Machine Organization and Assembly Language Programming

# Problem Set #7

Due: Friday May 30th

In this assignment you will write in a language of your choice (C, C++, Java) the skeleton of a trace-driven simulator for assessing the performance of a cache. You should do this assignment by yourself.

It is recommended that you use Java so that it will be easier for Tapan to compile/run your programs. If you use C or C++, and even in the case of Java, limit the use of any external GUI libraries or other helper libraries. For anything non-standard, include detailed explanations in the README file.

Trace-driven simulation is a widely used technique to assess the performance of the components of the memory hierarchy. A sample trace file (see below for details) is on the net. In broad terms, the simulator works as follows

**Input**

- A trace, i.e., a string, of memory references.
- One or more cache descriptions (I-cache, D-cache, cache hierarchy) with size, associativity, block size, replacement algorithm.
- A write policy.
- Access times of the various components of the memory hierarchy.

**Output**

- A set of statistics, e.g. hit ratio, or more precisely read hit ratio, write hit ratio, average memory access time etc.
- Cycles spent waiting at the various levels of the memory hierarchy etc.

**Algorithm**

- Process each memory reference in turn. Decompose the address in (tag, index, displacement) components
- Check if the memory reference hits in the appropriate cache (note that you don't bring data in the simulated cache; you only simulate the presence/absence of particular blocks)
- Take appropriate actions. In case of a hit, record statistics, maybe turn on some valid/dirty bits etc. In case of a miss, bring in the missing block, replace an old one, record statistics etc.

Your assignment is to write a cache simulator (you can do exercise 7.7 in your book by hand to warm you up) with the following specifications.

- The trace will consist of a string of data memory references (we will simulate only data caches) with the following format:$< operation > < address >$ where *operation* is "1" for a read and "2" for a write, and *address* is a 32-bit byte address.

  The trace ends with the pair 0 0.

  There is a link on the net for such a trace.

- Because the trace is small, we'll have a ridiculously small cache so that you don't have only compulsory misses. The caches that you'll have to consider are:

    1. Cache 1: 128 bytes, direct-mapped, line size 8 bytes. It uses a write-back, write-allocate policy.
    2. Cache 2: 128 bytes, 2-way set-associative, line size 8 bytes. It uses a write-back, write-allocate policy and an LRU replacement algorithm.
    3. Cache 3: Cache 1 backed up by a 2-entry victim cache using an LRU replacement algorithm.

The output statistics that you should generate are:

- The number of data references that you processed.
- For each of the 3 cache configurations, the number of hits in the cache (in the case of Cache 3, count separately the number of hits in the cache and the number of hits in the victim cache)
- For each of the 3 cache configurations, the number of blocks you needed to write back.
- Assuming that a hit in the cache takes 1 cycle, a hit in the victim cache takes 2 cycles, and a miss is resolved in 50 cycles, what are the average memory access times for the 3 configurations. (don't draw any definite conclusion on the advantages/drawbacks of one configuration based on the results of this single minuscule and totally unrepresentative experiment!)

You should **turnin** a README file indicating briefly how to run your program and how to read the output. In case you are aware of bugs and did not have time to correct them, indicate it in the README file. Send e-mail to Tapan (tapan@cs) with subject 378ASS7 with your file called LASTNAME.xyz (xyz depends on the language you used).