Lab #1 Report

Christopher Daffron

Eric Reinsmidt

2014.10.26



# Cache Simulator

## Overview

For development of our cache simulator we decided to use Python. Neither of us has a large amount of experience with Python, so we thought it would be a good opportunity to try out a new language while working on the simulator. Python adds some nice functionality such as the **argparse** library. This library allows us to easily enforce the use of command-line arguments, as well as prompting the user with a help menu if an improper combination of arguments is passed to the program.

While we could have provided default values for the arguments required by the simulator, we opted to make all of the arguments explicit. From the command-line, a user can run the cache simulator with varying levels of cache, ranging from L1 up to L3. In addition, each level of cache has its own arguments to define how it will function. These arguments include the cache’s block size, number of lines in the cache, the cache’s associativity, as well as the hit time. For the purposes of our simulator, miss time and hit time are equal. In addition to the arguments passed for each individual cache, there are four more arguments that must be user-supplied. These arguments hold true for all levels of cache and include the word size used, write time to main memory, the cache write policy, and a filename of a cache trace file to be used. In addition to these arguments, there is a non-required debug flag that can be set from the command line to provide more verbose output from the program.

We used a data structure to hold each of the entries in the cache. Each instance of a cache entry has four parameters. Three of these parameters; valid bit, tag, and last access are needed for both of the write policies used, write-through and write-back. In addition the fourth parameter, dirty bit, is needed only for the write-back write policy. At the start of a cache simulation, the program parses the user-supplied arguments, error checks the word size, block size, and number of lines to make sure all are a power of two. In addition variables for each level of cache simulated are initialized. These include, but are not limited to, the number of sets in the cache, set size, index, cache start block, cache end block and block offset. In addition, each level of cache is initialized.

Our cache implementation is fairly linear in its approach. The input file is parsed line by line to retrieve the address and the mode to use, either read or write. First, the L1 cache is looped through and the valid bit of each block is checked. When the valid bit of a block is set to true, its tag is compared against the address from the parsed line. If there is a match and the request is a read, then a cache hit occurs and the access time is updated. If there is a match and the request is a write the access time is updated and the dirty bit is set if the write policy is write-back.

If the address is not found in L1, then the same procedure takes place in L2. In addition, if the address was found in L1 and the write policy is set to write-through, then the L2 cache is updated as needed. The same general program flow takes place in L3 if the address was not found in L1 or L2. Next the cache takes a main memory write penalty if the request is a write and the write policy is write-through.

After checking for, and not finding, the address in any level of cache the program places the address in cache starting with L3. The program loops through the range of L3 set addresses and if an empty block is found, which is denoted by the valid bit being set to zero, the address is placed in the cache. The valid bit and tag are set, last access is updated, and the dirty bit is cleared. If there are no blocks that are empty, the program then loops through the appropriate set addresses and finds the block with the oldest access time and places the address in that least recently used (LRU) block while also updating the access time and setting the valid bit and tag. If the write policy is write-back, and the replaced address had its dirty bit set, then there is a write to main memory. After placing the address in the L3 cache, the program then continues by doing the same in the lower levels of cache. The logic for L2 and L1 are slightly more complicated when the write policy is write-back. When this is the write-policy, replacing using LRU also requires a write to higher levels of cache when the cache line being replaced has the dirty bit set.

Finally, the cache simulator outputs all of the appropriate information about the simulation. Hit and miss numbers for each level of simulated cache are output as well as the hit and miss ratio and the average memory access time (AMAT). If the user has set the aforementioned debug flag when running the simulation, much more information is displayed for each individual cache request.

## Usage

**usage: cacheSim.py [-h] --L1 block\_size num\_lines associativity hit\_time**

**[--L2 block\_size num\_lines associativity hit\_time]**

**[--L3 block\_size num\_lines associativity hit\_time]**

**[--debug] (--WB | --WT)**

**word\_size write\_time file\_name**

**Cache simulator arguments**

**positional arguments:**

**word\_size Word size used for cache**

**write\_time Write time to main memory**

**file\_name Path to trace file**

**optional arguments:**

**-h, --help show this help message and exit**

**--L1 block\_size num\_lines associativity hit\_time**

**L1 cache parameters**

**--L2 block\_size num\_lines associativity hit\_time**

**L2 cache parameters**

**--L3 block\_size num\_lines associativity hit\_time**

**L3 cache parameters**

**--debug Set verbose output**

**--WB Set write-back policy**

**--WT Set write-through policy**

## Testing

Since the cache simulator has a lot of flexibility in terms of the parameters that a user can set, it was necessary to test the simulator with varying trace files. We created test files programmatically using Python. The program used to create the test files, makeTestCases.py, is included in the tar file that included this document. The created test files contain anywhere from 65,535 to 1,048,575 cache requests. The following trace files were created:

* spatialLocality.txt
  + Read/Write pseudo-randomized
  + Continuous addresses from 0 to 1,048,575
* temporalLocality.txt
  + Read/Write pseudo-randomized
  + Loop of 256 addresses repeated 256 times
* disparate.txt
  + Read/Write pseudo-randomized
  + 1,048,575 random values chosen from 0 to 4,294,967,295

The first two trace files were created to test out simulator performance when there is a high level of locality, while the final trace file was created to test simulator performance with a disparate data set.

Each of the trace files was tested using varying amounts of associativity and levels of cache. Associativity levels of direct, 2-way, and 4-way set associative were used with either all three levels of cache or just a single level. Each of these tests was run in both write-through and write-back write policies. Block size and word size were kept at 16 and 4 bytes respectively as this mimics the 64KB cache lines found in modern caches like that used in a core i7 processor. The same write time to main memory and hit/miss times were used for all tests. Full results can be seen in the included testing folder.

## Conclusion

Overall this was an interesting lab assignment. Some of the insights we garnered from the lab came from the actual coding of the simulator itself. While not germane to the actual assignment, getting to try out a new language was worthwhile. One of the most important things that we can take away from this assignment is that having a consistent block size throughout multiple levels of cache is important. Our design for our simulator was dictated by a desire to have an inclusive multilevel cache. After beginning coding it became apparent that having differing block sizes in the various levels of the cache, in combination with a LRU replacement policy, could create a situation where a block could be removed from L2 before it was removed from L1. This would have caused an inconsistency in the addresses that were stored in cache. Another interesting idea that we had to consider was that when replacing a value when a LRU replacement policy, it was not straightforward when determining the maximum of the range of possible addresses that are in a particular cache line that is the most recently used. To get around this problem we used the tag of the old address, added the appropriate offsets to its length, and then used a bitwise OR to place a ‘1’ at the least significant bit of the tag and then subtracted one from this value to recreate our maximum address.

In addition to the coding aspect of the assignment, there were some interesting concepts that we gleaned from the testing results. The disparate data set we used for testing was pathological in that there was little to no locality present at all in the cache requests. However, it was interesting to see that with just such a data set a multilevel cache is actually a detriment to the AMAT. The overhead associated with having to search through the L2 and L3 cache simply added time without producing any hits. This shows that for certain use cases, a single level cache could actually be advantageous. In our temporal locality test case, we also had slight improvements when using a single level of cache compared with a multilevel cache. Finally, in our spatial locality test case, depending on whether a write-back or write-through policy is chosen, using a single level of cache can either help or hurt the AMAT. When using write-back policy and a single level of cache, the AMAT can improve over a multilevel cache if L2 and L3 only have misses. At the same time however, with a write-through policy a multilevel cache improves the AMAT when compared to a single level. These examples show that choosing a multilevel cache is not always the answer.