Lab #4: Cache Simulation Experiments
due Friday, Week 10 by 1pm

Important note: You may work in the same team as before, but you
MUST USE PAIR PROGRAMMING TECHNIQUES.
(more details in lab section)

For this lab assignment, you will write a configurable cache simulator (in C or C++). Your cache simulator will read an address trace (a chronological list of memory addresses referenced), simulate the cache, generate cache hit and miss data. 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 explore their impact on performance on two 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.bz and swim.trace.bz, and are both compressed with bzip. 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:

bunzip -c ~/../public/trace_gcc.bz2 | cachesim [cachesim args]
bunzip -c ~/../public/trace_gzip.bz2 | cachesim [cachesim args]

Because your workload is two programs, you will run two simulations for each architecture you simulate, and then report on the number of misses, number of hits and hit rate for each program. The simulator arguments should be something like this, so we can run it (and PLEASE call it cachesim):
cachesim -s 32 -a 4 -l 32

would simulate a 32 KB (s is for size), 4-way set-associative (a is for associativity) cache with 32-byte lines (l is for line size). The line replacement policy for all cache designs will be least recently used (LRU).

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. **We will be ignoring the instruction count information.** We will be designing a non-functional, simple cache simulator – it will not be able to produce performance results nor will it actually keep track of “real data”. If you wanted to get performance in terms of cycle count – assuming a single cycle processor design – you could use the instruction count numbers. A sample address trace starts out like this:

# 0 7fffed80 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:

- Number of Misses (for all loads and stores)
- Number of Total Accesses (for all loads and stores)
- Hit Rate (for all loads and stores)
- Number of load misses
- Number of load accesses
- Hit rate for loads
- Number of all accesses (lds and sts) that missed in cache, and hit in the victim cache (to see how helpful this was)
- Percent of all hits that came from the victim cache.

Our overall design will feature a write-through, write-around store policy, no cache update on a store miss, and a 4-element FIFO victim cache. Details below:

- On a load, if a value hits in cache, you may need to mark it as “recently used” (unless you are direct mapped). This is a HIT.
- On a load, if a value misses in cache:
  - Check the victim cache, if you hit here it’s still called a HIT. (But remember to keep track of the victim cache hits separately as well.) DO NOT UPDATE CACHE, but move victim element to back of queue (e.g. last “in” to the cache).
  - If it’s not in the victim cache either, it’s a MISS. Grab the data from main memory and figure out where to put it in cache. Put the line you are kicking out (if any) into the victim cache. Update recently used if necessary on the cache line.
- On a store, if a value hits in cache, write both the cache line AND the line in memory (this is a write-through policy). Update recently used. This is a HIT.
- On a store, if a value misses in cache:
  - Check the victim cache. Behave similarly to load, but also write through to memory. This is still a HIT.
  - If it’s not in victim cache, use the write-around policy and just update main memory. This is a MISS.

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).

The cache:
The baseline cache configuration will be 16-byte line size, direct-mapped, 16 KB cache size, with a 4-entry victim cache. You will re-evaluate the first three 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 (in the real world), so we’ll give them a “penalty” on their total hit rate—since we’re only working with a functional simulator, we can’t figure out the real penalty in cycle time. Fill in this table

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Overall Hit Rate gcc</th>
<th>Overall Hit Rate gzip</th>
<th>Calculation</th>
<th>Weighted Overall Hit Rate gcc</th>
<th>Weighted Overall Hit Rate gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>16KB</td>
<td></td>
<td></td>
<td>OHR * 1 =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32KB</td>
<td></td>
<td></td>
<td>OHR * 0.98 =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128KB</td>
<td></td>
<td></td>
<td>OHR * 0.94 =</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pick the design with the best weighted hit rate and continue.

Best Design gcc:_______________________  Best Design gzip:_______________________

I choose to continue pursuing design: __________________________
B. Look at cache associativity of direct-mapped, 2-way set-associative, and 8-way set-associative. Assume that 2-way associative adds a penalty of 2% to the hit rate, and 8-way adds 10%. Choose the best associativity and proceed.

<table>
<thead>
<tr>
<th>Cache Associativity</th>
<th>Overall Hit Rate for gcc</th>
<th>Overall Hit Rate for gzip</th>
<th>Calculation</th>
<th>Weighted Overall Hit Rate for gcc</th>
<th>Weighted Overall Hit Rate for gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td></td>
<td></td>
<td>OHR * 1 =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-way set associative</td>
<td></td>
<td></td>
<td>OHR * 0.98 =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-way set associative</td>
<td></td>
<td></td>
<td>OHR * 0.90 =</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best Design for gcc:_______________________   Best Design for gzip:_______________________

I choose to continue pursuing design: __________________________

C. Look at cache line sizes of 16, 32, and 64 bytes. Again, the cost of increasing the line size is that the time to get data from memory into cache should increase. We’ll approximate with a “weight” on the hit rate. Fill in the table below.

<table>
<thead>
<tr>
<th>Line Size</th>
<th>Overall Hit Rate for gcc</th>
<th>Overall Hit Rate for gzip</th>
<th>Calculation</th>
<th>Weighted Overall Hit Rate for gcc</th>
<th>Weighted Overall Hit Rate for gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td></td>
<td></td>
<td>OHR * 1 =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 bytes</td>
<td></td>
<td></td>
<td>OHR * 0.98 =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 bytes</td>
<td></td>
<td></td>
<td>OHR * 0.96 =</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best Design for gcc:_________________  Best Design for gzip:_________________

I believe the best design to serve these two traces is:______________________________________________________________

What to turn in:

Your cache simulator must run on a Unix system. You can actually do the experiments elsewhere, if you’d like, but you MUST MAKE SURE THAT YOUR CODE RUNS ON IENG6. You should also submit a file (either in Word or PDF) with:

- Name(s) and PID(s) of the team or individual submitting this work
- Copies of the tables from above including the “Best Design” line from each
- Answers written in complete English sentences to the following questions.

TURN IN VIA EMAIL TO BOTH TAS (rayoub@cs.ucsd.edu, kwj002@cs.ucsd.edu) with the SUBJECT LINE: CSE141L YOURNAME YOURPARTNERSNAME

Questions for lab 4:
1. What is the best design overall and how did you come to that decision?
2. What is the impact of the victim cache? Describe in what scenarios you saw it perform well. Pontificate on the impact you think it would have on cycle time and design complexity.
3. What was the most difficult part of the process of designing your cache simulator?
4. Suppose you wanted to modify your cache simulator to be able to mimic performance. For example, you would be able to set a miss penalty for the number of instructions/cycles that would pass before a value was loaded from cache etc. Describe in complete English sentences some of the complications or additions you would have to make to your code. Consider the victim cache, and the (common case) that usually a processor doesn’t stall on a store miss.

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). That is – make yourself a small input test case (like 100 lines).
- 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.
- Big hint for those using C: scanf("%c %d %lx %d\n", &marker, &loadstore, &address, &icount);
- 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. We have the cache simulators from last term as well and reserve the right to use them to compare against your code. Any code with more than a 90% match of source code lines with another program will receive an F.