Lab #4: Cache Simulation Experiments  
**due Friday, November 30**

*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), 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 two programs, art and swim, from the SPEC benchmarks. The files are art.trace.gz and swim.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 ~cs141f/traces/art.trace | cachesim [cachesim args]
gunzip -c ~cs141f/traces/swim.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 32 –a 4 –l 32 –mp 24 –wl
```
would simulate a 32 KB, 4-way set-associative cache with 32-byte lines, a 24-cycle miss penalty, and write-allocate store policy. For the write-around option, specify –wr.

**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 7ffffe80 1
# 0 10010000 10
# 0 10010060 3
# 1 10010030 4
# 0 10010004 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 24 cycles for a cache miss (unless specified otherwise). Stores can hit or miss in the cache (and thus change the contents of the cache for a write-allocate cache), but do not incur a miss penalty, since no data is returned to the CPU. 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 24 instructions should take 103 cycles, assuming three cache misses and 2 cache hits for the 5 loads, and a 24-cycle miss penalty. 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-through and write-allocate. A write-allocate cache, on a write miss, will make a spot for the line in the cache, load it from memory, and it becomes available to read "miss penalty" cycles later. If the processor accesses it (with a load) before that, it stalls until the data is ready. So for a write-allocate cache, you can get a store miss at time T, followed by a load hit at time T + 7. However, the load still must stall until the data becomes available at time $T + \text{miss\_penalty}$ (for bookkeeping, it is still a load hit, but for timing, you must record the stall), so the processor will see a miss penalty on this cache “hit” of $\text{miss\_penalty} – 7$ cycles. Thus, you never stall for a store, but you may stall on a later load to the same cache line. 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.
D. Compare the write allocate policy with a write-around policy, where the processor still does not stall for stores, and a store miss does not change the contents of the cache.

Your cache simulator must run on a Unix system. You can actually do the experiments elsewhere, if you’d like, but you would need to port the code back to Unix for the electronic turnin, in case we need to run it for verification.

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 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, but we will also have electronic turnin 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, 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.
- Give execution time in some reasonable and consistent form. You can make up a cycle time and give ET in seconds. Or you can just give ET in number of cycles. But if the cycle time changes, you need to say something like "machine B took X cycles, which corresponds to Y cycles on machine A because it has a slower clock."
- The 2-way or 8-way caches will use LRU as their replacement policy.
- Big hint for those using C: `scanf("%c %d %lx %d\n", &marker, &loadstore, &address, &icount);`
- UNIX machines are available in AP&M B-402 and EBU2 313.
- A word to the wise: these labs will be turned in electronically, as have programs from previous years – duplicates (even with lots of superficial changes) can be detected.