**Implementing Cache Replacement Policy – ADPG**

**for High Associativity Cache**

|  |  |  |  |
| --- | --- | --- | --- |
| *Rohan Hulsure*  University of Colorado Denver  [rohan.hulsure@](mailto:rohan.hulsure@ucdenver.edu)  [ucdenver.edu](mailto:rohan.hulsure@ucdenver.edu) | *Ranjan Raut*  University of Colorado Denver  [ranjan.raut@](mailto:ranjan.raut@ucdenver.edu)  [ucdenver.edu](mailto:ranjan.raut@ucdenver.edu) | *Vatsal Vador*  University of Colorado Denver  [vatsal.vador@](mailto:vatsal.vador@ucdenver.edu)  [ucdenver.edu](mailto:vatsal.vador@ucdenver.edu) | *Piyush Deshpande*  University of Colorado Denver  [piyush.deshpande@ucdenver.edu](mailto:piyush.deshpande@ucdenver.edu) |

***Abstract:* Cache has limited resources and it is always better practise to use all the resources to fullest. Because of limited cache size we cannot accommodate all the data at the same time so solution for that is cache replacement policies. When we have high set associativity cache, that time this problem becomes serious challenge. Simplest possible cache replacement policy is Random replacement policy. But that does not give us the satisfactory results and so policy like LRU has high hardware logic to implement for same. Another policy with low hardware cost is Adaptive Demotion Policy considering Global Fluctuations (ADPG) proposed in which priority values of blocks are modified with update logic which suits various applications. In ADPG, according to priority value fluctuations policies like Insertion, Promotion, Demotion and Selection are changed which helps better application performance by effectively replacing cache blocks. In order to achieve this here cache is first partitioned into four parts. We are maintaining one PTR register for every such partition and one prePTR register. Sum of all PTR will be stored in GTR and sum of all prePTR will be stored in preGTR. In such configuration in this experiment we will be having 10 registers in total in which 4 registers will be PTR, 4 registers will be prePTR, 1 register will be GTR and 1 register will be preGTR. At every**

**cache access fluctuations of global priority values will be compared, and application behaviour is detected.**

1. INTRODUCTION

In recent years computer system has become extensive use in wide range of domain and so the amount of handling data. Many research shows that a large part of energy consumption is occupied by data transfer in the memory system.

There are many ways to improve performance and decrease data transfer for cache. One can focus on replacement policy because it can directly affect the performance of cache. But when to apply that policy if we need more hardware so it will add up the energy and other cost of consumption for computer. Which we do not want and to achieve same goal with low energy and hardware consumption is desired.

Although the LRU policy is known as a simple but high-performance cache replacement policy, high-associativity caches hardly adopt the LRU policy due to the following problem. The LRU policy typically requires log2n state bits at all the cache blocks for the n-way set associative cache [1]. The growth of the associativity enlarges not only hardware resources for LRU state bits but also the complexity of the control logic for updating the LRU state bit. This will affect the cycle time of the processor, so it is hard to adopt the LRU policy in high-associativity caches. In addition, the LRU policy does not have a tolerance for scan access patterns because the LRU policy just considers the order of references [2]. ADPG performs very well even with small hardware overhead for high -associative cache. Plus, ADPG suits both scan and thrashing access patterns by changing its insertion, promotion and selection policies [1]. And with set-dueling ADPG selects parameters suitable for running application.

The paper which we followed to develop ADPG has mechanism which will dynamically detect application behaviour and select the suitable policy. In order to achieve what is proposed we first partitioned cache into four partition here. And after every cache access propriety value of each partition is calculated. At every set interval, fluctuations of these priority values are checked and based on difference application behaviour is detected. For example, if totals in all priority values are decreased then we can say scan or thrashing access can occur.

In this project implementation setup was with Gem5 simulator as it is considered to be best possible simulator for computer architecture and for the evaluation and benchmarking SPEC 2017 licensed version was used which was provided by the Department of Computer Science, University of Colorado Denver.

Organization of this report is as follows. Section 2 will go through basics of cache replacement policies and other related work. Section 3 will focus on global fluctuation of ADPG. Section 4 contains performance of ADPG, and Section 5 will conclude.

1. RELATED WORK

So, replacement policies are vital component when we are dealing in set associative cache as having limited amount of available cache requires effective replacement policy to evict block and insert new block. Generally, it is very hard to detect that which block will require in short future so the ultimately detecting behaviour of application is challenge.

Many cache replacement policies exist but not every policy is best in all cases. Simplest replacement policy is Random. It selects any cache block on random logic and evict it from the from cache which will be replaced by new block. So biggest advantage using this policy is that there is no history for this logic. Hence, it does not require any additional hardware which cuts down the implementation cost. Because of its simplicity it is used in many of special purpose processors.

Widely used replacement policy is LRU in most of the case. In LRU, ‘age-bit’ is maintained which is used to make decision which block is least recently used and whether to evict or not. For every time a cache line is used, age of other cache line is changed. And cache line stays longer in cache until its refereed again which unnecessarily increases the overhead when we are dealing with larger cache size with high number of associativity. LRU has low tolerance for scan access patterns.

Round Robin or First In First Out is another example of simple replacement policy for cache. As we know that in queues first inserted element is enqueued from head of the queue first. Same logic is with FIFO replacement policy. Disadvantage of this policy is regardless of how often or how many times the block is accessed; it is replaced once it is in the front of the replacement queue.

Any cache replacement policy which is based on priority value, has a priority value for each block which gets incremented or decremented in different situation. Block with the smallest priority is evicted at a cache miss. Four priority policies are used to decide the application behaviour which are called IPDS policies as follows. Insertion, Promotion, Demotion and Selection.

Insertion policy decides the initial priority value for incoming block. If initial priority value of incoming block is lower than the priorities of the blocks in the set, the incoming block will be evicted before the next cache miss [1]. Whereas if the block has high priority value and it is not re-referenced then it will impact the performance. Small priority value suits best for scan access pattern and large initial priority value will suit temporal locality.

Promotion policy decides that if we get hit then how to promote the priority values. There are two possible approach, Hit Priority (HP) and Frequently Priority (FP). For HP, priority will be set to max to suit temporal locality and in FP priority will be increased by one at every cache hit. Scan access pattern is supported in FP.

Demotion policy decides how to decrease the priority of block after a cache access. We calculate subtraction value for each cache hit or miss. Two proposed methods are Half Of Average (HOA) policy and Average (AVE) policy. For HOA, priority value for all blocks is calculated. Now average for those value is considered half of them and subtract it from each block priority value. For other approach in AVE only average of priority value is subtracted from each cache block. When the hit rate of the cache is high, a cache miss minimizes the priority of all the existing blocks in the set. This causes a harmful effect to the performance by evicting useful data [1]. ADP solve this problem by dynamically calculating subtraction values of accessed set.

Last policy for priority-based cache replacement policy is Selection policy. Selection policy decides the victim block to evict. It will select the block with the minimum priority value for replacement. There are possible cases where more than one block has same lowest priority value. In such case, Left-side Selection (LS) policy or Random Selection (RS) policy is considered. LS selects the leftmost lowest priority block from the set and RS will select randomly.

ADPG suits for variety of application behaviour as above mentioned IPDS policies are decided dynamically.

1. GLOBAL FLUCTUATIONS OF ADPG

ADPG selects the IPDS policies dynamically based on the fluctuations of priority values. Fig. 1 shows the outline of ADPG. Here cache is partitioned into four partitions. Each partition maintains one register which will store the sum of all priority value in one partition which we call PTR register. Partial Total Register (PTR) also has older values which are stored in prePTR. These PTRs are summed into Global Total Register (GTR). GTR also has prior value which is stored in preGTR. So, in total we are maintaining 10 registers to detect the behaviour of application. 4 are PTR, 4 prePTR, 1 GTR and 1 preGTR.

The details are as follows. There are total four states on which selection of IPDS policy is dependent. State 1 and State 2 outputs the policies suitable for temporal locality, and State 3 and State 4 output the policies suitable for scan or thrashing access [1].

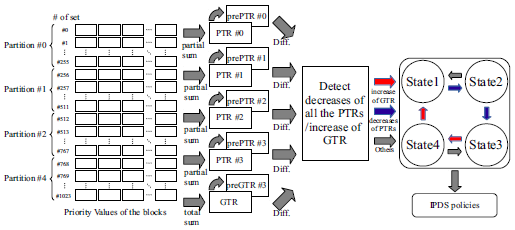


Fig. 1 Outline of ADPG

State behaviour is as follows. For state 1, at every regular intervals GTR and preGTR are compared. If the decrease in GTR value is larger than the set threshold value, then PTR and prePTR are compared and if all PTR are decreased then state is changed to 2. Behaviour for state 2 is same as state 1 but instead of changing to state 2 it changes to state 3. In state 3 GTR and preGTR are compared at regular intervals. If increase is larger than the threshold values, then state is changed to state 4. For state 4 everything is same except next change state is state 1.

Table 1. Cache Parameters

|  |  |  |
| --- | --- | --- |
| **Cache Level** | **Size** | **Associativity** |
| L1I | 32 KB | 8 |
| L1D | 32 KB | 8 |
| L2 | 256 KB | 8 |
| L3 | 2 MB | 16 |
| Main Memory | 4 GB | - |

1. ADPG PERFORMANCE

ADPG is developed and simulated on Gem5 [4] for this whole project. For benchmarks we have used SPEC 2017 [5] licensed test suits provided by Department of Computer Science, University of Colorado Denver. ADPG performance is compared with LRU, MRU and Random cache replacement policies. Variety of benchmark programs are selected to evaluate performance. For ADPG we have used Ruby memory module and compiled and tested on x86 architecture instructions. MESI\_THREE\_LEVEL protocol was used which is strictly inclusive of cache-hierarchy.

Benchmark programs as selected on variety of domain to check behaviour of all the different types of application. Programs are as follows.

* *perlbench:* It is constrained version of perl which tries to remove most operating-system-specific functions, while leaving behind a set of features which are interesting as a CPU benchmark. The SPEC CPU suites intentionally do not spend significant time in operating system services or IO.
* *mcf:* It is written in C and well know example is Single-depot vehicle scheduling.
* omnetpp: It Is written in C++. The model computes summary statistics during the simulation execution to minimize disk space usage and facilitate the results analysis.
* *povray:* POV-Ray is a free and open source ray-tracing application. CPU 2017 version is based on POV-Ray version 3.7.
* *lbm:* It is written in ANSI C, the benchmark is a good source of optimization of different architectures.
* *leela:*  A Go playing engine featuring Monte Carlo based position estimation, selective tree search based on Upper Confidence Bounds. Also used in performing cache coherence.
* *imagick:* The Imagick program in the SPEC benchmark configuration takes an input, manipulates it and produces a transformed image.
* *exhange2:* Written in Fortran 95. No floating-point arithmetic involved and handles large processing.

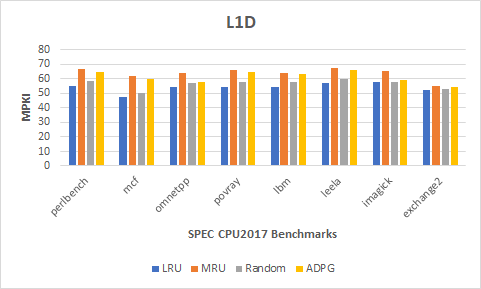


Fig.2 SPEC Benchmark L1D Cache Results

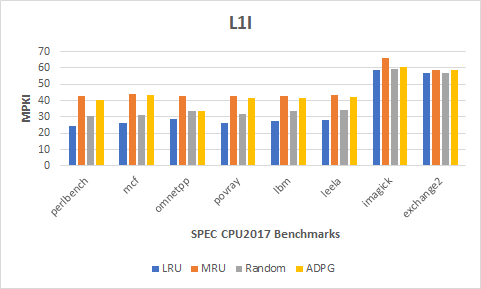


Fig.3 SPEC Benchmark L1I Cache Results

Figure 2 and Figure 3 shows the benchmark results for cache parameters set in the experiment. The MPKI results of LRU (blue) shows least misses per instructions. Compared to MRU and Random policy ADPG performs better in L1D, L1I, L2 and L3 cache levels.

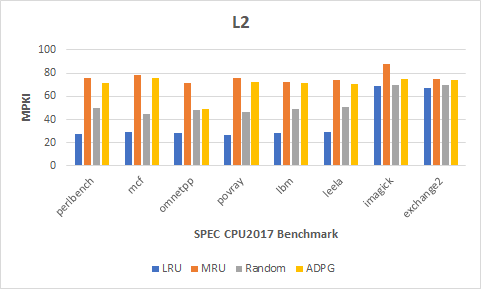


Fig. 4 SPEC Benchmark L2 Cache Results

Figure 4 shows the L2 cache level results achieved. The exchange2 benchmark shows LRU performance almost similar to that of our ADPG compared to the rest of benchmark applications.

In Figure 5 we observe imagick and exhange2 applications showing same performance results of LRU and ADPG. The best results are so far achieved in these benchmark running.

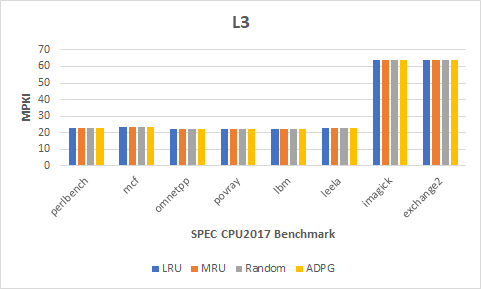


Fig. 5 SPEC Benchmark L3 Cache Results

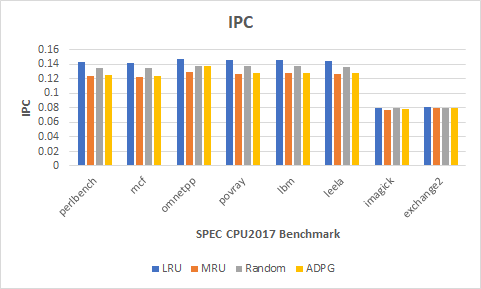


Fig. 6 SPEC Benchmark IPC Results

***Instruction Per Cycle (IPC)***

The IPC is the throughput of the application at given instance of state. In ADPG, we calculate the hit and misses to analyse the IPC at current state of the application. This helps us know the behaviour of application at that particular state.

Figure 6 shows Instructions per cycle results. The ADPG policy for benchmarks except imagick and exchange2 shows less IPC throughput compared to LRU. The performance (IPC) of the policy also depends upon the time complexity of algorithm developed. The simulation experiments are performed to show the performance of the ADPG and the effects of IPDS policies.

In the evaluations, the performance of LRU and ADPG are evaluated at different cache levels. The essential parameters to run the experiment are set from initial configurations. The configuration of ADPG are illustrated like as I2/HP/HOA/LS, in state 1 and 2, whereas state 3 and 4 output I0, FP, HOA and RS. The GTR and PTR values are compared at every hit after miss and so does help to switch between states.

1. CONCLUSION

This project helped us to learn and understand the cache performance with respect to cache misses well. Let us re-iterate the research topics and conclude with the results we observed while running the experiment.

The proposed algorithm implements a high-performance cache replacement technique to handle cache misses. Since cache are initially partitioned into sets, prioritizing and dynamic cache parameters as discussed makes the eviction easy for the victim sets.

Cache level 3 performs better for ADPG and LRU executing with fewer miss rate. The algorithmic complexities involved computations overhead but was an overall effective idea to deal with large cache sizes. By this approach, future architectural designs could be made more efficient in dealing with caches.

1. REFERENCES

[1] Jubee Tada, “A Cache Replacement Policy with Considering Global Fluctuations of Priority Values” in 2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW).

[2] T. S. B. Sudarshan et al., "Highly efficient LRU implementations for high associativity cache memory," in Proceedings of 12th IEEE

International Conference on Advanced Computing and Communications, 2004

[3] J. Tada, et al., “An Adaptive Demotion Policy for High-Associativity Caches,” in Proceedings of international symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART2017), 2017.

[4] <http://www.gem5.org/Main_Page>

[5] <https://www.spec.org/>