CS 211: Computer Architecture, Fall 2011
Programming Assignment 4: Cache
Due: 11:55 PM, 12/15/2011

1 Introduction

This assignment is designed to give us a better understanding about cache behavior. We will write a cache simulator in C programming language The assignment is much complex than it looks like. So, the usual warning goes double for this assignment: **do not procrastinate**.

Different set of memory access pattern will give different amount of cache hit. So, we need some tool to generate multiple memory access traces. Section 2 discusses about a tool to generate memory access traces. Functionality of this assignment is described in Section 3. Section 4 and 5 contains submission and grading policy, respectively.

2 Memory Access Trace

The files trace[1|2|3].txt, provided with assignment description, are example memory access traces of different length. Each line is for one memory access. First column is the address of the instruction that is causing memory access. R (W) indicates the operation is a memory read (write). Finally, last column is the memory address that is being accessed.

You can also use the tool pin to generate different memory access traces for any program you want. The program pin and instructions for using it are located on http://www.pintool.org/.

3 Implementation

3.1 Invocation Interface

Implement a program sim that will simulate the operation of a direct mapped cache. Your program sim should support the following usage interface:

sim [-h] <write policy> <trace file>

where:

<write policy> is one of:

wt simulate a write through cache.

wb simulate a write back cache.

<trace file> is the name of a file that contains a memory access trace.

If -h is given as an argument, your program should just print out help for how a user can run the program and then quit. For the implementation of your simulator, consider the following values:

- Cache size = 16384, Cache size is the total size of the cache.

- Block size = 4, Block size is an power of 2 integer that specifies the size of the cache block.

As an example, running the simulator using the wt option should produce:

```
./sim wt trace0.txt
```

```
CACHE HITS: 710738
CACHE MISSES: 30478
MEMORY R (READ): 30478
MEMORY W (WRITE): 238846
```

3.2 Simulation Details

At the start of your simulation - that is, when your program first starts running - all entries in the cache should be invalid. You can assume that the memory size is 2^{32} . The number of bits in the tag, cache address, and byte address are then determine by the cache size and block size. Your simulator should simulate the operation of a cache according to the given parameters for the given trace. At the end, it should print out the number of memory reads, memory writes, cache hits, and cache misses. For a write-through cache, there is the question of what should happen if a write generated by the processor results in a cache miss. For this assignment, your cache simulator should bring the corresponding block in, then execute the write. This has two implications: (1) a write-miss causes both a read and a write from the cache to the memory, and (2) future reads or write to any location in the newly brought in block will hit in the cache until the block is evicted because of replacement. This makes write-through and write-back caches look alike as much as possible, easing your implementation.

4 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa3.tar`. To create this file, put your source code, Makefile, and readme.pdf in a folder named pa1; `cd` to the directory containing this folder; then run the following command:

```
tar -cvf pa3.tar pa3
```

To check that you have correctly created the tar file, you should copy it (`pa3.tar`) into an empty directory and run the following command:

```
tar -xvf pa3.tar
```

This should extract all the files that we are asking for below directly into the empty directory.

Use ONLY tar command to compress your submission. Using other kind of compression tool like zip, 7zip, etc will cause us troubles in opening your submission, and you will likely lose points for doing so. Ask your TAs for help on this if you are unsure.

Your tar file must contain:

- readme.pdf: this file should describe the design and implementation of your cache simulator. How did you implement write back vs. write through? How did you account for the different types of misses? etc. For this assignment, you do not have to worry about analyzing the time and space behavior of your program.
- Makefile: there should be at least two rules in your Makefile:

  `sim:` build your `sim` executable.

`clean:` prepare for rebuilding from scratch.

- source code: all source code files necessary for building `sim`. You are recommended to put any global definitions and function declarations in the header file, while put function definitions in the source file.

We will compile and test your program on the Cereal machines so you should make sure that your program compiles and runs correctly on these machines. You must compile all C code using the gcc compiler with the `-ansi -pedantic -Wall` flags.

5 Grading Guidelines

5.1 Functionality

This is a large class so that necessarily a significant part of your grade will be based on programmatic checking of your program. That is, we will build a binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can and MAKE SURE they can run on the ilab machines. *In particular, your code should be adept at handling exceptional cases.* For example, `sim` should *not* crash if the argument trace file does not exist.

Be careful to follow all instructions. If something doesn't seem right, ask.

5.2 Coding Style

Having said the above about functionality, it is also important that you write "good" code. Thus, *part of your grade will depend on the quality of your code.* Here are some guidelines for what we consider to be good:

- Your code is modularized. That is, your code is split into pieces that make sense, where the pieces are neither too small nor too big.
- Your code is well documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.
- You use variable names that have some meaning (rather than cryptic names like `aa`).

Further, you should observe the following protocols to make it easier for us to look at your code:

- Define prototypes for all functions.
- Place all prototype, `typedef`, and `struct` definitions in header (.h) files.