Readme.pdf

Project Info:

Author: Mike Swift <theycallmeswift@gmail.com>

Date Created: April 28th, 2011

Date Modified: May 1st, 2011

Description:

This project simulates a write through or a write back direct mapped cache in C. The program takes in a write method and a trace file and computes the number of cache hits and misses as well as the number of main memory reads and writes. More detailed information about the project can be found within the pa3.pdf file.

Structure & Compiling:

The project is structured as follows:

```
bin/
src/
      sim.c
      sim.h
traces/
      trace0.txt
      trace1.txt
      trace2.txt
      trace3.txt
Makefile
pa3.pdf
readme.md
readme.pdf
results.txt
testplan.txt
```

The makefile contains two rules: sim and clean. Calling “make” will create a new sim executable in the bin/ directory. All *.o files are removed automatically during this process. Clean will remove any files in the bin/ directory.

```
Example calls:
      ./bin/sim wt traces/trace0.txt
      ./bin/sim wb traces/trace3.txt
      ...etc...
```

Design & Implementation:

The main algorithm was the following:

1. Validate inputs
2. Open the trace file for reading
3. Create a new cache object
4. Read a line from the file
5. Parse the line and read or write accordingly
6. If the line is "#eof" continue, otherwise go back to step 4
7. Print the results
8. Destroy the cache object
9. Close the file

The functions were separated into three main groups: the main function, cache functions, and utility functions. The main function executed the aforementioned algorithm. The utility functions were primarily related to converting the hexadecimal memory addresses to various binary and decimal equivalents.

There were four main cache functions: 1) createCache, 2) destroyCache, 3) readFromCache, and 4) writeToCache. The create and destroy functions are fairly straightforward. The reading and writing function algorithms went as follows:

Read Algorithm:

1. Validate the inputs
2. Convert the hexadecimal address to a parsable binary format
3. From the formatted binary string, extract the tag and index
4. Convert the index to an integer corresponding to a slot in the array
5. If the current block is valid and has the same tag as the one we are searching for, increment the cache hits, and then go to step 8
6. Otherwise, increment the misses and reads values.
7. If the write policy is write back and the block is marked as dirty, increment the writes and mark the block as clean
8. Set the block as valid and store the tag within the block

Write Algorithm:

1. Validate 1. Validate the inputs
2. Convert the hexadecimal address to a parsable binary format

3. From the formatted binary string, extract the tag and index

4. Convert the index to an integer corresponding to a slot in the array

5. If the current block is valid and has the same tag as the one we are searching for, increment the cache hits, and mark the block as dirty. In addition, if the write policy is write through, increment the writes. Go to step 8.

6. Increment the misses and the reads.

7. If the write policy is write back and the block is dirty, increment the writes

8. Set the block to valid, dirty, and store the tag in it.