Adaptive Cache Replacement Policy

Awrad Mohammed Ali, Stacy Gramajo, Neslisah Torosdagli
Dept. of Computer Science
University of Central Florida
Orlando, USA

ABSTRACT

Adaptive cache is mimicking best replacement policy through observing more than one cache replacement algorithms, such as Least Recently Used, LRU and First In First Out, FIFO. This is accomplished by tracking the policy that would lead to a better performance than using each policy individually. One way to determine the performance of a processor is using CPI. According to Iron Laws of Memory Hierarchy, latency introduced by the caches is one of the factors, which have negative impact on CPI. In order to overcome this bottleneck, intelligent caching mechanisms is a must. In the referenced paper, an adaptive replacement policy is applied to decrease latency hence to increase CPI. In this research, as proposed in the referred paper, we defined new data caches in SimpleScalar in addition to data level 1 and data level 2 caches, each of which with different replacement policies. We refined the adaptively selection policy to cover all local and global metrics. Furthermore, we applied adaptivity to both level 1 and level 2 caches, and also just level 2 caches to observe impact of adaptivity more clearly.

I. INTRODUCTION

The distance between the processor and main memory has been increasing requiring enhanced caching algorithms, which affects processor performance when accessing memory. To improve the caching algorithms, computer scientists have been expanding cache sizes and hiding latency through prefetching and out-of-order execution. However, there hasn't been much work on refining cache replacement algorithms. Such algorithms can be found in the SimpleScalar, a hardware simulation tool developed by Todd Austin where it includes more than one type of replacement policies [1]. In our implementation, we will be using two types: Least Recently Used (LRU) and Least Frequently Used (LFU). As the names suggest it, LRU is a type of cache replacement algorithm that swaps out the least recently used item; whereas, LFU replaces the item in the cache that has the lowest reference frequency.

Each replacement policy has its own strengths and weaknesses. For example, LRU works well where access is made mainly to the most recent items, such as an application computing average temperature from the last two hours. In the other hand, LFU performs great in conditions where large regions of blocks are used only once from commonly accessed data. By combining these two algorithms and choosing them in their best situation, this would enhance the cache performance.

The following section will explain the different caching algorithms and branching predictor that can be combined to improve performance. We extended the referred paper by two of the following ways

- Using global and local metrics for making a decision of replacement policy.
- Using adaptivity replacement policy either for data level1 and level2 caches or for only data level2 cache.

Adaptive cache management involves distinguishing the amount of miss rate over the total for each algorithm in the SimpleScalar. The method will follow up by choosing the algorithm with the least amount of cache misses. In addition, we will be implementing a branch prediction method that will track the most frequently used policy.

The rest of the paper will thoroughly depict the object and implementation of the adaptive caching in the hardware and software. The paper will be followed by our results and conclusion of the research.

II. ADAPTIVE CACHING PAPER IMPLEMENTATION (BASELINE)

In this section, we discuss the adaptive replacement technique implemented and illustrated from the research paper. Afterwards, we will explain how we changed their version and incorporated it into SimpleScalar.

A. Hardware Structure

As previously mentioned, the adaptive cache policy chooses the algorithm that performed better on recent

memory accesses. To do this, the research paper explained that two extra elements were added into the hardware.

The first structure added is a set of parallel tag arrays with identical sets and set associativity as the regular tag array of adaptive cache. The tag array would track down what would have been in the regular tag array for the replacement policy that was used.

The second structure is a miss history buffer. The buffer will track the past performance for each cache set. Whenever there is a cache miss that occurs in one of the policy, the adaptive algorithm will update the miss history buffer.

Moreover, the miss history buffer contains a bit-vector of m bits, where m is equal to the cache associativity. It only records the current misses for only one policy. If the miss occurs in both methods, then nothing will be recorded.

B. Software Implementation

For every memory block reference, the parallel tag structure for caches A and B are updated as well as the miss history buffer. When there is a reference miss in the cache, the adaptive replacement algorithm is used.

Specifically, in the algorithm, it first compares A and B misses. If A is less than or equal to B, then A method is chosen. If A has a cache miss and the block wants to evict is in adaptive cache, the adaptive cache also evicts it. Otherwise, the adaptive cache will evict any of the block that is not in A. The same algorithm can be implemented in B, if this policy is chosen.

III. OUR IMPLEMENTATION

Section three of the paper explains about how we applied the adaptive cache algorithm into the SimpleScalar.

A. Overview of Adaptive Cache

The goal of using adaptive cache is to implement the best caching algorithm. In a given situation, as previously mentioned, we used LRU, as Policy A, and LFU, as Policy B, and Adaptive policy. Each of these algorithms are best implemented in different types of situation. We are trying to reduce the overall number of miss rate and increase the execution time of the whole program.

B. Additional Structures (our approach)

In the basic implementation where adaptively is applied to both data level 1 and data level 2 caches, 4 extra caches are defined in SimpleScalar. That is, there are totally 3 data level 1 caches and 3 data level 2 caches. All new caches are registered to stats database (sdb) so that their collected statistics are displayed at the end of the run.

Each cache has a tag describing as if it is policy A, policy B, or policy Adaptive. In addition, each cache is bound to its sibling with policies A/B/Adaptive cache by pointers. Thus, when cache_access is called, according to policy tag and sibling pointers, other data level1/level2 caches can be accessed.

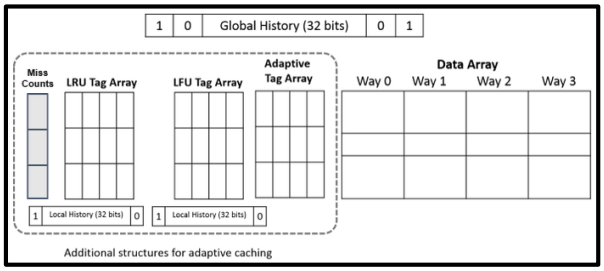

Figure 1: shows the two tag arrays, three 32-bit size arrays, and a miss count buffer.

As shown in Figure 1, we implemented three tag arrays for each policy, two local history arrays, and one global array. The tag array is the same size the data cache size; whereas, the arrays in our implementation are statically given thirty-two bits.

The two local history arrays are used to keep track of how many misses or hits has recently occurred. When there is a cache miss, one is placed into the array; however, when there is a cache hit, zero is inserted. When it comes time to choose either array to execute, the program will check both arrays and choose the one with the least number of bits (or hit misses).

Unlike the other's implementation on the adaptive cache algorithm, we wanted to implement a branch-like prediction method. To achieve this idea, global history array was added, where it keeps track of the algorithm policy that is currently executing. When LRU is used, zero is inserted; whereas, one will be added when LFU is used. By counting the number of frequencies of each policy, we were able to have an idea which next policy had a high chance of executing.

However, with these implementations added to the code, randomization was an issue. To resolve it, three history buffers were added for the global and local histories. Just like their parent arrays, the history arrays are also thirty two bit in size, as shown in figure 2. Their main objective is to let the program knows when their parent arrays are filled. To accomplish this, one was continuously shifted 1 into the array. When there was

thirty two ones, the program starts implementing the branch-like prediction method into the adaptive cache algorithm. In the branch-like prediction, global and local histories arrays bits are counted. Then the results go through a voting algorithm which dictates what policy will be chosen to run with the adaptive cache.

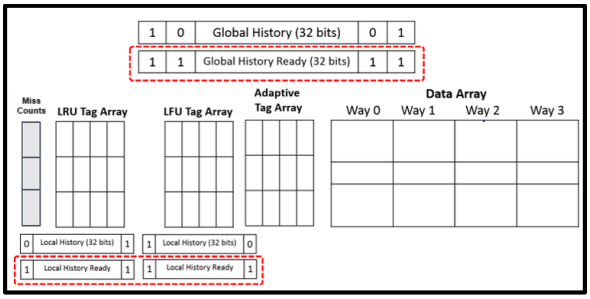

Figure 2: shows the three history buffers that will be used to determine when branch like prediction algorithm will be executed.

As mentioned previously, we added various arrays and buffers into the SimpleScalar. Adaptive tag caches, GlobalHistory, LocalHistory, miss count buffer, HistoryReady buffers were added into the **main.c** file. In order to be able to access global variables declared in **main.c**, they are declared as externs in **sim.h**, which enables usage of this variables in the rest of the sim-cache part of the SimpleScalar project.

The policy and adaptive tag caches were initialized in **sim-cache.c**. Local and Global History Ready Buffers are initialized to 0. The newly declared adaptive tag caches are registered to sdb by calling **cache_reg_stats** in **sim-cache.c** file. Hence, the statistics of newly declared adaptive tag caches are appended to statistics report displayed at the end of simulation.

Local and global history ready buffers are once left shifted with bit '1' at the beginning of **cache_access** function defined in **cache.c**. When it is time to check whether history buffers are ready or not, it is only checked whether there are any 0 bits in history ready buffers. If so, it means, not all bits are filled up, so history buffers are not ready. If all history ready buffers are filled with 1's, voting algorithm uses history information accumulated in history buffers.

The rest of the changes for the algorithms were made in **cache.c** file. In the function **cache_access**, there are more than one exit points of the function. First of all, all exit points are jumped to a exit label, and exit of this function is ensured to be handled at just one point at the very end of the function. Next, first 2 runs of the same instruction is performed with user defined policy and LFU. When it is time for the actual adaptive run, first of all, it is checked whether history buffers are ready to use. If not, adaptive replacement policy depends on just the miss counts just like the original paper. However, if the history buffers are ready, most voted policy is used as the adaptive policy.

IV. DISCUSSION

To show our results, we ran our experiments using two different configurations as follows

1- dl1 and il1: 16KB cache, 64 sets, 64Bytes blocks , 4 way associative, dl2: 512KB cache, 64Bytes blocks, 8 way associative

2- dl1 and il1: 16K cache, 64Bytes blocks, 2 way associative, dl2: 512K cache size, 64Bytes blocks, 4 way associative

The first configuration is similar to the one used on the original paper and we used it to compare our work with the original one while in the second configuration, we changed the associativity (in the first configuration) and kept the cache size with the block size the same as the first one, to measure if the associativity has any effect on the results.

We perform our experiments on five benchmarks (bzip2, gcc, twolf, vpr and mcf). Besides, we used two metrics to show results: the number of misses per thousand instructions and elapsed time during execution for each policy.

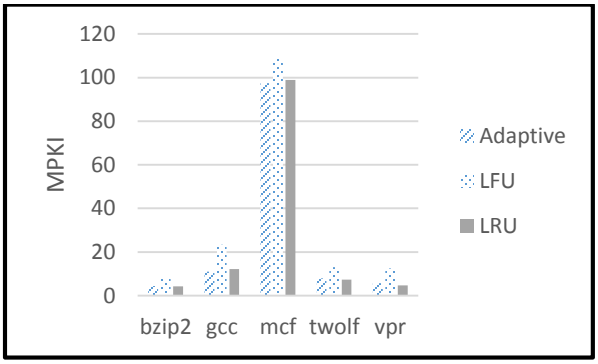

Figure 3: measures MPKI for the second level cache over bzip2, gcc, mcf, twolf and vpr benchmarks for the three policies (adaptive, LFU and LRU) using the first configration.

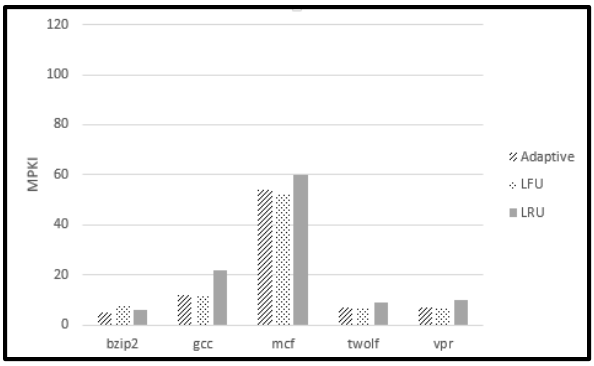

Figure 4: Paper results. The figure measures MPKI for the second level cache over bzip2, gcc, mcf, twolf and vpr benchmarks for the three policies (adaptive, LFU and LRU) using the first configration.

Comparing the results from Figure 3 with the ones in Figure 4, we can observe that the MPKI values in our graph is different from the one in the paper. We expected that since we used different simulator than the one on the original paper which cause a variation on the results. Despite this fact, we can still notice that our results are similar to the functionality of the original paper in term of having our adaptive cache performing either nearly or the same to the better policy.

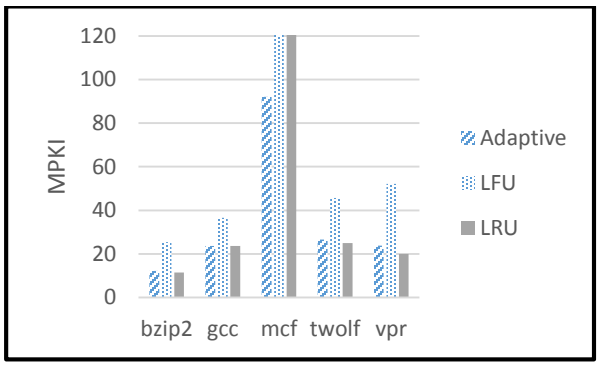

Figure 5: measures MPKI for the first level cache over bzip2, gcc, mcf, twolf and vpr benchmarks for the three policies (adaptive, LFU and LRU) using the first configration.

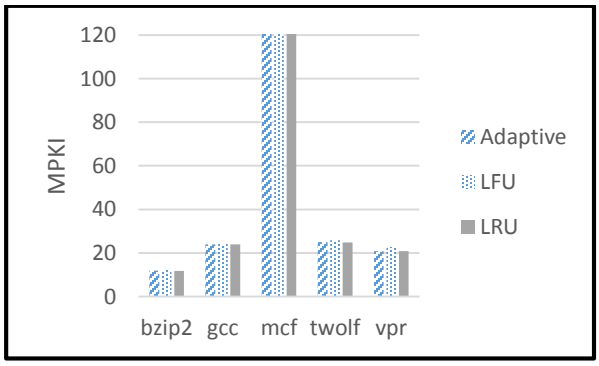

Figure 6: measures MPKI for the first level cache over bzip2, gcc, mcf, twolf and vpr benchmarks for the three policies (adaptive, LFU and LRU) using the second configration.

As we mentioned earlier, during our experiments, we tried different implementation of the adaptive caches, i.e., we applied adaptive cache for the first level cache only, on both levels of the cache and for only level two cache. We found that applying the adaptive cache to both levels gives the best performance. Thus Figures 5 and 6 measure MPKI using configurations one and two respectively for the first level cache. In both figures, the adaptive policy is preforming as good as the best policy.

Figure 7 shows the MPKI for the level 2 cache using the second configuration. By comparing Figure 3 with Figure 7, one can see that there is no obvious influence in modifying the associativity of the caches since both of them are almost preforming the same. However adaptive policy still perform better than the other two policies in Figure 7.

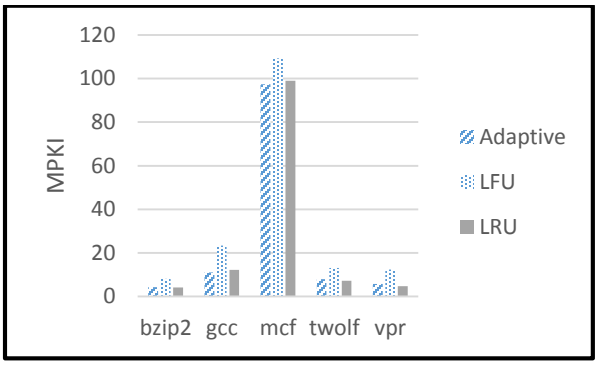

Figure7: measures MPKI for the second level cache over bzip2, gcc, mcf, twolf and vpr benchmarks for the three policies (adaptive, LFU and LRU) using the second configration..

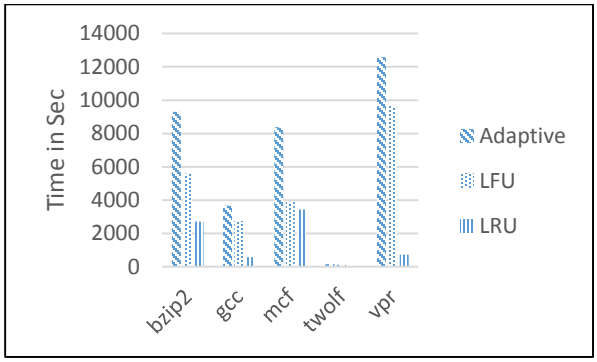

Figure 8: The graph depicts the stimulation time of the first configuration.

Figure 8 shows the simulation time measured by seconds for the first configuration (the one in the original paper). The simulation time for adaptive cache is more than LFU and LRU. This is acceptable since the adaptive cache waits for both LRU and LFU to finish their executions before it decides which policy it needs to adapt. However, the adaptive cache simulation time still reasonable since it is only the sum of the times of other policies which is expected.

It is worth to mention, that all the proposed figures are only for one run and not an average of multiple runs, since during our implementation we stopped the random seeds, thus all the runs gives similar results.

BACKGROUND

There are many previous works on improving the cache performance, especially on using cache adaptive policy replacement [2]. The work that is the closest to our project is a work done by Subramanian et al. [3]. This work used the count of misses on deciding the cache policy replacement and it determines the hardware needs to improve the cache. We inspired our work based on this project and we also used the number of misses as one of the inputs to our algorithm as described earlier. This same work presents an adaptive policy that is a result of a combination of two other policies in such a way that the resulting policy do not perform worse than the original policies. Also, it is possible to switch online between policies based on the previous performance of those policies. This can be considered as a solution since most of the well-known replacement policies have limitations and fails in particular situations, for example, LRU fails to handle loops that are larger than the memory size. Other works on caching include increasing the associativity of the cache which will result in decreasing the number of misses. Such work is a work done by Hallnor and Reinhardt [4].

Our work differs from the previous approaches in using the idea of branch prediction in such a way that enables predicting and choosing the best policy among the available policies.

V. CONCLUSION

Overall, our project was divided equally among the three members. We all contributed to the code, running benchmarks, creating graphs, and generating the PowerPoint and the final document. During the project, we continuously communicated with one another through varies emails and texts in order to confirm our results and completion in the tasks.

Based on our results and research, we were able to conclude that the adaptive cache replacement policy is guaranteed a good performance with small percentage of error. We also show that it is worth to use adaptive cache since the simulation time remain linear. In addition, it is always guaranteed a good performance which is clear from the results.

For future research, we would like to implement other replacement algorithms, such as Pseudo LRU and Segmented LRU. In this way, the adaptively will be able to decide on a replacement algorithm for more than two replacement policies.

REFERENCES

[1] "About SimpleScalar LLC." SimpleScalar LLC., 2004. Web. 20 Apr. 2015.

[2] Smaragdakis, Yannis. "General adaptive replacement policies." Proceedings of the 4th international symposium on Memory management. ACM, 2004.

[3] Subramanian, Ranjith, Yannis Smaragdakis, and Gabriel H. Loh. "Adaptive caches: Effective shaping of cache behavior to workloads." Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2006.

[4] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In Proceedings of the 27th International Symposium on Computer Architecture, Vancouver, Canada, June 2000.