Australian Journal of Basic and Applied Sciences, 9(11) May 2015, Pages: 804-810

ISSN:1991-8178

Australian Journal of Basic and Applied Sciences

Journal home page: www.ajbasweb.com

Adaptive Replacement and Insertion Policy for Last Level Cache

[1]Muthukumar S. and [2]HariHaran S.

[1]*Professor, Department of Electronics and Communication Engineering, Sri Venkateswara College of Engineering, Sriperumbudur, India,*
[2]*PG Scholar, Department of Computer Science and Engineering, Sri Venkateswara College of Engineering, Sriperumbudur, India*

ARTICLE INFO

Article history:
Received 25 March 2015
Accepted 28 May 2015
Available online 14 June 2015

Keywords:
LRU, LLC, SIP

ABSTRACT

Processor technology has witnessed advances in recent years. The cache memory is used to bridge the speed gap between the processor and the DRAM. The LRU based shared last level cache (LLC) is currently employed in chip multiprocessors. The LRU insertion and promotion policies assume only one type of application with a particular memory access behavior while dictating the replacement policy for the cache. In this paper, we propose a method that considers different types of applications with different memory access behaviors while dictating the replacement policy for the cache. In this method, as the application arrives the cache access behavior of the application is monitored along with the other applications that are already being executed. Few sample sets are selected from the cache and the LRU insertion/promotion policy is applied on those sample sets. An identical number of different sample sets is selected from the cache and BI-model insertion and SIP promotion policy is applied on those sets. The insertion/promotion policy exhibiting better performance is chosen by using a reference bit and that insertion/promotion policy is applied to all the remaining sets of the cache called follower sets. This proposed technique is found to perform well when compared to LRU for most of the workloads in PARSEC bench mark suite.

© 2015 AENSI Publisher All rights reserved.

To Cite This Article: . Muthukumar S. and HariHaran S., Adaptive Replacement and Insertion Policy for Last Level Cache. ***Aust. J. Basic & Appl. Sci.,*** *9(11): 804-810, 2015*

INTRODUCTION

LRU replacement policy is prominently used as the replacement method in shared last level cache. The performance of LRU is severely degraded mainly because of two reasons. They are 1) When two or more applications with totally different cache access behavior run simultaneously and 2) The application's working set size is more than that of the cache size.

The existing approaches tackling this performance degradation problem do not bring any notable improvement in the cache performance because they only focus on a single type of memory access behavior. They do not consider tradeoffs between the different types of memory access behavior. Because of the above reasons the performance of the cache is degraded. This observation forms the background of this algorithm. Figure 1 shows the memory hierarchy in multi-core processor. The proposed algorithm is implemented in the shared last level cache of a multi-core processor.

Related Works:

In (Wu, Junmin, 2010) a unified cache management policy called partition aware eviction and thread aware insertion/promotion policy is proposed. This method enhances capacity management, adaptive insertion/promotion policy and it also improves the overall cache performance. It employs an adaptive mechanism to decide the position to insert the incoming lines or to promote the hit lines, and chooses a victim line based on the target partitioning given by utility-based cache partitioning (UCP).

(Qureshi, K., *et al*., 2006) proposes a policy in which the cache resources will be assigned based on performance. The proposed UCP algorithm monitors the application's behavior and cache resources will be allocated to applications which will benefit more from the cache(i.e based on reduction in cache misses). The applications are monitored during runtime using a hardware circuit with less than 2kb of storage. The information thus collected is used by the algorithm to decide the amount of cache resources allocated to each application.

(Muthukumar, S., P. Jawahar, 2014)This paper takes into account the number of threads accessing a particular data. The data that is accessed by more number of threads is given more priority to stay in cache as the miss in these data will lead to stalling of many threads. The proposed method accounts for the sharing status of data in the cache. The extent to

Corresponding Author: Muthukumar S., Professor, Department of Electronics and Communication Engineering, Sri Venkateswara College of Engineering, Sriperumbudur, India, E-mail: smuthukumar.65@gmail.com

which every element is shared is indicated by a two bit counter called sharing degree counter that is associated with each element.

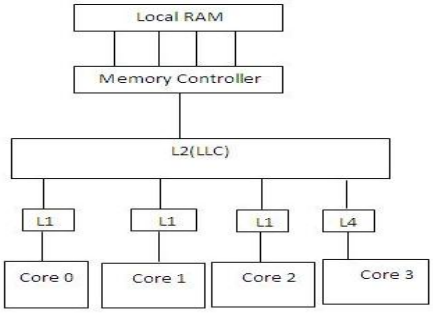

Fig. 1: Multi Core Processors.

This counter takes four values corresponding to four states- not shared, lightly shared, heavily shared and very heavily shared. A dynamic array called thread track filter is associated with every cache block is used to keep track of the threads that access the block. The priority value for a cache block is determined by using both hit count and sharing degree. Eviction takes place based on increasing priority values. If there is a tie, the first encountered cache block is taken as the victim. In addition, the priority value of all cache blocks is decremented by one after an eviction to ensure the cache is not polluted by stale data.

In (Muthukumar, S., P.K. Jawahar, 2013) the parallel multi-threaded application is given prime focus. A method called context based exploitation technique is employed. A cache counter is assigned to every block of the l2 cache. This counter is updated based on the data access pattern to maximize the percentage of hits.

In (Muthukumar, S., P.K. Jawahar, 2014) the cache elements are classified into four logical partitions or zones based on likelihood of the element being referenced by the processor in the immediate future. A three bit counter is associated with each cache line. This counter is used for classification of elements into zones. Elements are promoted between zones in ascending order of the counter value. Periodic demotion is also done to prevent stale data from polluting the cache.

The performance of the multithreaded applications is sensitive to the jobs that are co-scheduled with them. The SMT hardware can take full advantage, if the scheduler is aware of thread interactions. A job scheduler called SOS combines an overhead free sample phase. It collects all the possible schedules and uses the information to predict which schedule will provide the best performance (Snavely, 2000).

(Chaudhuri, 2012) The Last-Level Cache (LLC) policies lack information about the inner-levels activity of the hierarchy which result in sub optimal performance. To eliminate this, cache hierarchy-aware replacement (CHAR) algorithms for inclusive LLCs (or L3 caches) are introduced.

These algorithms have been used to implement efficient bypass techniques for exclusive LLCs in a three-level hierarchy. Thus, an effective algorithm for selecting the subset of blocks (clean or dirty) removed from the L2 cache can be bypassed.

This paper proposes a replacement algorithm for the last level cache. A victim is selected based on some priority values. The algorithm seeks to ensure that a line that is evicted is not immediately needed by another core by taking shared nature into consideration. The blocks from which victims are chosen are divided into four groups- shared but clean, private and exclusive, private and modified, other shared lines.

The cache lines having highest reuse register value is first determined for the purpose of eviction. A block that is continually accessed is given a higher reuse register value. The value is high, if the block is private. Otherwise it decreases to a value close to zero. In terms of eviction, lowest priority is assigned to shared lines as their eviction is assumed to be more costly. This logic is also applicable for private modified lines which are however given a higher priority as they are present only in private caches of a single core.

Highest priority is accorded to the clean lines which are evicted first. However, in some sets the highest priority is assigned to private exclusive lines. The first time any block is accessed, it is in private mode. The second access will determine whether it remains in private mode or shared mode based on whether or not the accessing thread is the same as the first. Eviction is performed based on calculations involving reuse-register value and the priority. The block with the highest value is evicted first. Upon simulation, the WARP algorithm showed a better performance than LRU (Balaji, S.,).

The major problem in cache memory is contention. This issue of contention can be solved by identifying high priority threads. A technique called adaptive time keeping replacement proposed in (WU, C., M. Martonosi, 2011) implements such a method. It assigns a life time to cache lines referring to the OS assigned process priority and application memory reference pattern. A decay counter is assigned to each cache line based on process priority and decremented over time. Whenever there is a hit, the cache line will be reset to decay interval, so when the cache hits zero the corresponding cache line will be considered for replacement. The decay counter for the higher priority process will receive higher decay interval. This ensures that the higher priority process will stay in the cache for longer time.

Background Study:

Whenever the processor needs a word it searches for the word in the cache memory. If the particular word is found in the cache memory then the scenario

is termed CACHE HIT. If the particular word is not found in the cache memory then it is termed as CACHE MISS. Whenever there is going to be a cache miss, the data is searched and obtained from the main memory and placed in the cache memory. The cache memory is smaller and so it can hold only fewer amount of data. So it is ideal to have such type of data in the cache so that cache misses are reduced as much as possible. If there is going to be a cache miss, cache will go for replacement policy and if there is going to be a cache hit, cache will go for promotion policies. The replacement policies are dictated by principles of locality(temporal and spatial locality).

The principle of locality works with an assumption that the access of data is going to be sequential as in single core single thread system. But for multithreading applications, the access of data is random. So it becomes imperative to choose the replacement policy dynamically.

The effective cache replacement and insertion can be achieved by discovering the pattern in which the threads accesses the data or by devising effective methods for keeping up with random accesses.

A. Workload Characteristics:

Based on utility, the applications in cache can be classified into four major groups like

High utility applications:

These are the applications that benefit from caches.

Saturation utility applications:

These are the applications that require certain amount of cache space to perform well.

Low utility applications:

Those applications whose working set is larger than that of the cache size can be classified as low utility.

Very low utility applications:

Those applications that have extremely large working set and very little cache reuse can be classified among very low utility.

In low utility and very low utility applications, the access behavior is such that the reuse of the cache block is at a minimum. It means that these applications produce large number of dead blocks(blocks that are seldom reused). It is ideal to remove those dead blocks from the cache as early as possible but the widely used LRU algorithm does not eliminate those blocks early. In the proposed algorithm the access behavior of the applications that are currently under execution are monitored and an alternate replacement and insertion policy are applied when the application is found out to be low utility or very low utility type applications.

Cache Insertion Policies:

LRU insertion:

Whenever a cache miss occurs and a new block is fetched that block will be placed at the beginning (MRU position) of the cache.

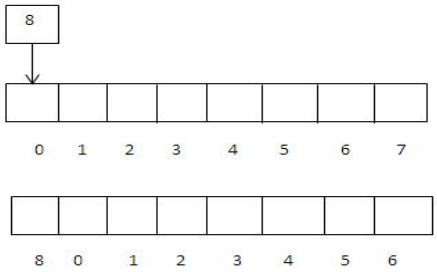

Fig. 2:LRU insertion policy.

Bimodal Insertion Policy (BIP):

Whenever a cache miss occurs and a new block is fetched that block will be placed at the end (LRU position) of the cache.

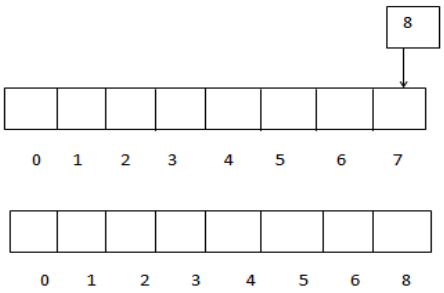

Fig. 3: Bimodal insertion policy.

Cache Promotion Policies:

LRU insertion promotion (LIP):When a cache hit occurs the block that gets hit is promoted to the MRU position of the cache block.

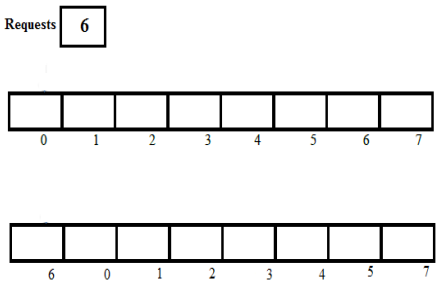

Fig. 4: LRU insertion and promotion.

SINGLE insertion promotion(SIP):

When a cache hit occurs the block that gets hit is promoted by a single position in the cache block.

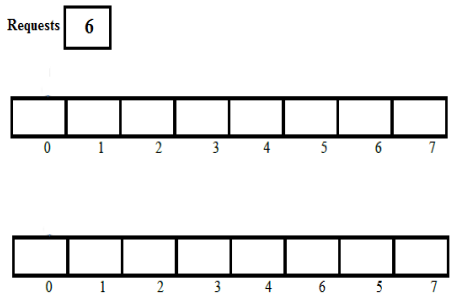

Fig. 5: Single insertion and promotion.

Proposed Mechanism:

The proposed mechanism uses an 8-way set associative cache. Two sets of sample sets are chosen from among the total sets. In the first set of samples, LRU insertion and promotion policies are applied and in the second set of samples, bimodal insertion policy and single insertion promotion policies are applied.

A four bit saturation counter is used to monitor the cache sets. The MSB bit of the saturation counter is used to determine which algorithm is better for the applications that are currently in execution. Initially the counter is set to 1000. In the proposed method both the sample sets are monitored. Whenever there is going to be a cache miss in the first set of samples, the saturation counter is incremented by 1 and when there is going to be a cache miss in the second set of samples, the saturation counter is decremented by 1.

As a new application comes into execution the above process is carried out. If the MSB bit of the saturation counter is found out to be 0 then LRU insertion and promotion policies are applied to the follower set. If the MSB is found to be 1 then bimodal insertion and single insertion promotion policies are applied.

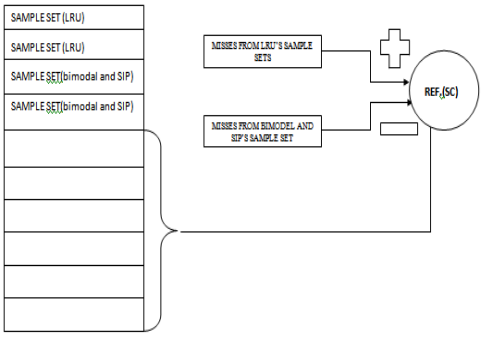

Fig. 6: Proposed architecture.

Algorithm:

```
Algorithm: imp(set,blk,numset)
{
//numset->total number of sets
Ref_counter=0
If blk==NULL then
{
Set=extractset(addr)
If set>numset/8 then
{
Decrease ref_counter by 1
If ref_counter<=1then
Initialise Ref_counter to 0
}}
Else if set>numset/8 or set>numset/4 then
Increment Ref_counter by 1
If ref_counter>=16 then
Ref_counter=16
Else if blk!=NULL then
{
If blk->set<numset/8 then
{
Sets[blk->set].promote(blk)
Is_secure?"s":"ns"
}
Else if blk->set>numset/8 and blk->set>numset/4
then
{
sets[blk->set].moveToHead
 blk->set, regenerateBlkAddr(blk->tag, blk->set),
is_secure
 ? "s" : "ns");
}
Else
{
If LRU::ref_counter >= 8 then
{
sets[blk->set].promote(blk)
blk->set, regenerateBlkAddr(blk->tag, blk->set),
is_secure ? "s" : "ns");
}
else if  LRU::ref_counter < 8 then
{
sets[blk->set].moveToHead(blk)
blk->set, regenerateBlkAddr(blk->tag, blk->set),
is_secure ? "s" : "ns");
}} }
return blk;
}
```

Simulators and Benchmarks:

The efficiency of the proposed algorithm is compared with LRU algorithm by using some standard inputs called PARSEC benchmarks. The simulator we use to test our algorithm's performance is the gem5 simulator which m5sim.org defines as a modular discrete event driven computer system simulator platform. Gem5 was designed in python and C++ and most of its constituents are delivered under a license along the lines of BSD. The gem5 can be used to simulate benchmarks like PARSEC, SPLACH and SPEC benchmarks (Main Page, 2015; Introduction, 2015).

According to http://parsec.cs.princeton.edu/, the Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite which consists of multithreaded applications (PARSEC, 2015; The Parsec Benchmark Suite, 2015). It has a

collection of programs that are written in C++ and consists of wide range of multi threaded applications.

Result and Analysis:

For applications in which the cache reuse is very high, the existing LRU algorithm will be suitable. Consider an application whose cache reuse is minimum(e.g. video frames). In such a case, the block that gets inserted in the cache will never be reused and hence becomes a dead block(occupying cache but will never be reused). In the already existing LRU policy, the block will be inserted in the MRU position which means the dead block will take many cycles to get out of the cache where as in bimodal policy it takes minimum number of cycles to eliminate the unused dead blocks which leads to better performances for such applications. The proposed algorithm, which dynamically decides the policy, combines the goodness of both LRU insertion and promotion and bimodal insertion and promotion policies.

The proposed algorithm was implemented in the GEM5 simulator which simulates the multi-core architecture environment. The performance of the algorithm was evaluated by using PARSEC benchmark and the proposed algorithm was found to have shown better performance for significant parameters like cache hits, cache misses, cache miss rate and average reference to the valid block. The results obtained for both proposed method and LRU are plotted.

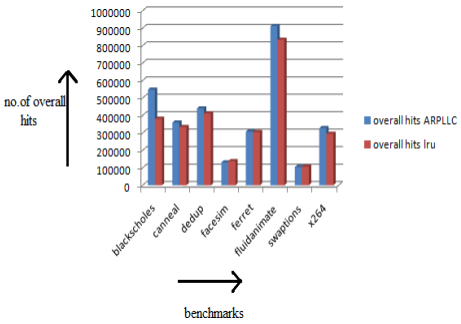

Fig. 6: Comparison of overall hits in L2 cache.

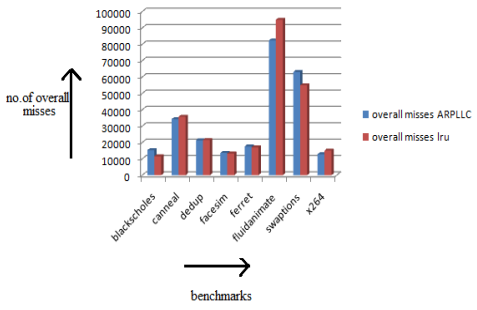

Fig.7: Comparison of overall misses.

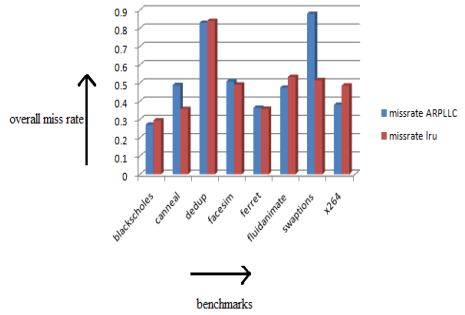

Fig. 8: comparison of overall missrate.

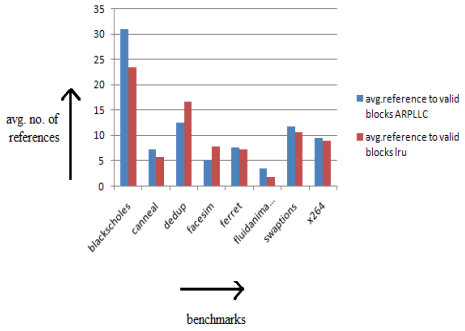

Fig. 9: Comparison of average reference to valid blocks.

Table 1: Total no. of hits in l2 cache.

BENCH MARKS	ARPLLC	LRU	PERCENTAGE INCREASE
blackscholes	547931	381541	43.6
canneal	3599076	3331566	8.02
dedup	440162	411186	7.04
facesim	131736447	138265393	-4.7
ferret	307482	305419	0.6
fluidanimate	912149	833898	9.38
swaptions	1047521	1061613	-1.32
X264	32734687	29555008	10.75

Discussion:

It is vital to improve the performance of cache memory if the overall system performance has to be improved. Advent of multiprocessor technology has given rise to parallel programming for which the existing LRU algorithm is not particularly efficient. Hence, in this paper we endeavored to eliminate the short comings of the LRU algorithm and increase the efficiency of cache replacement for majority of benchmarks.

The following is a tabulated representation of the results obtained.

Table 2: Total number of overall misses.

BENCH MARKS	ARPLLC	LRU	PERCENTAGE INCREASE
blackscholes	15304	11612	31.7
canneal	3429094	3577561	-4.15
dedup	2137138	2152965	-0.7
facesim	136298360	133108911	2.39
ferret	175706	170467	3
fluidanimate	823848	950373	-13.31
swaptions	6314	5499	14.8
X264	1295982	1508058	-14.06

Table 3: Miss rate of l2 cache.

BENCH MARKS	ARPLLC	LRU	PERCENTAGE INCREASE
blackscholes	0.027172	0.029536	-8.0
canneal	0.487907	0.517802	-5.77
dedup	0.829216	0.839640	-1.2
facesim	0.508510	0.490499	3.6
ferret	0.363639	0.358210	1.5
fluidanimate	0.474568	0.532639	-10.9
swaptions	0.008786	0.005153	70.6
X264	0.038063	0.048548	-21.6

Table 4: No. of references to valid block.

BENCH MARKS	ARPLLC	LRU	PERCENTAGE INCREASE
blackscholes	30.9344	23.3336	32.57
canneal	7.1311	5.5615	28.22
dedup	12.4959	16.5825	-24.64
facesim	5.0869	7.6541	-33.54
ferret	7.5607	7.1413	5.88
fluidanimate	3.3332	1.7213	93.64
swaptions	11.6568	10.6113	9.85
X264	9.4272	8.8889	6.05

Future Work:

An improvement in performance obtained is of great value because even a very small improvement can make a huge difference in real time. In future we expect to develop a modified version of our algorithm in which each applications behavior can be monitored separately and suitable algorithm can be applied for each application.

Conclusion:

The efficiency of the shared last level cache memory determines the overall performance of the chip multiprocessors and the replacement policy used in it is important due to its impact on performance. LRU's performance is poor for all parallel and concurrent workloads and in this paper a new algorithm is proposed which has improved the performance of cache for majority of workloads in PARSEC benchmark. The significant parameters like total number of hits in L2 cache, the average miss rate and number of references to valid blocks were found to yield an average improvement of 11.65%, 4% and 14.8% respectively for the tested benchmarks.

REFERENCES

Hennessy, L., John and A. David, 2007. Patterson. Computer Architecture a Quantitative Approach. 4th ed. Amsterdam: Elsevier/Morgan Kaufmann. Print.

"Cache Algorithms, 2015." Wikipedia.Accessed. http://en.wikipedia.org/wiki/Cache_algorithms.

Wu, Junmin, Xiufeng Sui, Yixuan Tang, Xiaodong Zhu, Jing Wang and Guoliang Chen, 2010. "Cache management with partitioning-aware eviction and thread-aware insertion/promotion policy."In Parallel and Distributed Processing with Applications (ISPA), International Symposium on, pp: 374-381.IEEE.

Qureshi, K., Moinuddin and N. Yale, 2006. Patt. "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches." In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp: 423-432. IEEE Computer Society.

Muthukumar, S., P. Jawahar, 2014. "Sharing and Hit based Prioritizing Replacement Algorithm for Multi-Threaded Applications." International Journal of Computer Applications,Vol 90, No. 12,pp. 34-38.

Muthukumar, S., P.K. Jawahar, 2013. "Cache Replacement for Multi-Threaded Applications Using Context Based Data Pattern Exploitation Technique." Malaysian Journal of Computer Science, 26(4): 277-293.

Muthukumar, S., P.K. Jawahar, 2014. "Hit Rate Maximization by Logical Cache Partitioning in a Multicore Environment." Journal of Computer Science, 10(3): 492-498.

Snavely, Allan and Dean M. Tullsen, 2000. "Symbiotic jobscheduling for a simultaneous mutlithreading processor."ACM SIGPLAN Notices, 35(11): 234-244.

Chaudhuri, Mainak, Jayesh Gaur, NithiyanandanBashyam, SreenivasSubramoney and Joseph Nuzman, 2012. "Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches."In Proceedings of the 21st international conference on Parallel architectures and compilation techniques, 293-304.ACM.

Balaji, S., R. Gautham Shankar and P. Arvind Krishna, "WARP: Workload Nature Adaptive Replacement Policy."

WU, C., M. Martonosi, 2011. "Adaptive Timekeeping Replacement: Fine-Grained Capacity Management for Shared CMP Caches." ACM Transactions on Architecture and Code Optimization (TACO), 8-1.

"Main Page, 2015." Gem5. Accessed. http://www.gem5.org/Main_Page.

"Introduction, 2015." - Gem5. Accessed.http://www.m5sim.org/Introduction.

"PARSEC, 2015. Benchmarks." - Gem5. Accessed. http://www.m5sim.org/PARSEC_benchmarks.

"The Parsec Benchmark Suite, 2015." The PARSEC Benchmark Suite.Accessed. http://parsec.cs.princeton.edu/index.htm.