Lab #4: Cache Simulation Experiments
due Friday, March 17

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 or Java). Your cache simulator will read an address trace (a chronological list of memory addresses referenced by a program), simulate the cache, generate cache hit and miss data, and calculate the execution time for the executing program. The address traces have been generated by a CPU simulator executing real programs, and saved into two files. 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 traces:
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 this trace only has data addresses. These traces were generated by a simulator of a RISC processor running two programs, m88ksim and compress, from the SPEC benchmarks. The files are m88.trace.gz and comp.trace.gz, and are both compressed with gzip. They are large, so you don’t want to copy them. Instead, use the following commands to generate the trace and pipe it through your cache simulator, like so:

    gunzip –c ~cs141w/traces/m88.trace | cachesim [cachesim args]
    gunzip –c ~cs141w/traces/comp.trace | cachesim [cachesim args]

Because your workload is two programs, you will run two 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 16 –a 4 –l 32 –lmp 18 –smp 18

would simulate a 16 KB, 4-way set-associative cache with 32-byte lines, and 18-cycle miss penalties for both loads and stores.

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. It should also show CPI, MCPI, and AMAT (cycles per access). 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 18 cycles for a cache miss (unless specified otherwise). Stores can hit or miss in the cache and also incur the same miss penalty; that is, we will assume that the processor also stalls for store misses. For this lab, we only simulate the data cache; thus, we assume a 0% instruction cache miss rate (i.e., instruction addresses are not part of the trace). In the trace shown, the first 18 instructions should take 72 cycles, assuming three cache misses and a cache hit, and an 18-cycle miss penalty. Your simulation should process all of the 1 million references in each trace.

The cache:
The baseline cache configuration will be 32-byte line (block) size, direct-mapped, 16 KB cache size, write-through and write-around. (A write-around cache, on a write miss, will not make a spot for the line in the cache, but store it to memory. Although the cache is write-through, you do not stall the processor on a store hit.) 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 cache line sizes of 32, 64, and 128 bytes. Assume that it takes two extra cycles to load 64 bytes into the cache, and 6 extra cycles to load 128 bytes (i.e., raise the miss penalty accordingly). (What, if anything, should happen to the miss penalty for stores?) Choose the best size and miss penalty and proceed.
B. Look at cache associativity of direct-mapped, 2-way set-associative, and 3-way set-associative. Assume that 2-way associative adds 5% to the cycle time, and 3-way adds 15%. Conversely, look at what happens if 3-way doesn’t change the cycle time, but adds 1 cycle to the hit time and 1 cycle to the miss penalty.
C. Look at 8 KB, 16 KB, and 32 KB cache sizes. Larger caches take longer to access, so assume that the 16KB cache requires a 5% longer cycle time, and the 32 KB 10% longer.
D. Compare the write-around policy with a write-allocate policy, which makes room for the data in the cache, but still stalls until the cache line is filled. For your simulator args, -wall should make it a write-allocate cache.

Questions for lab 4:
1. Show your results and design decisions for each of the four 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 two 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. We also want an electronic version of your simulator. We’ll figure out how to use the turnin program and get back to you via the mailing list!

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 about 4 minutes (actually, less) on an unloaded uAPE machine. If it is taking much more than that, do yourself a favor and think about what you are doing inefficiently.
-Big Hint for those using C: scanf("%c %d %lx %d",&marker,&loadstore,&address,&icount);
-Unix machines are available in AP&M B-402 and EBU2 313.