CS 3853 Computer Architecture

Project Report

May 2, 2019 7:30 p.m.

Team 03

Group Members

1. Richard Azille

2. Keegan Knisely

3. Sabrina Mosher

By signing this report, I affirm that I know and agree with the contents.

Signatures:

1. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Objectives:**

1. Comprehend how a cache implementation works
2. Determine the performance benefits based on cache configuration
3. Learn to work in a small group
   1. Learned the benefits of working with efficient team members that each carried equal weight over the course of the project.
   2. Divided tasks fairly and equally and played off each other’s strengths and specific areas of knowledge/expertise

**Algorithm**

For every valid combination of cache size, block size, and associativity, our simulator would construct a virtual cache based on the specifications. The primary cache structure (implemented via a typedef struct, “Cache”) was used to record all pertinent cache information and characteristics. The algorithm used to actually configure the cache would first calculate additional cache values based on the initial command line specifications and then dynamically allocate memory to support an array of Index structs based on the number of indexes calculated, each with a dynamically allocated array of Block structs based on the cache associativity.

Our algorithm for determining a cache hit was relatively simple. For each address read in from the trace file, we parsed the address into its tag, index, and offset values first. Then we used the index value to access the corresponding cache row. For each block in the index’s array of blocks, if the block was valid and the tag matched that of the current address being handled, we declared a hit. Otherwise, if the tag could not be found in the index, a miss was recorded, and a replacement would be performed according to the policy in place.

In order to implement our round robin replacement algorithm, each index of the cache tracks a “replacement block index” which is the index of the next block to be replaced in that specific index’s array of blocks. With each replacement, this value is incremented by one within the range of [0, associativity - 1] with modulo math in place to prevent an index outside of the valid range of the row (ex. Associativity = 4, replacement block index = 3; after replacement, replacement block index = 0).

Our random replacement algorithm simply generates a random value within the range of [0, associativity - 1] and replaces the block at the randomly generated index value.

Our LRU algorithm was achieved by utilizing timestamps and associating each block with the time that it was last accessed. This allowed for a simple algorithm in which we could quickly identify the block with the highest time value (least recently used) and designate it to be replaced. After replacing the block with the new tag value, the block’s clock time is then updated to the current time, making it the most recently used and the least likely to be replaced in that specific index.

**Analysis**

**Technical Issues**

The most challenging technical aspect of completing this project was coming up with an effective design for the cache itself. In order to accomplish this task, we first had to identify all of the requirements that a virtual cache implementation would need to satisfy. One such requirement was the ability of the cache to maintain a collection of indexes (rows) each with up to 16 cache blocks depending on the cache’s associativity. Also, the cache needed to be capable of taking on variable configurations based on several unique combinations of cache size, block size, and associativity. After identifying each of the critical requirements of the virtual cache, we found our ideal solution by creating three unique typedef structs to support the implementation: Cache, Index, and Block. With these data structures in place, we were able to dynamically allocate the components to build any valid cache configuration that our simulator could be tasked with.

**Group Member Contributions**

Each member of our team played a consistent and active role in contributing to the progress of this project over the course of the semester. Overall, the responsibilities of coding, testing, and troubleshooting the cache simulator were shared equally among all members. Specific tasks completed by each team member are listed below.

1. Richard
   1. Implementation of cache data structures and cache configuration and deconstruction functions
   2. Implementation of round robin and random replacement algorithms
   3. Fair contributions in final project report
2. Keegan
   1. Implementation of command line input processing and validation
   2. Implementation of address to tag, index, and offset logic
   3. Fair contributions in final project report
3. Sabrina
   1. Implementation of file-tracing and address-parsing logic
   2. Implementation of LRU algorithm
   3. Developed script for plotting simulation outputs and compiled all results into charts and graphs for post-experiment analysis
   4. Fair contributions in final project report

**Group Issues and Resolutions**

None.

**Conclusion**

Overall, this project was a valuable hands-on effort that provided a greater understanding of cache implementations, operations, and performance. The work load was very manageable and not overwhelming, especially with a solid team to share the responsibilities with. One of the most enjoyable aspects of this project was that it forced us to to figure out how to convert our preexisting knowledge of caches into actual implementation and code to produce an accurate simulator.