CS/COE 1541 Project 1: Cache Simulator Due 3/8 by 11:59 PM

Because there are so many parameters and options for caches, it can be very difficult to mathematically quantify how a cache will perform. Implementing different kinds of caches in real hardware is expensive and time-consuming. Instead, we use **simulators** to tweak these parameters and see what effects they have. Obviously software is much cheaper, faster, and easier to change than silicon!

This project is designed to help you understand the operation of complex caches by having you simulate caches yourself. It will support many of the features we've talked about in class. You will be given *memory access traces* from the executions of real programs. You will use these memory accesses to test the behavior of your cache simulator, and then to run experiments.

Basic Info

You will write your simulator in C (C99 standard or earlier). Please **do not** write it in C++. If you need help brushing up on C, I'm happy to help. I'm also teaching CS 0449 this semester, and there are slides/examples for that class on my site as well. Just click "CS/COE 0449" at the top.

I will give you the skeleton of your program. It will have code to read the trace files easily, parse the arguments given to your program on the command line, and some other things like constants needed to tell you what the parameters are. I will also give you a few memory traces – and more as time goes on – so you can get started testing your cache.

You do not need to simulate the contents of memory or the cache! You'll just be simulating all the metadata – the tags, valid and dirty bits – and the logic that controls them. **You can assume that this is a 32-bit CPU, and all memory accesses are words.** You can also assume that **the data bus between cache and memory is 32 bits wide.**

Since you're not simulating the contents of memory/cache, byte-level access is not important. You **WILL,** however, have to account for those 2 "byte select" bits in the LSBs of the addresses.

You will simulate two caches: one for instructions and one for data. The instruction cache will be read-only. The data cache will be read-write. The data cache will also **optionally (for extra credit!)** support having two or three levels. Because all the caches will have similar features and options, you should write your code in a **reusable** style by defining a Cache "object" that can be **configured** to do different things. Then you can make e.g. a multilevel cache by just hooking multiple cache objects together. The Cache object can also hold your statistics tracking data (like number of hits/misses etc.).

Part 1: A direct-mapped read-only instruction cache

Let's start simple! There are four parameters for the instruction cache: the **number of blocks**, how many **words per block**, the **associativity**, and the **replacement scheme.** For a direct-mapped cache, the associativity is 1, and therefore the replacement scheme does not matter.

In the **`setup_caches`** function, inspect the parameters present in the **`icache_info`** global variable to see how many blocks and words per blocks the instruction cache should be. Ignore the associativity for now. Then set up your data structures to hold the appropriate information. It'll just be easiest to make some global variables to hold it. I won't tell anyone.

Remember that the addresses are **32 bits**, and there are **2 bits** assumed to be used for byte select. Since we're using software, it's fine to use more space than needed to store this info. Memory is cheap after all ☺

In the **`handle_access`** function, add code to the **`Access_I_FETCH`** case to handle accesses to your instruction cache. (Try to keep this function, and all your functions *small*. Just have this call another function! Trust me!!)

Make liberal use of debugging prints and stuff to make sure your cache is working the way you expect. **Don't try it on huge instruction traces or with huge cache sizes first.** It's fine to start with, say, 8 blocks and an instruction trace 20 accesses long, with tiny memory addresses. Work it out on paper and confirm that your cache is doing the right thing *before* moving on to the big traces.

Add statistics tracking to your cache. It should keep track of:

- The number of reads from the cache.
- The number of **words** you had to load from main memory (when a miss happened).
- The number of *compulsory* read misses – when you fill an empty cache block for the first time.
- The number of *conflict* read misses – when you replace a full cache block with a different one.

Then, in the **`print_statistics`** function, add code to print out these statistics in a nice, neat, formatted way. You should also print:

- The total number of read misses.
- The read miss rate, as a percentage, with two digits after the decimal point. (**`printf`** specifiers are "fun.")

(As I work on my own implementation, I will give traces and what kinds of stats my cache simulator generates so you can compare and see if you're on the right track.)

Part 2: adding associativity

Now you'll modify your instruction cache to support associativity greater than 1.

You'll probably have to add some things to your data structures to support this. Then, in **`setup_caches`**, inspect the **`associativity`** and **`replacement`** fields of **`icache_info`** to configure your cache.

Start by implementing and testing the **random replacement.** The C standard library **rand()** function is good enough for our purposes. In **`setup_caches`**, you can use **`srand(30);`** or some other constant number to make sure that the same sequence of random numbers are generated every time you run your program; this will make it much easier to debug if something goes wrong. Once you're sure it's working, you can change that to **`srand((unsigned int)time(NULL));`** instead to use different numbers every time.

Next, implement and test **LRU (least-recently-used) replacement.** Unlike hardware, you've got plenty of memory to work with! Go ahead and allocate as much as you need to keep track of things. Keep it simple. Don't go overboard trying to implement a priority queue or something. Whenever you read something from the cache, that block is now the most-recently-used.

Again, start with small caches and small traces before moving onto the big ones. Software development is an incremental process. No one writes a huge program capable of doing everything in one try.

To your statistics tracking, add:

- The number of *capacity* read misses – that is, when a set is full, and you have to kick out a block (either randomly or with LRU), that's a capacity miss.

For associativity > 1, the definition of a conflict miss gets a little confused; in a fully-associative cache, there is only 1 set, so every address maps to the same set. For this reason, **you should not track conflict misses on caches with associativity > 1.**

Change your **`print_statistics`** function to account for these differences when using associative caches. Now your total misses will be compulsory + capacity.

Part 3: writing

Oh boy. Now for the data cache.

The data cache adds two more cache parameters to deal with: the **write scheme** and the **allocation scheme.** Start with a **direct-mapped (associativity = 1) cache** with write-through and write-no-allocate. Then implement write-allocate. Remember when doing write-allocate with **>1 word per block**, you've got to do *multiple reads* to fill the cache block **before** writing to it!

Now implement writing for **associativity > 1.** You already implemented the replacement schemes, but you will have to use the replacements on *writes* as well. Also, whenever you **read OR write** a block, it becomes the most-recently-used.

Last, implement **write-back.** Now you have to keep track of the dirty bits for each block. Whenever a block needs to be taken out of the cache, you now have to check if it's dirty, and if so write it back to main memory. If there are multiple words in the block, that means multiple writes to memory!

To your statistics tracking, add:

- The number of writes.
- The number of *compulsory* write misses.
- The number of *capacity* or *conflict* write misses, depending on the associativity.
- The number of **words** you had to write to memory (when a miss happened).

Change your **`print_statistics`** method to display both the read AND write statistics for the data cache.

Extra credit: multi-level data cache

If you got this far and it works, woooo! Great job! For some bonus points, you can implement a multi-level data cache. Each level can have its own configuration. When the L1 cache misses, it asks the L2 cache for the data. That, in turn, can cause it to ask the L3 cache (if any) for data, which can then ask main memory for data.

If you've written your cache code in a reusable way, implementing this shouldn't be too difficult! Each cache layer acts completely independently. When you print out your statistics, print out the stats for all the caches (the I-cache and all levels of the D-cache).

Tiny writeup

Please also write **a plain text (.txt) file** named **username.txt** (e.g. **jfb42.txt**) containing a couple paragraphs explaining:

- How much of the project you completed (including extra credit)
- What parts are not working, if any
- The compiler and compilation options you used to create your program

Doing this will help me greatly during grading. <3

Submission

Please follow these directions exactly. Pleaaaaaaaaaaaaaase

To submit your project, create a ZIP file named **username_proj1.zip** (e.g. **jfb42_proj1.zip**). It should contain:

- Your (well-commented) **cachesim.c** and **cachesim.h** files
- Your **tiny writeup**, as explained above
- Any custom memory traces you made to test your program

Then **email it to me** as an attachment, with **[CS1541 PROJECT SUBMISSION]** in the subject.