvement of ACR policy as compared to LRU and TA-DRRIP techniques are 8.6% and 5.03%, respectively. ACR policy outperforms LRU and TA-DRRIP policies in all categories of workloads for both 2- and 4-core systems. The difference in performance improvement between ACR and TA-DRRIP policies is largest for (2,0) and (4,0) workload categories in 2- and 4-core systems, respectively. This difference reduces as the number of H category applications increases in the workloads. Most of the applications in L category have low miss rate or good temporal locality (refer to Table 1) with LRU policy and hence are LRU friendly. ACR policy improves the performance of workloads with LRU friendly patterns as the modification of eviction priority chain for each application is similar to LRU chain update. On the other hand, TA-DRRIP policy does not have recency information and hence degrades the performance of workloads with similar access patterns. Further, due to the dynamic update of length of the eviction priority chain based on *hit-gap* in ACR policy, applications with scan patterns will have shorter life-time in the cache. Hence ACR policy is able to improve performance of workloads with both scan and LRU friendly patterns. With increase in high access rate applications in the workloads, both ACR and TA-DRRIP policies have improved performance but the performance of ACR policy is still higher than that of TA-DRRIP technique as it is access rate aware along with being access pattern aware.

Figure 6 shows the speedup improvement of ACR and TA-DRRIP policies in a 4-core system with respect to LRU technique for all the workloads. It can be observed that unlike TA-DRRIP, the performance of ACR is always better than LRU policy.

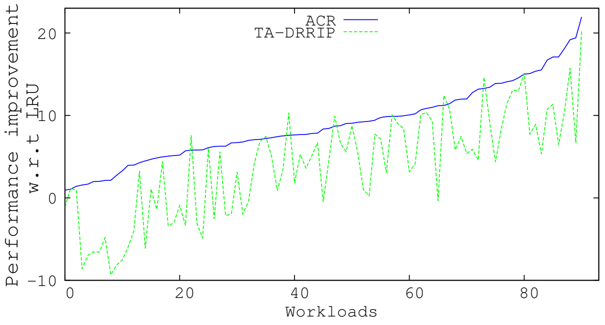


Fig. 6. Effectiveness of ACR and TA-DRRIP policies for all workloads in 4-core system.

Table 3. Overhead of replacement policies for 4-core system with 4-MB LLC

Replacem-ent Policy	Overhead
LRU	$32KB$
TA - DRRIP	$\approx$ 16KB
ACR	$\approx 36KB$

Victim selection procedure for our technique involves comparing the counter values to identify the victim candidate. This procedure is similar to that of LRU. Since the victim selection is in parallel to memory access, it does not increase the critical path to the processor. For a 'k-way' set associative cache, LRU and TA-DRRIP replacement policies have an overhead of $k * logk$ and $2 * k$ bits per cache set, respectively. In addition, TA-DRRIP has 10-bit PSEL counter per core to support SDMs. ACR policy has an overhead of $k * logk$ (for HG counters) + $NlogN$ (to implement core-wise LRU), where N is the number of cores in the system. In addition to this, ACR policy has two 4-bit register for PHG and $sPHG$, and one 1-bit register for $hitFloag$ per application and a single 12-bit $misCounter$. Table 3 shows the overhead involved in implementation of LRU, TA-DRRIP, and ACR replacement policies for 4-core system with 4MB LLC. Even though ACR policy incurs slightly larger area overhead than TA-DRRIP, ACR achieves significant performance improvement in both 2- and 4-core systems. Hence it can be used in environments where performance is critical and hardware overhead is not a constraint.

Effect of the size of HG counter: Performance of 2- and 4-core systems is evaluated for different sizes of the HG counter. Performance of ACR policy improves with the increase in the number of bits in the HG counter from 3 bits to 5 bits for both 2- and 4-core systems. Increase in the number of bits for HG counter provides better control on the length of eviction priority chains. This gain is almost constant on increasing the size of HG counter beyond 5 bits. Thus, we consider 4-bit HG counter as it provides significant performance improvement without incurring much hardware overhead.

6 Conclusion

A cache eviction policy for multicore shared LLC is proposed to exploit application wise access rate and pattern. Evaluation of ACR technique using SPEC CPU 2000 and 2006 benchmark suites has shown to improve the performance with respect to LRU and TA-RRIP techniques. Experiments on 2- and 4-core systems indicate that incorporating awareness of access-rates and *hit-gaps* during cache eviction will improve the LLC utilization. As the proposed policy is aware of the access rates of the applications, it prevents domination of high access rate application over low access rate application. It also performs well in the presence of both LRU friendly and scan access patterns. As part of future work, we plan to i) look at the challenges of using this technique in a multi-threaded scenario; ii) apply fine grain control on the life of individual cache lines of applications at run-time along with the coarse grain maximum life constraints exerted by ACR policy.

Acknowledgement. This work was supported in part by grant from Department of Science and Technology, India, Project No. SR/S3/EECE/0018/2009.

References

[1] Belady, L.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5(2), 78–101 (1966)
[2] Keramidas, G., Petoumenos, P., Kaxiras, S.: Cache replacement based on reuse distance prediction. In: International Conference on Computer Design (2007)
[3] Kharbutli, M., Solihin, Y.: Counter-based cache replacement and bypassing algorithms. IEEE Transactions on Computers 57 (2008)
[4] Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely Jr., S.C., Emer, J.: Adaptive insertion policies for managing shared caches. In: ACM International Conference on Parallel Architectures and Compilation Techniques, pp. 208–219 (2008)
[5] Jaleel, A., Theobald, K.B., Steely Jr., S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). In: ACM International Symposium on Computer Architecture, pp. 60–71 (2010)
[6] Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely Jr., S.C., Emer, J.: Adaptive insertion policies for high-performance caching. In: ACM International Symposium on Computer Architecture, pp. 381–391 (2007)
[7] Wu, C.J., Martonosi, M.: Adaptive timekeeping replacement: Fine-grained capacity management for shared cmp caches. ACM Transactions on Architecture and Code Optimization 11, 27 (2011)
[8] Xie, Y., Loh, G.H.: PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In: ACM International Symposium on Computer Architecture, pp. 174–183 (2009)
[9] Xie, Y., Loh, G.H.: Scalable Shared-Cache Management by Containing Thrashing Workloads. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 262–276. Springer, Heidelberg (2010)
[10] SPEC CPU benchmark suite, `http://www.spec.org`

[11] Srikantaiah, S., Kandemir, M.: Irwin: Adaptive set pinning: Managing shared caches in chip multiprocessors. In: International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 135–144 (2008)
[12] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The GEM5 simulator. ACM SIGARCH Computer Architecture News 39, 1–7 (2011)