CSE 240A Project 2

Cache Tuning Project

Due Monday, December 9, midnight

In this project you will write a cache simulator that will simulate a small cache hierarchy, which is composed of up to three levels (an L1 cache, a victim cache accessed after the L1, and an L2 cache). You will have a total budget of 300,000 bytes to optimize the performance of the cache hierarchy – that can all be allocated to the data arrays of the caches (you get the tags, LRU bits, etc. for free). You do not have to use all three levels (e.g., each level of the hierarchy adds at least one cycle to the worst case delay of a memory access), but I'm guessing you will want to.

Ashish has provided some skeleton C++ code which will help you write your simulator. You will in particular write the code to check for hits and misses, manage LRU activity, and update the cache on replacements. You will calculate the delays associated with hits and misses for each access.

As in real caches, there is an additional latency associated with larger sizes and associativities. We will model the hit time of each cache like this:

Hit Latency = ceil(size/16KB) + (associativity – 1) + ceil(log2(blocksize/16))

Thus, a 32KB, 4-way set associative, 64-byte line size cache will have a hit latency of 2+3+2=7 cycles. For small, associative caches (like, perhaps, the victim cache) you can use a different formula, which is

Hit Latency = ceil(size*associativity/2048)

So a fully (8-way) associative victim cache with 8 32-byte lines would take 1 cycle. A fully associative victim cache with 16 32-byte lines would take 4. You can use whichever formula gives you the better result for each cache – right now the code assumes the first formula.

Two other latencies may be incurred, beyond the hit latencies. Any access that gets as far as the L2 cache will see an L1-L2 transfer latency of 15 cycles, and any access that misses in the L2 will see a 350 cycle memory latency.

Right now, there are two traces, but we may add to or even replace those. The traces contain not just load and store addresses, but also the number of instructions that execute between those, so that we can calculate a CPI value (we'll assume the base CPI without cache misses is 1, and the memory latencies are added to that). Right now, you have two traces for two programs, so let's assume the metric you are trying to minimize is the average of the two CPIs. The code already does all the hard work of reading the traces.

Your program will take a bunch of arguments: L1size, L1assoc, L1blocksize, victimsize, victimassoc, victimblocksize, L2size, L2assoc, L2blocksize. The traces are compressed. So to run the simulator on the art trace and simulate an L1 (16KB, 2-way SA, 32-byte blocksize), victim (512-byte, 4-say SA, 32-byte blocksize), and L2 (256 KB, 8-way, 32-byte), we would do this:

```
gunzip –c art.trace.gz | cache-sim -t 16384 2 32 512 4 32 262144 8 32
```

cache-sim also has a –r option that you will use to hardcode a single default configuration – you will use this to encode your optimal configuration when you turn in the code. But it should also work with any reasonable –t input values that you (or we) give it.

We will make the following policy decisions. These caches are all write-back, write-allocate caches. We stall immediately on either a load miss or a store miss, until the access is complete. We will not model the cost of writing back dirty lines.

You will turn in a short report describing your methodology and design decisions. That report should describe your best design, and accompanying results. Also, for reference, please give miss rates and CPI for the following baseline cache design: L1 (16K 2-way 32-byte blocks), victim (16 32-byte lines, 4-way), L2 (256K, 4-way, 32-byte). In addition, you will turn in your code. Instructions for turnin will be posted on Piazza.