Lab #4: Cache Simulation Experiments  
due Wednesday, December 12

Important note: These are not group labs. Everyone does this lab SEPARATELY.

For this lab assignment, you will write a configurable cache simulator (in C, C++, or Java). Your cache simulator will read an address trace (a chronological list of memory addresses referenced), simulate the cache, generate cache hit and miss data, and calculate the execution time for the executing program. The address trace has been generated by a simulator executing a real program. Your cache simulator is not the end product of this lab, but a tool you will use to complete it. In this lab, you will experiment with various cache configurations and make conclusions about the optimal cache organization for this set of programs.

Using the address trace:
An address trace is simply a list of addresses produced by a program running on a processor. These are the addresses resulting from load and store instructions in the code as it is executed. Some address traces would include both instruction fetch addresses and data (load and store) addresses, but you will be simulating only a data cache, so these traces only have data addresses. These traces were generated by a simulator of a RISC processor running three programs, art, crafty, and eon from the SPEC benchmarks. The files are art.trace.gz, crafty.trace.gz, and eon.trace.gz. They are all compressed with gzip, but should be readable by a Windows decompress utility. If you are running on a Unix machine, you do not need to ever store the traces uncompressed. Instead, use the following commands to generate the trace and pipe it through your cache simulator, like so:

gunzip –c art.trace.gz | cachesim [cachesim args]

Because your workload is three programs, you will run three simulations for each architecture you simulate, and then combine the results in some meaningful way. The simulator arguments should be something like this, so we can run it:

cachesim -s 32 -a 4 -l 32 -mp 30

would simulate a 32 KB, 4-way set-associative cache with 32-byte lines, and a 30-cycle miss penalty.

If you do not run it on the local Unix system, you need to give us specific instructions on how to run your code in a Unix, Windows, or mac environment, so we can reproduce it if necessary. Even if you use an environment that does not support command-line arguments, your code should support any reasonable values for cache size, associativity, etc.

Format of the address trace:
All lines of the address trace are of the format:

   # LS ADDRESS IC

where LS is a 0 for a load and 1 for a store, ADDRESS is an 8-character hexadecimal number, and IC is the number of instructions executed between the previous memory access and this one (including the load or store instruction itself). There is a single space between each field. The instruction count information will be used to calculate execution time (or at least cycle count). A sample address trace starts out like this:

   # 0 7fffed80 1
   # 0 10010000 10
   # 0 10010060 3
   # 1 10010030 4
   # 0 10010040 6
   # 0 10010064 3
   # 1 10010034 4

You should assume no accesses address multiple cache lines (e.g., assume all accesses are for 32 bits or less).

The simulator output:
Your program should produce miss rates for all accesses, miss rates for loads only, and execution time for the program, in cycles. It should also show total CPI, and average memory access time (cycles per access, assuming 0 cycles for a hit and miss penalty for a miss). For execution time, assume the following: All
instructions (except loads) take one cycle. A load takes one cycle plus the miss penalty. The miss penalty is 0 cycles for a cache hit and 30 cycles for a cache miss (unless specified otherwise). Loads or stores each result in a stall for miss-penalty cycles. You will simulate a write-allocate cache. In the trace shown, the first 31 instructions should take 151 cycles, assuming four cache misses and 3 cache hits for the 5 loads and 2 stores, and a 30-cycle miss penalty. You will be modeling a write-back cache, but we assume the write of a dirty line takes place mostly in the background. So we assume an extra 2-cycle delay to write a dirty line to a write buffer. So, using the parameters from above, a load or store miss that would evict a clean line takes 32 cycles, and if evicting a dirty line takes 32 cycles. Cache replacement policy is always LRU for associative caches. If useful, assume a 2 GHz processor. Each trace contains the memory accesses of just over 5 million instructions. Your simulations should process all of them.

The cache:
The baseline cache configuration will be 16-byte line size, direct-mapped, 16 KB cache size, write-back, and write-allocate. You will re-evaluate these parameters one at a time, in the following order. In each case, choose a best value for each parameter, then use that for all subsequent analyses.

A. Look at 16 KB, 32 KB, and 128 KB cache sizes. Larger caches take longer to access, so assume that the 32 KB cache requires a 5% longer cycle time, and the 128 KB 15% longer. Choose the best size/cycle time combination and proceed to the next step.

B. Look at cache associativity of direct-mapped, 2-way set-associative, and 8-way set-associative. Assume that 2-way associative adds 5% to the cycle time, and 8-way adds 10%. Choose the best associativity and cycle time, and proceed.

C. Look at cache line sizes of 16, 32, and 64 bytes. Assume that it takes two extra cycles to load 32 bytes into the cache, and 6 extra cycles to load 64 bytes. (i.e., raise the miss penalty accordingly). Choose the best size and miss penalty and proceed.

You will turn in a written lab report, but also mail your code to leporter@cs.ucsd.edu. You will find the traces and a skeleton C program on the class webpage for download.

Questions for lab 4:
1. Show your results and design decisions for each of the three configuration parameters above.
2. Turn in your cache simulator code with sample output.
3. Is cache miss rate a good indicator of performance? In what cases did the option with the lowest miss rate not have the lowest execution time? Why?
4. Were results uniform across the three programs? In what cases did different programs give different conclusions? Speculate as to why that may have been true.
5. What was the speedup of your final design over the default?
6. Your report will be turned in on paper, but we also want your code for verification.

Hints:
- Think about how to intelligently debug and test your program. Running immediately on the entire input gives you little insight on whether it is working (unless it is way off).
- Speed matters. These simulations should take a couple minutes (actually, much less) on an unloaded lab machine. If it is taking much more than that, do yourself a favor and think about what you are doing inefficiently.
- Give execution time in some reasonable and consistent form.