Cache Simulator Assignment - ECE550 Fall 2017 Rev 1.2 (modified 11/27/17)

UPDATE 1: 11/27/17: Discussions after class today revealed some ambiguity in the simulator assignment that we will resolve by making some simplifying assumptions.

First, since the assignment was silent on the issue of write-back vs write-through caches, let's assume that all caches are write-back.

Second, questions came up about what to do in Level 2 if there is write traffic on an eviction from Level 1. Let's assume if Level 1 decides to write for any reason (either an eviction of a dirty block, or a write to a miss in a no-write-allocate situation), the write traffic first goes to the Level 2 cache.

Third, the assignment already indicated that modeling write buffers was optional. Though real systems are likely to have them, let's not do them in our simulators since getting all the details of a write buffer correct is quite tricky and involves things like snooping logic that we have not talked about in any detail. ECE552 will give you all the details!

Write a program to simulate the cache behavior of a very general memory hierarchy. You should be able to simulate one or two level caches of arbitrary configuration (direct-mapped, set-associative, fully associative) in either unified (one cache shared by instructions and data) or split (separate instruction and data caches) mode - that is, either the first or second level caches (if present) can be either unified or split.

Input will come from a trace file, in ``Dinero III'' format (the name of a well-known cache simulator). The format is one address per line:

```
<code> <hexaddress>
```

where

```
<code>
```

is 0, 1, or 2 meaning 0: Data Read, 1: Data Write, 2: Instruction Fetch

and <hexaddress> is the memory address in hex (without a leading 0x).

An example fragment from a trace input file is:

```
2 118
2 11c
1 fff8
2 120
1 fff4
2 124
2 128
2 12c
1 ffec
2 130
2 134
2 138
2 13c
1 ffe4
2 140
2 144
```

where we are fetching a number of instructions and writing 4 quantities to memory; no data reads appear in this fragment.

You will need to design command line switches or another input format that allows you to specify the configuration of the cache(s), as well as the L1 hit time in cycles, the L2 hit time in cycles if applicable, and the DRAM access time in cycles. (Assume constant DRAM access time for our purposes - no fancy page mode or the like, unless you are ambitious.) Remember that a direct-mapped cache is the same as a 1-way set associative cache and a fully associative cache is a (C/B)-way set associative cache with one set; I would recommend specifying the parameters A, B, and C: C=capacity in bytes, A=associativity, B=block size for each of the (up to) 4 caches present, as well as the None/Split/Unified option for L1 and the None/Split/Unified option for L2. Another switch will pick between ``allocate on write miss'' or not; you need not model write buffers unless you want to. Further options should vary the replacement algorithm between (at minimum) LRU and RND. NMRU and FIFO are other possibilities, there are more.

A note on "allocate on write miss," a detail we did not emphasize in class: for a read miss we always bring the missed line into the cache, as we believe from locality that it is likely to be useful again in the near future. This argument is much weaker for writes - many writes are one-off events, so making room in cache for a line we have just had a write miss on might actually lower our performance. This is a parameter we can model in our cache simulators - do we allocate (i.e. move the missed line into the cache) on a write miss or not? Does this affect the overall hit rate?

The output should include hit/miss data on each cache present; hit rates should be computed separately for instruction fetches, reads, and writes even if a unified cache is used. Details of how to compare the results of your program with those of a professional cache simulator will follow. The output should also report effective average access time as a function of the hit and miss service times supplied in the input.

For extra credit: Classify misses as compulsory, capacity, or conflict.

This will be due on Friday December 8 2017.

Implementation notes:

A linear search strategy for checking your fully associative caches will lead to slow performace. If you know how to do so, you should use some kind of binary search, hash table, or other fast searching strategy for quickly checking all entries in a fully associative cache. We will keep the trace files and examples small enough that this should not be a huge problem though.