Comp Arch Lab 5

Due May 2nd, 2019

1. Introduction

Over the past few decades, processor speeds have doubled every eighteen months. However, memory speeds have not enjoyed the same stellar increases in performance. Therefore, modern computers have a hierarchy of memory in which smaller memories have faster access. In this assignment, you will simulate a two-stage hierarchy (data cache and main memory) in software and analyze its effectiveness.

2. Getting Started

Download the code files (cachesim.zip) available on the BlackBoard. Provided to you will be C and C++ header code (`memory.h/memory.hh`), which defines the classes and constants that you must use in your implementation, and six test programs (either .c or .cpp) for your project. The provided code contains template code and test programs in either C or C++. No other programming languages like Java, Perl, Python, are permitted. Depending on your preference, implement your code in one of template codes. The compilation scripts are also provided. Use `compile.sh` for C and `compile-cpp.sh` for C++. Note that the compilation scripts use gcc and g++.

Ultimately, you will provide mechanisms for reading and writing data from and to memory. To the user, all of the details should be transparent, but there will be some transactions occurring with the hardware to get the data to and from the processor. Your memory hierarchy will consist of the data cache and the main memory (MM). The cache can be implemented however you like.

3. Details

For this assignment, you will only deal with data items that are integers. Thus, only integer words will be loaded and stored. No other data types will be used. A block is 4 words, and the main memory has 512 blocks. Your cache will hold 8 blocks. Initially, the cache memory is empty. In this assignment, we only consider the behavior of memory hierarchies, so you do not need to worry about actual values stored or loaded. You only need to consider the address of each data access. The main function you need to implement is `getData()` and `putData()`. These methods will be called in the six test programs for matrix multiplications.

On a cache miss, the requested word, along with three other words (the block of MM), are loaded into the cache.

You should maintain a counter for the number of cache misses (`NumOfMisses`), the miss-rate, and a clock cycle that will keep track of the time occurred through the execution of the user program. When the user program ends, these statistics are displayed on the screen. Do not take into account the time required for fetching instructions, executing instructions, and using data from registers (represented as local data in the user program). Instead, just increment the clock by 2 for a cache access and 100 for a memory access.

As for the cache designs, use direct mapping, two way associative and fully associative with Least Recently Used (LRU) replacement policy. That is, write the code to simulate these three cache designs and run them against the test programs as described below.

As for the writing policies, use write-through scheme; on a cache hit, the data should be written to the cache block and the MM block. On a cache miss, write the data to MM block, and then load the MM block into the cache.

This assignment has no requirements as far as how you create this hierarchy other than those listed above. You may use any data structures, design, and any standard libraries granted the code will compile and run on Linux. Because of the flexibility of the assignment, the programs should be very different among each student's implementation. Therefore it is imperative that you give a solid description regarding how you designed and implemented the project.

4. Testing

Run the test programs from the BlackBoard against all versions of your simulation program (direct, fully associative, two way associative) and print out the results. Some of the test programs will attempt to do the same thing, but will use memory differently ("smart" vs. "naive" test programs) and thus should have different performance results. Other programs test correctness of loading and storing values. If you aren't getting the expected results, prepare to explain why you think that is, and if you know why that is, correct the problem.

5. Submission (single source code per group, and report should be handed in individually)

- **[10pts]** A detail explanation of your designs and implementations (maximum 10 pages, 11pt font size, single spaced, 1 inch margin) which should include the following:
    - The data structures you used to implement different cache designs, mechanisms you used to calculate the cache index, tag, word offset.
    - The status bits you used in your caches, and the technique that you employed to implement the LRU replacement policy.
- **[50pts]** The code for your memory library. Your programs must be well commented.
- **[10pts]** The print outs from running the test programs.
- **[10pts]** A print out of the address of the final 32 words stored in the cache after running your simulation programs on test programs "`naive6`" and "`smart6`". The "`memory.h`" includes the definition of the print routine (`showCacheAddress`) and you are required to implement the method according to your cache structure.
- **[10pts]** A table for each cache design with the following look:

	cache design (e.g., direct mapped)					
	naive program			smart program		
	# of cycle	# of misses	miss rate	# of cycle	# of misses	miss rate
N=6						
N=16						
N=25						

The tables should be generated from the results obtained after running your simulation programs on the last six test files:

- **[10pts]** Performance analysis, which should address the following:
    - Overall effectiveness of the memory hierarchy using the information gathered from running test programs. In other words, discuss the performance improvement for each of cache designs.
    - Comparisons of the performance of the different cache designs.
    - The reason for the “smart” programs having better performance than their “naive” counterparts. Hint: The role of smart programming that exploits memory hierarchy through better locality.
    - Compare and contrast the effectiveness of the “smart” vs. “naive” test programs for the three cache designs (direct mapping, fully associative and two way associative).
    - Any other interesting findings.

6. Instruction:
For compiling C code, simply enter "make". For C++ code, enter "make -f Makefile.cpp".