CSC 252 Project 4: C Cache Simulator

1 Introduction

In this project, you will get familiar with the operation of caches by designing a cache simulator in C. A basic skeleton for the project has been provided. Your job is to complete the simulator so that it can determine the hit rate, count transactions to main memory, and classify each hit and miss to the cache.

2 Teaching Assistants

- Alex Benishek (abenishe@u.rochester.edu)
- Ebrahim Shah (eshah2@u.rochester.edu)
- Lee Stovall (lstovall@u.rochester.edu)

3 Cache Specifications

- Allow user configuration of associativity, size and block size.
- Read the user specified trace.
- Classify each memory read/write as a:
  - Hit
  - Compulsory Miss
  - Capacity Miss
  - Conflict Miss

- Output to stdout the miss rate, the number of memory reads, and the number of memory writes.
- Output a file indicating the classification of each access.

4 Cache Operation

While there are many parameters important to designing a cache, this project will focus on the following parameters:

1. Line (Block) Size
2. Cache Size
3. Associativity

For this project, all addresses will be 32-bits. Because caches are smaller than main memory, multiple locations in main memory will map to the same location in the cache. When a cache is given an address, it breaks it down into three components: tag, index, and offset.

The offset is used to determine where in a line the data is to be read from or written to. Since the project does not care about the data in the cache, this offset may be ignored.

The index is used in all but fully associative caches (which may place data on any line), and is used to determine which line or set the data may be found in.

The tag is used to determine whether the data present in the cache is the data that would be found in memory (recall that multiple addresses in memory map to the same location in the cache).

To determine the size of the tag, index, and offset, first divide the cache size by the line size: this will determine the number of lines in the cache. Divide this by the associativity to determine the number of sets (or ways). In a direct-mapped cache, the associativity is 1. Take the logarithm of the number of sets to determine the number of bits in the index. Take the logarithm of the number of bytes in a line to get the offset. The remaining bits in the address are the tag.

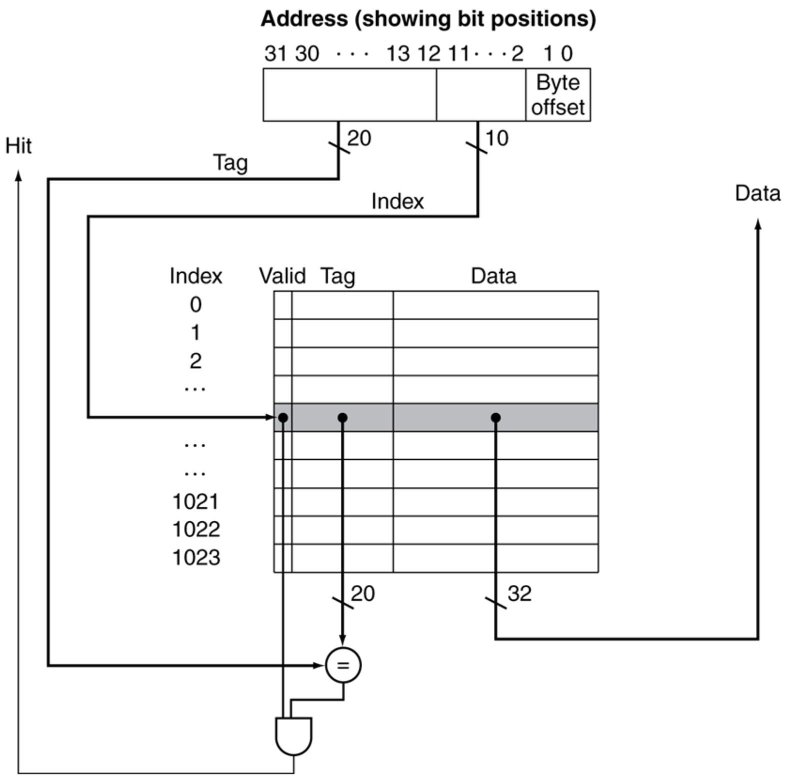

Figure 1: Direct Mapped Cache

$$bits_{offset} = log_2(size_{line})$$

$$sets = \frac{size_{cache}}{size_{line} \times associativity}$$

$$bit_{index} = log_2(sets)$$

$$bits_{tag} = 32 - (bits_{index} + bits_{offset})$$

For example, in Figure 1, we have a direct-mapped (associativity = 1), 4KB cache with 4B lines (blocks). Because there are 4 of data bytes to each line, the offset must be 2 bits. Likewise, there must be 1024 sets, as $\frac{4KB}{4B \times 1} = 1024$. To properly index into each of these sets, there must be 10 bits of index. This leaves 20 bits remaining as the tag.

Physically, in the cache, each line has the data, the tag, a valid bit, and a dirty bit (we are assuming a write-back cache). When checking for a hit in the cache, the valid bit must also be checked to determine if the data in the cache is merely junk. The reason for this is that on startup, the cache is cold (has no data in it), and all values currently in the cache have no meaning, as they dont represent the data in the memory. Whenever a line is brought into the cache, the valid bit is set, indicating the the data on that line is, in fact, valid. Whenever any data in a cache line is modified, the dirty bit is set so that when the line is eventually evicted, the data will be written back to main memory. For simplicity, we will assume this cache has a write-allocate policy (it retrieves data from main memory before writing on a write miss).

5 Cache Details

5.1 Replacement Policy

In set-associative and fully-associative caches, are able to decide which line to evict from the cache when servicing a miss. There are many replacement policies used to make these decisions, but for simplicity, this project will use a FIFO replacement policy. In the FIFO replacement policy, the line that is brought in the earliest will be replaced, regardless of when it was last accessed.

5.2 Policy for Associative Cache

The way one would program the cache is by simultaneously doing three simulations. First, there would be an infinite cache that simply checks if a cache block has been seen before. Second, is a finite, fully associative cache, **which uses the optimal (OPT) replacement policy**. Third, a finite cache with finite associativity and with the FIFO replacement policy. The reason for using OPT in the associative cache is that the purpose of the associative cache is to check for capacity. The only cause of a capacity miss is a cache which is too small. If the programmer uses a sub-optimal replacement policy then a cache block might not be in your associative cache not because you did not have enough room for it, but because you chose the wrong replacement policy. By choosing OPT as our replacement policy, we know that the only two reasons a block might not be in the associative cache is that we have not seen it before (compulsory) or we did not have enough room for it (capacity).

Now, since you did not know that you needed to use OPT as the replacement policy for the associative cache, we will accept either a simulator which uses FIFO as the replacement policy for the associative cache, or a simulator which uses OPT as the replacement policy for the associative cache. GOLD files for simulators using OPT are under **correct_GOLD**, while files for simulators using FIFO for the associative cache are under **incorrect_GOLD**.

6 Classifying Cache Accesses

Accesses to the cache can be classified in the following ways:

1. Hit (Line is valid, and tags match: data is already in the cache)
2. Compulsory Miss (Line is not valid or never accessed previously)
3. Conflict Miss (Line was evicted to make room for another line, even though the replacement policy could have evicted a different line with higher associativity. Fully associative caches do not have conflict misses.)
4. Capacity Miss (Not enough room in cache for line. Defined as a miss that is not a compulsory or conflict miss)

7 Traces

You will be testing you simulator with the following traces from the SPECINT 2000 Benchmarks:

1. gcc
2. gzip
3. mcf
4. swim
5. twolf

In each trace file, the first column is the access type (l is a load, s is a store). The second column is the address. An excerpt of the gcc trace is included below:

```
s  0x1fffff50
l  0x1fffff58
l  0x1fffff88
l  0x1fffff90
l  0x1fffff98
l  0x1fffffa0
l  0x1fffffa8
l  0x1fffffb0
l  0x1fffffb8
l  0x1fffffc0
```

8 Usage & Expected Output

8.1 Usage

The invocation of the simulator will be as follows:

```
./cacheSim [-s <size>] [-w <ways>] [-l <line>] [-t <trace>]
```

You can assume that both the cache and lines sizes will be a power of 2 number of bytes, where the number of lines will also be a power of 2 (e.g. 64KB cache with 16B lines).

Some additional specifications of the cache:

- Allow arbitrary associativity.
- FIFO replacement policy
- Write-back write policy

8.2 Running Parameters

For this project, you will run the cache with the following parameters:

1. 16KB, direct-mapped, with 32B blocks
2. 32KB, 2-way set associative, with 64B blocks

3. 64KB, 4-way set associative, with 64B blocks
4. 64KB, 8-way set associative, with 128B blocks

8.3 Expected Output

When run, the simulator should output the parameters of the cache (number of ways, number of sets, size of a line), the size of each field in the address (tag, index, offset), the miss rate (as a percentage), and the number of read and write transactions to main memory.

An example run is included below:

```
./cacheSim -t traces/gcc.trace -s 1 -w 2 -l 256
Ways:  2; Sets:  2; Line Size:  256 B
Tag:  23 bits; Index:  1 bits; Offset:  8 bits
Miss Rate:  21.030943%
Read Transactions:  108453
Write Transactions:  19014
```

Make sure that the output is formatted correctly, as we will be using diff to verify that it matches the expected outcome.

Finally, the simulator should output a file in the same directory as the trace with the name [trace].simulated (gcc.trace becomes gcc.trace.simulated). The format of this file is similar to the input trace, with one additional column.

Cache hits are classified with hit, compulsory misses with compulsory, capacity misses with capacity and conflict misses with conflict.

For example:

```
s  0x00000000  compulsory
l  0x00000000  hit
s  0x00000200  compulsory
l  0x01000200  compulsory
l  0x04000200  compulsory
l  0x00000200  conflict
l  0x00000000  conflict
l  0x05000200  compulsory
l  0x05900200  compulsory
l  0x05910200  compulsory
l  0x01000200  capacity
```

9 Grading

The project is out of 100 points, split between the five programs, 20 points each. For each program, if you get the percentage of hits and misses correct you will receive 4 points. 16 points are awarded for running each program in four configurations:

1. 16KB, direct-mapped, with 32B blocks

2. 32KB, 2-way set associative, with 64B blocks

3. 64KB, 4-way set associative, with 64B blocks

4. 64KB, 8-way set associative, with 128B blocks

For each configuration, 0 points are awarded for getting 50% or below of the miss classifications correct, and 4 points are given for getting 100% of the miss classifications correct. Points are scaled linearly between 50% and 100%.

9.1 Gold Files

In the project directory, there will be a directory named GOLD. This file contains the expected output from the simulator. These are so you can compare the output of your program to the gold files.

9.2 Trace Outputs

Trace output files will have the name of the trace followed by the cache parameters used when simulating the cache with the trace.

```
<trace>_s<cache size>_w<associativity>_l<block size>.GOLD
```

For example, gcc_s32_w1_l64.GOLD is the gold file for a run of gcc.trace with a 32KB direct-mapped cache with 64B blocks. You can diff your file with the GOLD file to determine if your simulator is working correctly.

9.3 Stdout

These gold outputs have .stdout in their name, and correspond to the terminal output of running the simulator.

10 Submission

We have provided a makefile and you should make sure that your project can be compiled with that makefile. Specifically, you should be able to run the following command to compile your project:

```
make
```

The executable should be named “cacheSim” and use the command line parameters that are specified in the “Usage” section. The executable will be run from the same directory as this file. Remember to follow the naming convection specified in the “Expected Output” section for any files written by your program.

11 Plagiarism

What you submit must be your own work (or that of your partner, if you are working in a two-person group). You are permitted (and encouraged) to discuss ideas with other groups, but you must not share code to avoid accidental appropriation.