Project Four: Cache Simulator

Overview:

You will write a cache simulator to evaluate aspects of cache architecture and the effect different cache parameters have on performance. Modeling is often the first step in the design process. This project will give you the opportunity to model cache alternatives.

Your simulator will read address traces from different programs. You will use C as the programming language and start with the template and provided materials on the Github repository.

We will assume that this is for an L1 cache closely associated with the CPU core. We will assume that there is an L2 downstream which mitigates extreme penalties for main memory access. This assumption can be changed with the miss penalty multiplier (see below).

Additionally, you will investigate the implications of LRU design in hardware for set associative caches.

The address traces:

The address traces are representative of load/store instruction address targets. The traces were generated by a simulator of a RISC processor from the SPEC benchmarks. You do not need to know the data associated with the instruction. You do need to know read/write. You will also be given a value indicating the number of instructions executed between previous memory access and the current (including the load or store instruction itself).

Because these are only load/store instructions, you are simulating a data cache; however, the techniques and your C code can be used, obviously for an instruction cache. For an instruction only cache, you would treat all addresses as reads, for example.

Format of the address traces:

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          Note: 0 instructions prior to this plus this one.
# 0 10010000 10         Note: 9 instructions prior to this plus this one.
# 0 10010060 3
```

```
# 1 10010030 4
# 0 10010004 6
# 0 10010064 3
# 1 10010034 4
```

This trace shows a total of 31 instructions.

Simulator "Knobs"

Your simulator should support:

1) Block size: This is the row size in bytes for data stored in the cache. A 16-byte block size has a cache with 16 bytes of data per row in the cache.
    a. Blocks are multiples of 2. Min 16 to max 64
2) Associativity: This is the number of "bedrooms in the house"
    a. Sets are multiples of 2. Min = 1 (Direct mapped), Max = 8
    b. You will use an LRU replacement policy for 2/4/8 way (See additional at end of assignment)
3) Allocate policy
    a. We will assume write allocate and write back. Reads always allocate.
    b. A write allocate cycle is a read miss penalty to fill the cache line (same as the read allocate).
4) Cache size for the data storage bytes
    a. Multiple of 2: Min of 16KB, max of 128KB
    b. This is total cache size. So watch number of rows based on associativity
5) Miss penalty
    a. This is the penalty in clock cycles for a cache miss causing a new cache line fetch to the cache. This will be an integer value equal to the clock rate in GHz x 15 for the base value and add to this the additional due to block size (see below). For example, clock rate 2GHz, MP = 2*15=30. Clock rate 4GHz, MP = 4*15 = 60. This is low if access is to main memory, but we will assume that there is an L2 that mitigates this value.
    b. Dirty line eviction on write back is 2 clock penalty (see below)

Architecture assumptions

Nothing comes for free. So, larger sizes take longer to respond.

Assume a base cycle time (clock rate) of 2Ghz. You can make this a parameter in your program.

All instructions take one cycle. Loads add miss penalties. The miss penalty for a hit is zero.
All misses result in a stall for miss-penalty cycles.
A writeback of a dirty line is mostly hidden by downstream buffering and/or caches. So use an extra 2-cycle delay to write a dirty line to a write buffer. A new load evicting a clean line takes the miss delay (30, for example). A load with a dirty line eviction takes the miss delay + 2 (32, for example).

Cache size implications. Cache sizes slow down the cycle time!

1) Starting with 16KB as base, add 5% cycle time for access for 32KB, 10% for 64KB, and 15% for 128KB.
    a. Note, this does NOT affect number of cycles. It affects the total time!
2) Associativity also takes longer. Add 5% cycle time for 2 way, 7.5% for 4 way, 10% for 8 way.
    a. Note, this does NOT affect number of cycles. It affects the total time!
3) Block size clock cycle adder to miss penalty
    a. 16 is base. 32 bytes is +2, 64 bytes is +6, 128 bytes is +14
4) Add penalties. 32KB with 4-Way is 12.5% cycle time adder.

Quick check on code

Assuming a miss penalty of 30 cycles and base sizes, the first instructions from the trace above of 31 instructions will take 151 cycles with 4 cache misses and 3 cache hits for 5 loads and 2 stores with 30 cycle miss penalty.

Defaults to assume:

1) No cache. Run the address trace assuming no cache. All loads take the miss penalty. All stores are 2 clocks (no miss penalty). We assume that there is still a downstream buffer. And, we have no cache to load lines after eviction
2) **Base cache (default)**: 16 bytes block, 16KB size, direct, miss penalty 30.

Results

Upload your code and results to ELC.

Make sure code is commented.

Flow diagram of code

Results in a spreadsheet for each of the three traces (art, mcf, and swim):

No cache, default cache

Cache size: 16KB, 32KB, 64KB, 128KB

Choose best three results and continue

Associativity: 1, 2, 4, 8

Choose best three results and continue

Block size 16, 32, 64, 128 bytes

Keep track of:

All of the initial parameters described above, plus:

1) Total instructions
2) Total cycles
3) Total CPU time (execution time)
4) Memory accesses
5) Overall miss rate
6) Read miss rate
7) Memory CPI
8) Total CPI
9) Average memory access time in cycles
10) Dirty evictions
11) Load misses
12) Store misses
13) Load hits
14) Store Hits

Using your results:

For each trace address trace (art, mcf, and swim), Plot graphs of

1) Graph Miss rate vs block size vs associativity and vs cache size
    a. Y axis is miss rate
    b. X axis is cache size
    c. Colors for associativity
    d. Line types for Block size
    e. Label clearly your graph
2) Do your results for Total CPI match miss rate results? In other words do you see better overall Total CPI with better overall miss rate? Discuss.

3) Does lowest miss rate correlate with overall best execution time? Discuss.
4) Does lowest CPI correlate with overall best execution time? Discuss.
5) Compare results for the different traces. Do you see any difference based on trace type or were the results uniform? Speculate on results and discuss your finding.
6) Choose what you believe to be the best case design for the cache assuming no limits. Justify choice. What is the difference compared to two default cases (CPI, execution time, and AMAT). What is the final clock rate? Why did you choose this design?
7) Now choose best design for fastest overall clock rate. The marketing folks want to advertise the highest clock speed. Now, what is your choice? What is the overall CPI, AMAT, and execution time and final clock rate. Justify choice. Compare to #6. What is your fastest overall clock rate?

NOTE: The traces and aspects of the lab come from Oberlin.edu Cynthia Taylor site.

From the site:

Helpful items:

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). To do this create separate memory tests (you can see the text format above) to ensure cache size, cache associativity, blocksize, and miss penalty are functioning correctly. You do not need to turn them in, but they will help tremendously.

Speed matters. These simulations should take a couple minutes (actually, much less) on an unloaded machine. If it is taking much more than that, do yourself a favor and think about what you are doing inefficiently.

Simulations are not the same as hardware. If your tag only takes 16 bits, feel free to use an integer for that value. Other time-saving optimizations along these lines might be useful.

Correctness check (I have not checked these, but I assume correct).

```
gunzip -c art.trace.gz | java cache -a 1 -s 16 -l 16 -mp 30
               Cache parameters:
               Cache Size (KB)                 16
               Cache Associativity             1
               Cache Block Size (bytes)        16
               Miss penalty (cyc)              30

               Simulation results:
               execution time 21857966 cycles
               instructions 5136716
               memory accesses 1957764
               overall miss rate 0.28
               read miss rate 0.30
               memory CPI 3.26
               total CPI 4.26
               average memory access time 8.54 cycles
               dirty evictions 60540
               load_misses 523277
               store_misses 30062
               load_hits 1208606
               store_hits 195819

> gunzip -c mcf.trace.gz  | java cache -a 8 -s 64 -l 32 -mp 42
               Cache parameters:
               Cache Size (KB)                 64
               Cache Associativity             8
               Cache Block Size (bytes)        32
               Miss penalty (cyc)              42

               Simulation results:
               execution time 143963250 cycles
               instructions 19999998
               memory accesses 6943857
               overall miss rate 0.42
               read miss rate 0.36
               memory CPI 6.20
               total CPI 7.20
               average memory access time 17.85 cycles
               dirty evictions 995694
               load_misses 2036666
               store_misses 867426
               load_hits 3552806
               store_hits 486959
```

LRU Design

It is fairly straightforward to implement an LRU policy in a high-level language like "C". However, hardware implementation can be quite a challenge.

Describe how to implement the LRU replacement policy hardware for your cache. Design for a 2-, 4-, and 8-way set associative cache.

For each, describe your solution (in words) and provide Verilog for it as well (if you'd like). You should assume a static design (clocked). Since your design is for the main L1 cache, assume the LRU decision must be completed in 1 clock cycle.

Estimate the complexity in delay stages and 2-input equivalents as a way to compare 2 vs 4 vs 8-way LRU policy in hardware. Don't forget the data selection from the ways.