CRC: Protected LRU Algorithm

Yuval Peress, Ian Finlayson, Gary Tyson, David Whalley

CRC: Protected LRU Algorithm

Yuval Peress peress@cs.fsu.edu
Ian Finlayson finlayso@cs.fsu.edu
Dr. Gary Tyson tyson@cs.fsu.edu
Dr. David Whalley whalley@cs.fsu.edu

Abstract

Additional on-chip transistors as well as more aggressive processors have led the way for an ever expanding memory hierarchy. Multi-core architectures often employ the use of a shared L3 cache to reduce accesses to off chip memory. Such memory structures often incur long latency (as much as 30 cycles in our framework) and are configured to retain sets as large as 16 way. A baseline replacement algorithm that has proven itself over and over again is the Least Recently Used (LRU) policy. This policy seeks to replace lines that were used least recently, which works well thanks to temporal locality. This paper seeks to improve on LRU by taking advantage of the 16 ways available to include a bias for replacement. By keeping track of the relative use of each lines, some frequently used lines may become "protected". By providing protection for such lines we have managed to reduce the miss rate to 62.89% from LRU's 70.08%. Using a memory reference trace, we also demonstrated that the best replacement algorithm, an oracle which knows about future accesses, could only provide a 58.80% miss rate for our benchmarks.

1 Framework

The Cache Replacement Competition (CRC) provides the competitors with a trace driven simulation environment. Competitors were provided access to the class structure holding meta-data and functions responsible for selecting the evicted line in each L3 cache set.

The competition can be subdivided into 2 phases: a single and multi core execution. The L3 Cache configuration maintains 16 way sets and 64 byte lines regardless of the number of cores, and allows for 1MB per core. In each phase, submissions are judged by the performance of the L3 cache replacement algorithm. Each submission is limited to 8 bits per cache line as well as 1 Kbits of global memory. Due to the long latency of the L3 and the low frequency of L3 cache accesses, algorithm complexity was not limited.

For our tests we chose 9 SPEC2006[1] benchmarks (astar, bzip2, gcc, h264ref, hmmer, libquantum, mcf, perlbench, and sjeng). These benchmarks provided us with a large variety of L3 cache misses (capacity, conflict, and compulsory).

2 Exploring the LRU Algorithm

The most common cache replacement algorithms are based on the idea of Least Recently Used (LRU). Here, the goal is to evict the lines that were accessed the longest time ago, because temporal locality tells us that they are unlikely to be needed again very soon. Algorithms in this family include the popular Pseudo-LRU, the LRU-K algorithm [5], and 2Q [3]. Another common approach in cache replacement algorithms is to try and replace the lines that are Least Frequently Used (LFU). The idea here is that lines that have been used many times are likely to be needed again.

The LRU algorithm was provided with the CRC kit and was a natural starting point for our work. Each of our benchmarks was executed using the LRU algorithm and provided us with a list of hits/misses and replacements within the cache as well as a baseline miss rate (70.08%). Using these results we began scanning for patterns that might reveal shortcomings of the LRU algorithm. Such a pattern was revealed when a line was brought into the L3 cache for a single reference, evicting a line that was referenced many times before. The heavily referenced line was then brought back into the cache to be used in what must have been another loop or iteration of an outer loop.

For example, a memory address X was used N_x times before being evicted by X'. X' was then only used a few times before X was brought back into the set. It is important to note that on the second appearance of X, it does not necessarily replace X'. Instead, it is possible for X' to remain in the set unused simply because it was referenced recently. This pattern brings up an interesting question, “Should *X* be kept in memory for being accessed many times?”

There have been several attempts to create an improved cache replacement algorithm by combining the LRU and LFU concepts. In [4] it is argued that there is a spectrum of cache replacement algorithms that include both least recently used and least frequently used based on how much weight is given to each policy. They use this weight to calculate a combined recency and frequency value for each line which is used in replacement decisions.

Another replacement algorithm that uses both recency and frequency is presented in [2]. Similar to our work, they identify a set of least recently used lines and, among those, choose to replace the line that is least frequently used. With each line they store a counter indicating the number of accesses to that line. Since our works are similar, it becomes important to note the differences. First, [2] uses both the LRU stack (4 bits) and an 8 bit counter value per line which would not fit in the CRC framework. Second, their work attempts to predict when caching a line is not necessary whereas our work will always cache a line that is fetched into the L3. Finally, and perhaps most importantly, their work first selects the *N* least recently used lines of which they select the least frequently used to replace. Our preliminary work revealed that more misses can be avoided if frequently used lines are given protection first. Assuming that 12 lines are to be given protection from eviction, [2] would select the least frequently used of the 4 least recently used lines. Such a scenario could end poorly if a phase of execution is about to repeat and all 4 lines will be needed again. Our algorithm would first protect such lines that were heavily used previously, and select a replacement from the remaining lines.

3 The LRU + Protected MUs

To further explore the questions above, an additional concept was added to each cache set: the MU (Most Used). The initial test used an arbitrarily large counter to keep count of the total number of references per memory address. This counter value was maintained in global memory space even while the memory references was evicted from the cache. The LRU algorithm was then augmented to never evict the 4 most used addresses within the set. Such a configuration provided us with a miss rate of 69.42% (0.94% lower than LRU alone).

Since such a design is unreasonable to implement in hardware, a second concept arose. In the second design, each cache line was augmented with an arbitrarily large counter. This counter was incremented on each access and zeroed when the line was replaced. The remaining algorithm was the same, when a line needed to be replaced the top 4 MU lines were removed from the list of LRU lines before selection. The new configuration provided an average miss rate of 67.83% (2.25% lower than LRU). As it turned out, keeping the counter around for too long resulted in the cache becoming polluted with lines that were accessed many times in a previous phase of execution becoming impossible to evict. The new results drove us to limit the counter size in hopes of further removing such stagnation in the cache and meeting the space requirements.

Replacing the original 64 bit global counter per memory address, the first configuration augmented the LRU algorithm (4 bits/line) with a 4 bit counter/line and protect the 4 MU lines during an eviction. This approach increased our miss rate from 67.83% to 67.89% (still beating LRU by 2.19%). Since this design appeared to be working well, more configurations were run, allowing for anywhere from 1 to 15 protected MU lines and anywhere from 3-5 bit counters (Figure 1). Of these, we found that the best configuration across the board was 14:3 (14 protected MU lines:3 bit counters), with an average miss rate of 62.89% (7.19% lower than LRU alone). While such a configuration provided the best average miss rate, a different approach reveals a different view. Since a single benchmark provided large gains for each additional MU, we wanted to find the best configuration without giving too much bias to a single benchmark. To do this, each benchmark’s best performance configuration was listed: 7:3 (astar), 10:3 (bzip2), 10:3 (gcc), 14:3 (h264ref), 15:3 (hmmer), 15:3 (libquantum), 12:4 (mcf), 14:3 (sjeng). Using the unweighted average we find that the ideal configuration (rounded down) should be 12:3.

The 12:3 configuration provides a miss rate of 63.6% (6.48% lower than LRU alone) and costs 7 bits per line with no additional global data. The

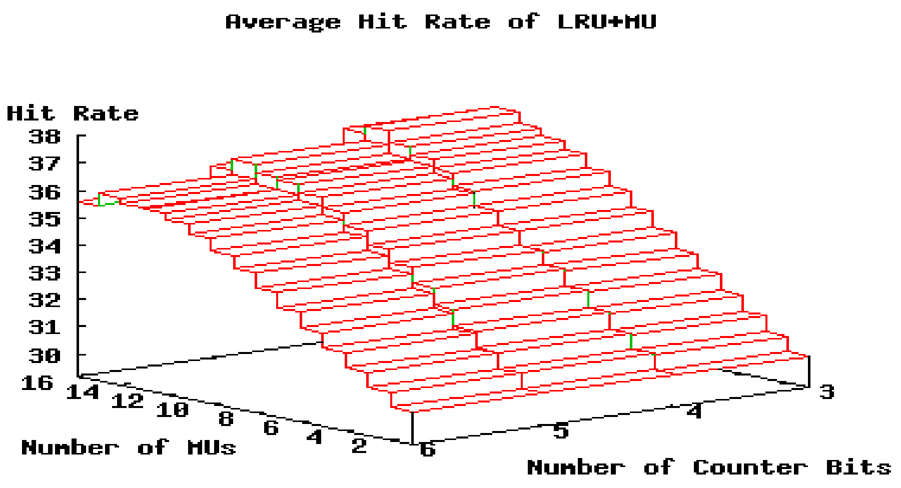

Figure 1: Hit rate of different LRU + Protected MU configurations

first 4 bits are used to store the LRU stack position in a 16 way cache; the following 3 bits provide a counter with a max value of 7. LRU would now be kept as it was previously, with each access to a line simply shifting that line stack index to 0 and incrementing the stack locations that were below it so that the stack values may remain continuous. The additional *counter* will be incremented on each access to a cache line. If the counter ever rolls over, all the counters within the set are right shifted by 1. Such a divide by 2 approach helps preserve the top most used values, while not allowing such values to become stagnant between phases of execution. Since algorithm complexity is of no issue, the top 12 counter values (in our 12:3 configuration) are ignored when looking for a line to replace using a simple scan through. Instead, the LRU of the remaining 4 lines is evicted and its counter set to 0.

An interesting observation is found when looking at the addition of counter bits. On average between all benchmarks and each number of protected MUs, we find that the addition of the 4^{th} bit increases the miss rate by 0.12%, while the 5^{th} bit further increases the miss rate by 0.24%. Such data further supports the concept of stagnation within the protected MUs. The additional bits allowed cache lines to become so dominant within the set that they became hard or even impossible to evict once they became useless.

A final run was performed where each benchmark produced a trace of which memory addresses are accessed. On a following run of the benchmark, the trace is loaded and is used during line eviction. When a line needs to be replaced, the oracle scans through the trace using only references that have not been seen yet. The lines in the set are check off as each memory address is seen until only a single line is left. This final line is the one that needs to be replaced since it will not be used for the most number of cycles. Using this oracle we found that the best replacement we could expect for our benchmarks will provide a 58.80% miss rate.

For completeness, the execution of 4 threads was performed. The 4 threads were selected by which threads had the greatest standard deviation between all the configurations we ran (bzip2, gcc, hmmer, and libquantum). Configuration for the 4 threaded execution followed the parameters of the competition by increasing the L3 cache size from 1MB to 4MB and altering no other parameters. The results showed that gcc and hmmer both were aggressively competing for the protected slots such that below a threshold of 6 MU's a 1:5 configuration was best with an average miss rate of 59.95%. Once the threshold is met performance improves with each additional protected MU, our selected 12:3 configuration provided a 56.96% miss rate (4.07% lower than the 61.03% miss rate provided by LRU).

4 Conclusion

In this work we have demonstrated that with large sets, the LRU algorithm can be improved by accounting for use frequency. With only an additional 3 bits per line over LRU, we can protect lines which may have been referenced a long time ago but used many times before. To prevent stagnation of such counter based protection, the counter is limited to 3 bits and upon saturation all counters within the set are right shifted by 1.

We would also like to compare our results with [2] since the concepts driving both works are very similar. Unfortunately, [2] used an older set of benchmarks which have only a single benchmark in common with our work, mcf. In this benchmark, their best performing scheme performed at 1% improvement from LRU (translating to a rough miss rate of 91%), where as our worst submitted configuration achieved a miss rates of 86.36% (or a 6.23% improvement over LRU when scaled as in [2]).

Using SPEC-INT CPU2006 benchmarks we have demonstrated that a 12:3 configured Protected LRU can achieve an average miss rates 6.48% lower than LRU alone and a 14:3 configuration further reduce the miss rate by 0.71%. We further demonstrated that additional counter bits adversely affect performance due to stagnation. If we were to normalize our results with the oracle performance; we would find that LRU's miss rate is 119.18%, while our configurations (10:3, 12:3, and 14:3) have 109.54%, 108.16%, and 106.96% miss rates respectively. While a 14:3 configuration did perform better under our benchmarks, we believe that a 12:3 configuration is more appropriate for general purpose applications. Our submission includes 10:3, 12:3, and 14:3 configurations.

5 Acknowledgements

This research was supported in part by NSF grants CNS-0615085 and CNS-0915926.

References

[1] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006.

[2] Haakon Dybdahl, Per Stenström, and Lasse Natvig. An lru-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. *SIGARCH Comput. Archit. News*, 35(4):45–52, 2007.

[3] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. *SIGMETRICS Perform. Eval. Rev.*, 27(1):134–143, 1999.

[5] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 297–306, New York, NY, USA, 1993. ACM.