Adaptive Subset Based Replacement Policy for High Performance Caching

Liqiang He Yan Sun Chaozhong Zhang
College of Computer Science, Inner Mongolia University
Hohhot, Inner Mongolia 010021 P.R.China
liqiang@imu.edu.cn brandon.sunyan@gmail.com zcz917@yahoo.com.cn

Abstract

An Adaptive Subset Based Replacement Policy (ASRP) is proposed in this paper. In ASRP policy, each set in Last-Level Cache (LLC) is divided into multiple subsets, and one subset is active and others are inactive at a given time. The victim block for a miss is only chosen from the active subset using LRU policy. A counter in each set records the cache misses in the set during a period of time. When the value of the counter is greater than a threshold, the active subset is changed to the neighbor one. To adapt the program behavior changing, set dueling is used to choose a threshold from different thresholds according to which one causes the least number of misses in the sampling sets. Using the given framework for this competition our ASRP policy gets a geometric average of improvement over LRU by 4.5% for 28 SPEC CPU 2006 programs and some programs gain improvements up to 50%.

1. Introduction

Cache replacement policy plays an important role in a cache design. The Least Recently Used (LRU) policy has been widely adapted in a microprocessor for the past decades. Recently progresses in this research topic include [1-10].

In this paper, we first propose a Subset Based Replacement Policy (SRP) for high performance caching. The idea is to divide one cache set into multiple subsets and victims are always taken from one active subset at a miss occurring. The reason of dividing is trying to maintain a longer access history through keeping part of the history in a group of blocks in the subset and make the cache more accurately captures the most frequently access blocks. And each subset maintains its own local LRU stack. On a cache miss, the incoming block is put into the local LRU position. And on a cache hit, the block is moved to the Most Recently Used (MRU) position like in LRU policy. The active subset periodically changes based on a count of misses to the set. To adapt the program behavior's changing, we further propose an Adaptive Subset Based Replacement Policy (ASRP) which uses a set dueling [3] technique to dynamically select a threshold from a pool of thresholds according to which one causes fewest misses in the sampling sets. Compare with LRU policy, ASRP obtains a 4.5% of geometric average miss reduction for 28 SPEC CPU 2006 programs, and some programs gain miss reductions up to 50%.

2. Experimental Methodology

We use the framework provided by this competition to do our experiments. All the simulations are done for a single-core processor. The details of the cache configurations are shown in Table 1.

Table 1. Baseline system configuration.

L1 I/D	32KB, 64B lines, 4/8-way, LRU
L2	256KB, 64B lines, 8-way, LRU, 10 cyc. hits
L3	1MB, 64B lines, 16-way, 30 cyc. hits

We select 28 SPEC CPU 2006 benchmarks in our experiment. The cache access traces are obtained using Pin [11]. All the programs are simulated for 100M instructions after fast forwarding 40 billion instructions.

3. Subset Based Replacement Policy

3.1 Motivation

Cache replacement policy tries to keep the ***best*** blocks in a set in order to contribute maximum number of hits for the future accesses. Traditional LRU brings an incoming block directly to the MRU position and assumes that the block has the highest probability to be used again in the future. Such policy is good for workloads whose working sets are smaller than the available cache size or for workloads that have high temporal locality.

When the working set is greater than the cache capacity, the frequently used blocks in LRU policy can be easily evicted from LRU stack by less frequently used blocks or even by other frequently used blocks when it suffers a cache-thrashing problem. At this case, [3] points out that cache performance can be improved by retaining some fraction of the working set long enough that at least that fraction of the working set contributes to cache hits. Our Subset Based Replacement Policy (SRP) is a new method to adopt this idea.

3.2 Subset based replacement policy

Pierre [12] shows that thread migration can boost the performance of single program through exploiting the total cache capacity in multicore processor. The analysis result inspires us that grouping a cache (each group likes a private cache in a core) and keeping part of accessed blocks in a group (like the cache in a core where the program does not run) may also help to reduce the misses of a program. To do this, partition is a normally used scheme. The SRP policy we propose in this section is an adoption of this scheme.

In SRP, we evenly divide one set into multiple subsets. At a given time, only one subset is active which can accept the incoming block. A local LRU stack same as in LRU policy is maintained among the blocks in the subset. On a cache miss, victim is the least recently used block in the active subset, and the incoming block is not moved to local MRU position. And a cache hit will move the block into the local MRU position in the corresponding subset no matter it is active or not.

Fig. 1 explains the rationale. If a block B brought by the miss m_{i-1} is not accessed again before the new miss m_i occurs, then the new incoming block B' will replace B. But if it is accessed, such as the sequence of m_k, h_k, m_{k+1}, miss m_{K+1} will replace the block by m_{k-1} instead of the block K by m_k because K has been moved the local MRU position in h_k. And for a same reason, if an access h_{i+1} hits a block H which was brought by a missed access m_{i+1} in a non-active subset, H will not be replaced when the subset changes to active.

As shown in Fig.1, the active subset changing in SRP policy is based on a count of misses to the set.

When the misses are more than a threshold *X*, the active subset is changed to the neighbor one. In another word, we force up to *X* consecutive misses only replacing the blocks in the active subset, and keep the blocks in other subsets who were brought into the set before the *X* misses staying the cache. For a set with *N* subsets, it must after *(N-1)*X* misses that a subset can change to active again, and during that period of time the blocks can stay in the subset without being replaced. So a greater value of *X*, a longer time that the blocks in non-active subsets can be kept in the set thus a more possibility that an old block in the subsets contributes to a hit if it is accessed again.

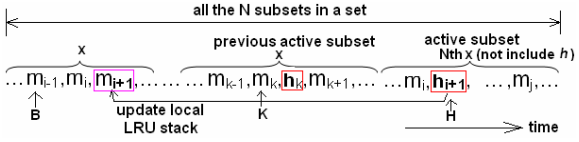


Figure 1. SRP policy.

As an example in Fig.2, a 4-ways cache set is divided into two subsets, and each subset contains two blocks. Subset 0 is set to active at initialization. The threshold of changing active subset is 4. From the figure, our SRP policy has 3 more hits than LRU policy. Keeping the replacement occurring in the active subset during the four misses potentially lengthen the window of access history thus some prior accessed blocks, block 1, 4 and 5 (accessed again after **six** or **five** consecutive misses) for example, can stay in the set and contribute to the hits for the future accesses. On the contrary, the LRU evicts a block if it is not accessed again during a **four** consecutive misses which is less than the history window stored in SRP policy and causes new miss in the future.

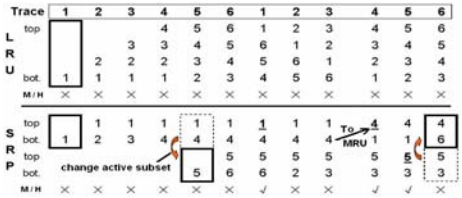


Figure 2. Example of SRP policy.

3.3 Hardware implementation

The hardware structure of SRP policy is shown in Fig.3. For a 16-ways last-level cache (LLC) used in this championship, each set is evenly divided into four subsets. A dedicated 2bits *active_group* counter in the set points to the active subset and the other subsets are inactive. Another *set_miss* counter is used to count the number of misses during a period of time. When the value is greater than the value (all sets use the same value) in the global *threshold* register, the active subset is moved to the neighbor one. In our design, we experientially set the *set_miss* as a 7 bits counter.

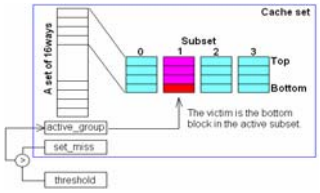


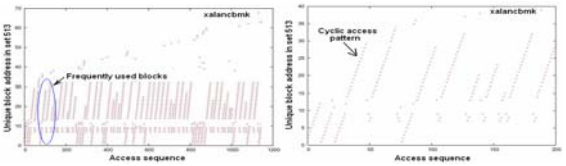
Figure 3. Hardware implementation of SRP policy.

At initialization time, the *active_group* and *set_miss* counters are set to zero, and the value in *threshold* register is set to an experiential value, 4 for example.

3.4 Case study of thrashing workload

Fig.4 shows the cache access pattern of *xalancbmk* in an arbitrary selected set, No. 513, represented by the unique block address. The Fig.4a is the whole picture of the accessing which has a very regular pattern, but the working set (approximate 35 blocks) is greater than the cache capacity, 16, thus causing a thrashing problem. And the Fig.4b is a zoom in snapshot of the first two hundreds accesses. It shows that the frequently accessed blocks express a cyclic pattern.

For such workload, at a cache miss happening SRP uses the LRU block in the active subset as a victim, and keep the other blocks staying in the cache. It assumes that the new incoming block will not be used again in the near future because of the big working set and limited cache size, and hopes the other blocks being retained in the cache can contribute hits for future accesses. When a block has a hit in a subset (no matter it is active or not), it is moved to MRU position of the subset thus the future replacement will select other LRU block if the subset is active at that moment. Through the LRU replacing on miss and MRU placing on hit, the cache can potentially hold maximum number of the frequently used blocks with the limited space.



(a) (b)

Figure 4. Unique block addresses (a: whole trace and b: a zoom in) accessed by *xalancbmk* in set 513.

SRP uses a threshold to control the frequency of active subset changing. A bigger threshold value, a less frequent changing of the active subset thus the more possible is the older blocks can stay in the cache. On the contrary, a smaller threshold value, a more frequent changing the active subset thus having more chance to select victim from multiple LRU blocks in different subsets and letting each LRU block staying in the cache longer than at the case with a bigger threshold.

Fig. 5 shows the cache performance of *xalancbmk* varies with different threshold values. SRP successfully reduces the cache misses comparing with LRU. Fig.5 also shows that the performance of SRP policy is very sensitive to the threshold value. For different thresholds, SRP can hold different fractions of cache data, and at threshold 4 and 8 it obtains the best performance.

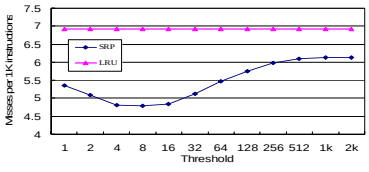


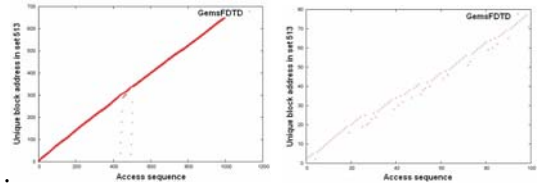
Figure 5. MPKI vs. threshold values for *xalancbmk*.

3.5 Case study of LRU-friendly workload

Fig. 6 shows the unique block addresses accessed by a LRU-friendly workload, *GemsFDTD*, in set 513.

The whole trace in Fig.6a shows the working set is much greater than the available cache capacity, but the zoom in figure, Fig.6b, shows that most of the block reuses are very close (high temporal locality) which favors the traditional LRU policy.

For *GemsFDTD*, for each miss SRP inserts the incoming block to the LRU position which causes the block to be evicted by the near future incoming block and increases the number of misses.



(a) (b)

Figure 6. Unique block addresses (a: whole trace and b: a zoom in) accessed by *GemsFDTD* in set 513.

Fig. 7 shows the varying of cache performance of *GemsFDTD* with the threshold. Same as in Fig.5, SRP is sensitive to the threshold value. Due to the LRU insertion policy, SRP gets more cache misses than LRU at any threshold settings.

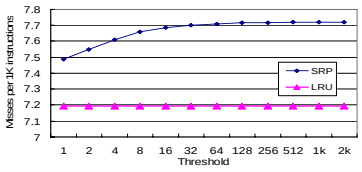


Figure 7. MPKI vs. threshold values for *GemsFDTD*.

3.6 Result

Fig.8 shows the cache performance of SRP policy for 28 benchmarks on three different threshold values. For five benchmarks, *astar*, *cactusADM*, *hmmer*, *sphinx3*, and *xalancbmk*, SRP obtains significant improvements over LRU policy. At the best case, *hmmer*, SRP reduces the cache misses up to 60%. The geometric average speedup of 28 programs is 4% over LRU. Among them, six programs get worse performances than LRU.

Fig.8 also shows that different programs have different sensitivities to the threshold varying. For three of the improved programs , *astar*, *sphinx3*, and *xalancbmk*, the cache performance are improved with the threshold increasing, whereas for some other programs, *bzip2* and *GemsFDTD* for example, the cache performances get worse when the threshold is increased.

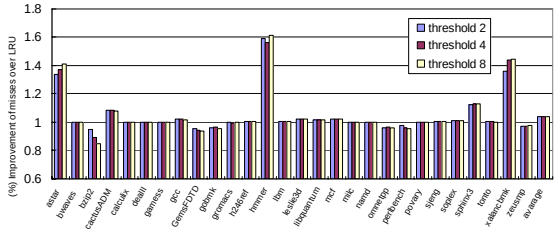


Figure 8. Cache performances of SRP on different thresholds.

4. Adaptive Subset Based Replacement Policy

In last section, fig. 5 and 7 show that SRP policy gets different cache performances with the threshold varying. There isn't a threshold that can get best performance for all the programs. To adapt the diversity of programs and the behavior changing inside a program itself, we propose an Adaptive Subset Based Replacement Policy (ASRP) which uses set dueling [3] to dynamically select one threshold from eight candidates according to which one causes fewest misses in the sampling sets. In ASRP, the block insertion on a miss and block moving on a hit are same as in SRP policy, so we only present the dynamic threshold sampling mechanism in this section.

4.1 Dynamic sampling mechanism

We use a set dueling technique to dynamically select one threshold from a pool of thresholds according to which one causes fewest misses in the sampling sets. The rationale is shown in Fig. 9.

Through experiments, we found that the maximum value of the threshold could be selected for a cache is 128. A threshold that is bigger than 128 always has a similar or worse performance than 128 (see Fig.5 and 7). So in our ASRP policy, we pick eight different thresholds that are 1, 2, 4, 8, 16, 32, 64, and 128. Each threshold has four dedicated sampling sets who use the threshold and not changed forever, thus totally 32 sets are sampling sets and they are evenly distributed in the cache. The other sets in the cache are follower sets whose thresholds are selected dynamically at each time of cache miss.

4.2 Hardware implementation

Fig.9 shows the implementation of our ASRP policy. In each set, a 4 bits *flag* indicates whether a set is a sampling set or a follower set. Only nine values are used that corresponds to eight thresholds and follower set. In addition, eight *miss_counter_group_N* registers are used to count the number of misses in the sampling sets. If a cache miss happens in a sampling set, the corresponding *miss_counter_group_N* is incremented by one.

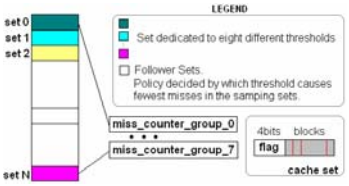


Figure 9. Implementation of ASRP policy.

As shown in Fig.10, when a miss happens in a follower set, the eight numbers in the *miss_counter_group_N* registers are read out, and the smallest value is selected to pick the corresponding threshold from the eight thresholds. Once the dynamic threshold is obtained, the cache set where the miss happens can use it to guide the active subset changing as in SRP policy. For the sampling set, it can use the dedicated **fixed** threshold to do the active subset alteration.

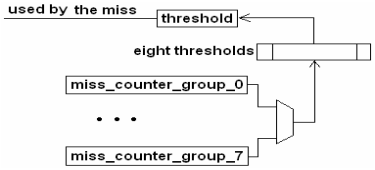


Figure 10. Threshold selection by a follower set.

4.3 Resetting mechanism

The value in the *miss_counter_group_N* will continuously increase with the misses in the sampling sets. If during a period of time, one

threshold, say *X*, causes much more misses in the corresponding sampling sets, then the value in *miss_counter_group_X* will be much greater than the values in other registers. For that duration, a follower set can successfully select a small value in the registers to do the active subset alteration. After that, if the program behavior changes, now the threshold *X* causes fewest misses in the sampling sets, but it can not be selected by the follower set due to the accumulative big value in the register in previous duration.

To solve this problem, we introduce two registers for the whole cache, *last_follow* and *global_follow*. The *last_follow* records which threshold is selected by last cache miss in a follower set, and the *global_follow* records the times of a same threshold selected by follower sets. So if a threshold selected by a follower set on a cache miss is same as the threshold selected last time by no matter the same set or not, the *global_follow* is incremented by one. Otherwise, the *global_follow* is decremented by one. To avoid the accumulative effect, when the value of *global_follow* is greater than a predefined number, 4096 in our policy, all the *miss_counter_group_N* registers are reset to zero, and our policy starts a new sampling process for the future misses.

4.4 Budget

The hardware budget of ASRP policy is computed as follows:

3 bits last_follow // corresponding to 8 thres.
+ 12 bits global_follow // the max. value is 4096
+ 8 * 32 bits miss_counter_group_N
+ (2 bits active_group * 1024 sets)
+ (7 bits set_miss * 1024 sets)
+ 7 bits threshold // the max. threshold is 128
+ (4 bits flag * 1024 sets) // sample or follower
+ (2 bits * 16K lines) // local LRU stack
= 13590 bits + 32K bits = 45K bits

Our ASRP policy only needs 70% of the hardware budget of LRU policy, and only 35% of the total budget provided in this championship.

4.5 Result

Fig. 11 shows the cache performance of ASRP policy in a single core with a 1MB 16-ways LLC. Our policy gets a geometric average speedup of 4.5% over LRU for 28 SPEC CPU 2006 benchmarks, and some programs (*astar*, *hmmer*, and *xalancbmk*) gain about 50% of improvements. Comparing with DIP, our policy gets better performance for most of the benchmarks.

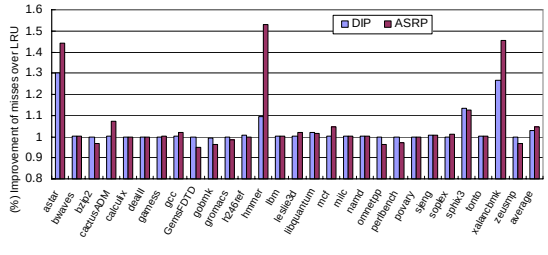
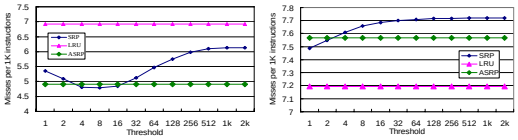


Figure 11. Result of single core experiment.

Fig. 12 shows that the cache performances of ASRP policy are very close to the best cases of SRP policy which means the sampling mechanism does help ASRP to find the best thresholds for different programs.



(a) xalancbmk (b) GemsFDTD

Figure 12. Cache performance of ASRP for xalancbmk and GemsFDTD.

5. Conclusion

In this paper, we firstly propose a subset based replacement policy for the last level cache. It uses one subset to receive incoming block during a period of time and keeps other blocks staying in the cache and contributing hits for future accesses. A threshold is used to control the frequency of active subset changing. Next, we further propose an Adaptive Subset Based Replacement Policy to adapt the program behavior's changing. With the help of set dueling mechanism, our policy dynamically selects a threshold from a pool of thresholds according to which one causes fewest misses in the sampling sets. The experiment result shows the cache performance of ASRP policy is better than DIP and LRU, and the geometric average speedup for 28 SPEC CPU 2006 programs is 4.5% over LRU.

6. Acknowledgement

This work was supported by Inner Mongolia Natural Science Research Foundation Project (No. 20080404MS0901 and 2009BS0901) and the Inner Mongolia University Science Research Foundation Project (No. NJ09009) in P. R. China. The author would like to thank Dr. Jaleel Aamer for his help during the writing of this paper.

7. Reference

[1] E. Perelman et al. Using simpoint for accurate and efficient simulation. SIGMETRICS Perform. Eval. Rev., 31(1):318–319, 2003.

[2] B. M. Beckmann et al. ASR: Adaptive selective replication for CMP caches. In MICRO-2006.

[3] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for high-performance caching. In ISCA-2007.

[4] M. K. Qureshi et al. A case for MLP-aware cache replacement. In ISCA-2006.

[5] M. Kharbutli and Y. Solihin. Counter-Based Cache Replacement and Bypassing Algorithms. Trans. On Computers, 57(4):433–447, Apr. 2008.

[6] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In MICRO-2008.

[7] M. Zahran. Cache Replacement Policy Revisited. In the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) held in conjunction with ISCA, 2007.

[8] M. Zahran and S. A. McKee. Adaptive Block Placement Policy for Cache Hierarchies. In SMART'09:3rd Workshop on Statistical and Machine learning approaches to Architectures and compilation, held in conjunction with HiPEAC 2009.

[9] C. Zhang and B. Xue. Divide-and-Conquer: A Bubble Replacement for Low Level Caches. In ICS'09, 2009.

[10] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In 9th HPCA, 2003.

[11] Pin - A Dynamic Binary Instrumentation Tool. http://www.pintool.org/

[12] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In 10th HPCA, 2004.