Implementation and evaluation of Pseudo-LRU cache eviction policy

Shubhojit Chattopadhyay
Georgia Institute of Technology
ssc3@gatech.edu

Abstract

Although the LRU replacement policy has been widely used in cache memory management, it is well known for its inability to be easily implemented in hardware. Most of primary caches employ a simple block replacement algorithm like Pseudo LRU to avoid the disadvantages of a complex hardware design. In this project, we look into the effectiveness of PLRU, and compare it's performance with conventional algorithms such as LRU, Random and FIFO. Comparison is made on four fronts: Replacement rate, IPC, Energy and storage requirements. Experimental results show that PLRU has a small impact on IPC and number of replacements as compared to LRU, however it incurs far lesser storage/update requirements as compared to other algorithms

1. Introduction

In computer systems, the performance of the memory is often one of the most critical issues. Caching is mainly the simplest cost effective way to achieve higher memory performance. The main parameters that affect the performance of the cache memories are block size, cache size, placement method, replacement scheme and write policy. As the speed gap between the processor and memory is growing, several techniques have been proposed to compensate or tolerate memory latency. Researches have presented several replacement algorithms to improve the performance of the caches and reduce their costs.

Cache memories are commonly organized in three ways: direct-mapped, set associative and fully associative. A set associative cache with n blocks is called n-way set associative cache and holds n blocks per set. Cache designers generally use set associative cache organization because it offers a good balance between hit rates and implementation costs.

In order to make room for a new entry on a cache miss, the cache needs to evict one of it's entries. The heuristic that it uses to choose the entry to evict is called the replacement policy. The fundamental problem with any replacement policy is that it must predict which existing cache entry is least likely to be used in the future.

There are three conventional cache replacement algorithms: "least recently used" (LRU) algorithm, "First In First Out" (FIFO) algorithm, and Random replacement algorithm. The idea behind the LRU algorithm, which is one of the most popular replacement policies, is to reduce the chance of discarding information that will be referred soon. In this algorithm, the accesses to the blocks are recorded and the replaced block is the one that has not been used for the longest time. FIFO algorithm removes blocks as order as they had been brought into the cache and Random replacement algorithm uses a block number that is generated by a pseudo random generator circuit to determine the block that should be discarded.

In this report, we are looking at the Pseudo LRU/tree based cache eviction algorithm's implementation in simplescalar and evaluate it's performance against traditional algorithms for cache eviction. In addition, some energy concerns and storage requirements will also be studied.

The rest of the paper is organized as follows: In section 2, we talk about some related work and how it leads to the need for a new cache eviction algorithm. Section 3 describes how basic pseudo LRU algorithm works. In section 4, we talk about experimental methodology done in order to properly model PLRU. Performance evaluation of the algorithm is presented in Section 5 and then finally conclude in section 6.

2. Related Work

The LRU mechanism uses a program's memory access patterns to guess that the cache line which has been accessed most recently will, most likely, be accessed again in the near future, and the cache line that has been "least recently used" should be replaced by the cache controller. Although the LRU replacement heuristic is relatively efficient, it does require a number of bits to track when each block is accessed, and relatively a complex logic. Another problem with the LRU heuristic is that each time the cache hit or miss occurs the block comparison and LRU stack shift operations require time and power. An example of how the LRU stack is maintained is shown in Figure 1.

To reduce the cost and complexity of the LRU heuristic, Random policy can be used, but potentially at the expense of performance. Random replacement policy chooses its victim randomly from all the cache lines in the set. An obvious way to implement it is with a simple Linear Feedback Shift Register (LFSR). Round Robin (or FIFO) replacement heuristic simply replaces the cache lines in a sequential order, replacing the oldest block in the set. Each cache memory set is accompanied with a circular counter which points to the next cache block to be replaced; the counter is updated on every cache miss.

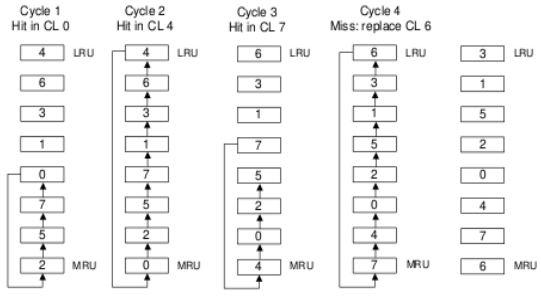

Figure 1. An Illustration of LRU Mechanism.

3. Pseudo least recently used policy

PLRU schemes employ approximations of the LRU mechanism to speed up operations and reduce the complexity of implementation [9], [10]. Due to the approximations, the least recently accessed cache memory is not always the location to be replaced. In the tree-based PLRU replacement heuristic nway-1 bits are used to track the accesses to the cache blocks, where nway represents the number of cache blocks (ways) in a set. Figure 2 illustrates tree-based PLRU (PLRUt) using a 4-way cache memory as an example. The track bits B0, B1, B2 form a decision binary tree. The track bit B1 indicates whether two lower cache blocks CL0 and CL1 (B1=1), or 2 higher cache blocks CL2 and CL3 (B1 = 0) have been recently used. The track bit B0 determines further which one of two blocks CL0 (B0=1) or CL1 (B0=0) has been recently used; bit B2 keeps the access track between cache lines CL2 and CL3. On a cache miss, bit B1 determines where to look for the least recently block (2 lower cache lines or 2 higher cache lines). Bit B0 or B2 determines the least recently used block. On a cache hit, the tree bits are set according to this policy.

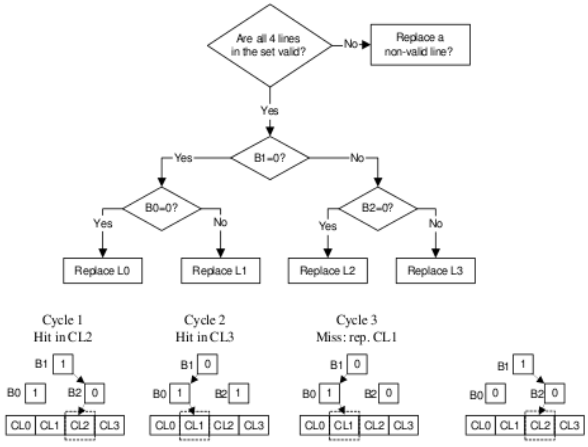

Figure 2. Tree-based Pseudo Least Recently Used Policy.

Table 1 gives storage requirements and corresponding actions taken on a cache hit and a cache miss for all replacement policies discussed. Random policy guarantees the minimal hardware cost, while the LRU hardware cost increases dramatically for caches with associativity larger than 8. In 2-way cache organizations PLRU policy requires only one track bit, and shows the same performance as other LRU-based polices, hence it is an obvious choice for 2-way caches.

The following table shows influence of each memory access over history bits

Referenced Block	AB/CD bit	A/B bit	C/D bit
A	1	1	no change
B	1	0	no change
C	0	no change	1
D	0	no change	0

Table1. Updating nodes during a Hit and Miss

AB/CD bit	A/B bit	C/D bit	Replaced/LRU block
0	0	0	A
0	0	1	A
0	1	0	B
0	1	1	B
1	0	0	C
1	0	1	D
1	1	0	C
1	1	1	D

Table 2. Finding the least recently used block

4. Experimental methodology

Performance evaluation of different cache replacement policies has been done using the sim-outorder simulators from the Alpha version of the SimpleScalar toolset [3]. The original simulators have been modified to support additional pseudo-LRU replacement policies and collect corresponding statistics. In order to allow tracking of the dynamic behavior of caches, the sim-outorder simulator has been modified to print interval statistics per specified number of instructions.

Selected benchmarks from SPEC CPU2000 [6] suite have been used as a simulation workload, representing the state-of-the-art applications for high-performance computing.

Selected SPEC CPU2000 integer and floating point benchmarks were used with reference data inputs. The first 1 Billion instructions were simulated. For each benchmark a number of simulations have been run for various cache organizations - 1-, 2-, 4-, 8-, and 16-replacement policies Random, FIFO, LRU and PLRU and various cache sizes. The first set of experiments concentrates on performance of split first level instruction and data caches (L1I, L1D) with 4KB, 8KB, 16KB, and 32KB sizes.

The second set of experiments considers performance of the second level unified cache memory (L2U) with 32KB, 64KB, and 128KB, assuming direct-mapped 4KB first level caches for both data and instructions. The experiments for first level cache have also been conducted for OPT replacement policy. Although OPT requires a perfect knowledge of future references, and hence cannot be implemented, it is useful as a yardstick in exploring potentials of replacement policies. In all experiments a cache line size is 32 bytes. In order to provide the monitoring of cache miss rate during program execution, cache misses are recorded for each 100000 instructions, as well as for the whole application.

5. Performance Evaluation

We evaluate pseudo lru with other cache replacement policies on four fronts: Miss rate, IPC, power consumption and storage/computation.

For most evaluation purposes, we have assumed Intel Clarkdale's cache configuration, since it has been used for a while in mainstream market and it's cache configuration is available in public. The typical cache configuration of Intel Clarkdale is as shown below:

- Split L1 Instruction and Data Cache – 32KB (8 ways)
- Unified L2 Cache – 256KB (8 ways)

The above configuration is used for collecting all results unless otherwise mentioned. Also, Clarkdale uses plru for both L1 and L2 cache.

For obtaining these results, Simplescalar sim-outorder simulator was used. The simulator was modified to include Pseudo LRU eviction policy and SPEC 2000 benchmarks were run for 10 million instructions from the start.

A) Replacement rate

The most important metric to be calculated with any new cache eviction policy is to compare the eviction policy's replacement rate as compared with the traditional replacement policies.

The following figure shows the cache replacement rate for the Clarkdale configuration (Miss rate when 32KB cache. Num of ways = 8).

	plru	lru	random	fifo
ammp	0.00262	0.001	0.002	0.002
eon	0.00249	0.002	0.0027	0.0024
equake	0.00328	0.0058	0.0064	0.0062
fma3d	0.00236	0.0017	0.0021	0.002
gcc	0.00466	0.0061	0.0096	0.0091
mcf	0.125	0.1249	0.125	0.125
mgrid	0.081	0.0807	0.0812	0.0811
parser	0.0069	0.0067	0.0071	0.0071
perlbmk	0.008	0.0078	0.01	0.0093
swim	0.0044	0.0038	0.0043	0.0043
twolf	0.002	0.0001	0.0003	0.0003
Average	**0.0220645455**	**0.0218727273**	**0.0227909091**	**0.0226181818**

As we can see, plru performs better than Random and FIFO however, LRU outperforms plru slightly.

Next we look at the effectiveness of plru in terms of varying number of ways (in L1 cache of 32KB)

Eviction rate with plru, varying with number of ways			
	Ways = 4	Ways = 8	Ways = 16
ammp	0.0149	0.00262	0.0355
eon	0.0114	0.00249	0.0472
equake	0.0102	0.00328	0.0502
fma3d	0.0069	0.00236	0.0387
gcc	0.0281	0.00466	0.0736
mcf	0.125	0.125	0.125
mgrid	0.0808	0.081	0.082
parser	0.0104	0.0069	0.0278
perlbmk	0.0242	0.008	0.091
swim	0.0041	0.0044	0.0059
twolf	0.0004	0.002	0.0136
Average	**0.0287636364**	**0.0220645455**	**0.0536818182**

Interesting observation is that eviction rate reduces when the number of ways increases from 4 to 8. However, increasing from 8 ways to 16 ways reduces the effectiveness of plru and hence the eviction rate is higher for 16 ways.

B) IPC

We now look at impact on IPC of various policies and cache configurations.

The following figure shows the average IPC with increasing cache sizes:

Average IPC of SPEC benchmarks				
Size of cache	**plru**	**lru**	**random**	**fifo**
4K, 8 ways	19.5868	19.6747	19.5727	19.4325
16K, 8 ways	20.5389	20.6814	20.5187	20.002
32K, 8 ways	20.103	20.1049	20.109	19.7572

We can observe that IPC of plru is only slightly lesser than lru in all cases. Random comes third and FIFO is the worst. Also, we observe that difference between plru and random keeps diminishing with increasing caches sizes.

Next we look at using plru in multiple level of caches and how it impacts IPC.

We see that a higher IPC is observed in using PLRU in only L2 as opposed to only in L1. This is because when we are using PLRU in only L2, we are using LRU in L1 and that gives a higher IPC. However, when we are using PLRU in both L1 and L2, there is a decrease in IPC, which can be attributes to the cumulative degradation of IPC in both levels of caches.

IPC when PLRU is in caches			
	Only in L1	**Only in L2**	**In both L1 and L2**
ammp	2.0764	2.175	2.044
eon	1.4218	1.4459	1.3967
equake	1.6454	1.6898	1.6441
fma3d	1.9814	1.9494	1.8788
gcc	1.2344	1.2278	1.1261
mcf	2.1915	2.1915	2.1915
mgrid	2.2806	2.2791	2.2775
parser	1.6115	1.6043	1.5868
perlbmk	1.2046	1.2276	1.1549
swim	1.9364	1.9365	1.9353
twolf	2.1732	2.1822	2.1682
Average	**1.7961090909**	**1.8099181818**	**1.7639909091**

C) Power Consumption

Power Consumption is calculated using CACTI [10][11] simulator. CACTI takes various cache configurations as input and then computes various cache related statistics like Dynamic Read Energy Consumption, Area and Wire complexity etc.

In this part, we are mainly focused on trying to find the extra energy consumption that occurs due to using plru policy. Power consumption tradeoff occurs mainly due to the amount of energy that would be required due to differences in cache evictions, causing differences in Hit rate. More the Hit rate in L1 cache, less the number of accesses in L2 and hence energy might be slightly lesser.

For comparison purposes, we will again be considering the clarkdale configuration (L1 32KB and 8 ways, L2 256KB and 8 ways). We will then find

Total energy consumption = Energy consumed in accessing L1 + energy consumed in accessing L2 (due to each policy)

In the above formula, *Energy consumed in each level of cache = Dynamic read energy per access*Total number of accesses in that cache level*

where *Dynamic read energy/access = Dynamic read energy of Data array/per access + Dynamic read energy of Tag/access*

The observed energy consumption are shown below:

So, we can see that plru overall consumes slightly more energy as compared to lru on account of more replacements. However the difference in energy consumption for 10 million cycles is very little.

Dynamic read energy (nj)				
	plru	lru	random	fifo
ammp	57142	58464	58729	58720
eon	92826	91858	92133	92202
equake	81051	84276	84447	84396
fma3d	45905	48836	48870	48859
gcc	117218	115467	116937	116704
mcf	266317	266285	266286	266285
mgrid	159076	158916	159090	159058
parser	91540	90413	90553	90517
perlbmk	117965	113083	114079	113727
swim	60143	59983	60078	60063
twolf	64471	64415	64438	64421
Average	104877.63636	104726.90909	105058.18182	104995.63636

D) Storage/Updating eviction vector requirements

For Complexity, we observe that the number of bits required to represent LRU is far greater than PLRU. The following table indicates the number of bits required to be updated, each time a cache block is referenced

Policy	Storage Requirement(bits)	Action on cache hit	Action on cache miss
Plru	*nsets*(nways - 1)*	Update the tree bits	Update the tree bits
LRU	*nsets*nways*log_2(n ways)*	Update the LRU stack	Update the LRU stack
Random	*log_2(nways)*	None	Update LFSR register
Fifo	*nsets*log_2(nways)*	None	Increment FIFO counter

Complexity comparison of different cache eviction policies

As an example, for a standard Intel clarkdale processor, with L1 cache size of 32KB and number of ways = 8, the number of sets = 64. The number of bits required to be updated in each access is:

Policy	Storage/Update Requirement(bits)
Plru	*448*
LRU	*1536*
Random	*3*
Fifo	*192*

Thus, it is clear that plru requires far fewer bits as compared with lru.

6. Conclusion

As the speed of processors increases much faster than the decrease in memory latency, eliminating cache misses will continue to be extremely important for improving overall processor performance. With caches becoming more set- associative, cache replacement policies will gain even more significance. This research encompasses a detailed performance evaluation of the common replacement policies using SimpleScalar toolset and the SPEC CPU2000 benchmark suite.

Our results show that, for data caches we observed that plru policy performance is near the performance of the best, lru. There is a small loss in IPC and a slightly higher power consumption in plru as compared with lru. However, plru definitely requires fewer number of bits that need to be stored and updated during a cache access. This is particularly beneficial for reducing latencies of L1 cache hits.

7. References

[1] Ackland B., Anesko D., Brinthaupt D., Daubert S.J, Kalavade A.., Knoblock J., Micca E., Moturi M., Nicol C.J., O'Neill J.H., Othmer J., Sackinger E., Singh K. J., Sweet J., Terman C. J., and Williams J., "A Single-Chip, 1.6 Billion, 16-b MAC/s Multiprocessor DSP," IEEE Journal of Solid-state circuits, Vol. 35, No. 3, March 2000, pp. 412-423.

[2] Belady, L.A., "A study of replacement algorithms for a virtual storage computer," IBM Systems Journal, 5(2):79- 101, 1966.

[3] Burger D., Austin T., "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison Technical Report #1342, June 1997.

[4] Cantin J. F, Hill M. D., Cache Performance of the SPEC CPU2000 Benchmarks,

http://www.cs.wisc.edu/multifacet/misc/spec2000cache- data/

[5]Hennessy J. L., Patterson D., Computer Architecture: A Quantitative Approach, Third Edition, Morgan Kaufmann Publishers, 2003.

[6] Henning J. L., SPEC CPU2000: Measuring CPU Performance in the New Millennium, IEEE Computer, vol. 33, no. 7, July 2000, pp. 28-35, http://www.spec.org/osg/cpu2000/papers/COMPUTER_20 0007-abstract.JLH.html

[7] Wong W., Baer J-L., "Modified LRU Policies for Improving Second-level Cache Behavior," Proceedings of the 6th International Symposium on High-Performance Computer Architecture, Toulouse, France, January 2000

[8] Hussein Al-zoubi, Ar Milenkovic, Milena Milenkovic, "Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite "

[9] Hsin-Hsien Lee, class notes ECE 6100, Georgia Institute of Technology

[10] http://etd.nd.edu/ETD-db/theses/available/etd-07122008-005947/unrestricted/ThoziyoorS072008.pdf for computation of dynamic power using CACTI

[11] http://www.hpl.hp.com/research/cacti/ HP Cacti simulator