

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 Cache Specifications

In this project, you will implement a cache simulator that will:

- 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, number of read and write transactions to memory and emit a file indicating the classification of each memory access.
- Output a file indicating the classification of each access.

3 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.

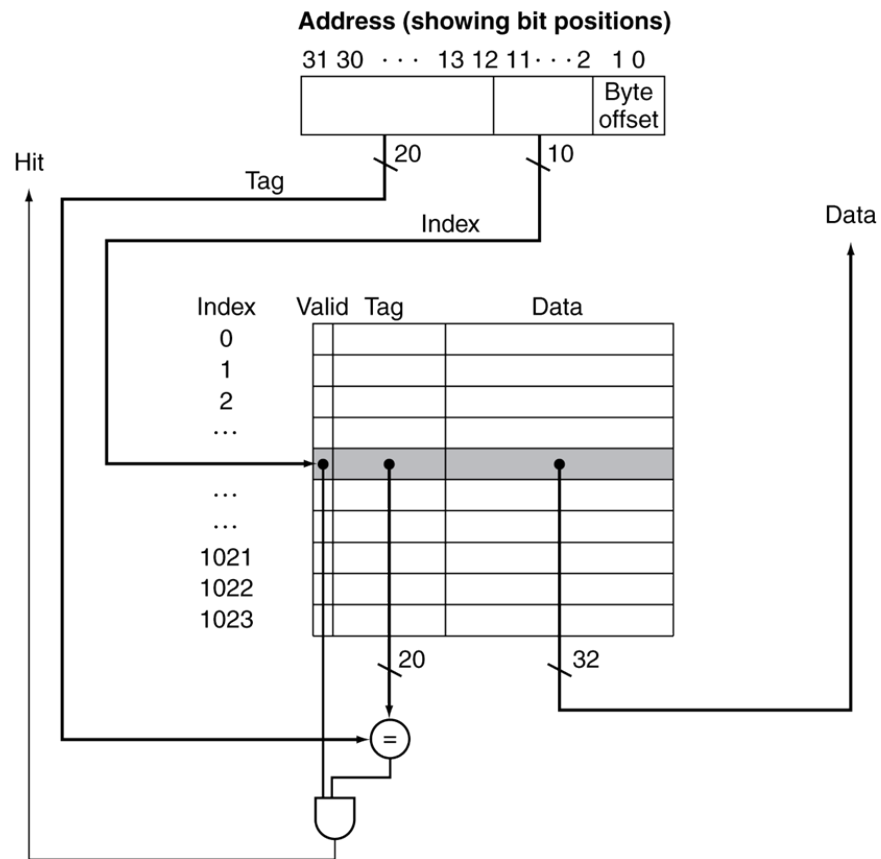


Figure 1: Design of a Direct Mapped Cache

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

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

$$bits_{index} = \log_2(sets)$$

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

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 $\frac{4KB}{4B*1} = 1024$. 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 don't 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).

3.1 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.

4 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)

5 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 0x1ffffff50
l 0x1ffffff58
l 0x1ffffff88
l 0x1ffffff90
l 0x1ffffff98
l 0x1ffffffa0
l 0x1ffffffa8
l 0x1ffffffb0
l 0x1ffffffb8
l 0x1ffffffc0
```

6 Usage & Expected Output

6.1 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:

- Allows associativity of 1 or 2 (direct-mapped or 2-way set associative). If given any other number, the simulator should quit and print an error.
- FIFO replacement policy
- Write-back write policy

6.2 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

6.3 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: 256B
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
```

```
1 0x04000200 compulsory
1 0x00000200 conflict
1 0x00000000 conflict
1 0x05000200 compulsory
1 0x05900200 compulsory
1 0x05910200 compulsory
1 0x01000200 capacity
```

7 Grading

Your grade will be based on:

1. Accuracy of miss rate and number of read and write transactions to the memory
2. Correct classification of hits and misses

Each trace will be worth a total of 20 points.

For each trace, 4 points will be awarded for correctly identifying the miss rate and counting memory transactions.

For each trace, 16 points will be awarded for correctly classifying each hit/miss in a direct-mapped cache and a 2-way set-associative cache (8 points each). For each cache that cannot correctly classify misses (but still identifies hits vs. misses), 4 points will be awarded.

7.1 Gold Files

In the project directory, there will be a directory named GOLD. This file contains the expected output from the simulator.

7.1.1 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.

Some GOLD files will also have a LRU at the end of the name. Ignore these unless doing the LRU extra credit.

7.1.2 Stdout

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

8 Extra Credit

This project will include 100 points of extra credit (for comparison, this project is worth 100 points).

1. Higher Associativity (4+ way set associative and fully associative) [25 pts]

To test this, run your cache with the following parameters (gcc trace only):

- (a) 32KB, 512-way set associative, with 64B blocks

2. LRU Replacement Policy (in addition to FIFO) **Requires Higher Associativity** [50 pts]

For this piece of extra credit, the simulator will accept an *optional* additional argument, which will be the flag "-lru". If this flag is passed, the simulator should use a LRU replacement policy; otherwise it should use FIFO.

To test this, run your cache with the following parameters (gcc trace only):

- (a) 32KB, 8-way set associative, with 128B blocks, LRU

3. Design a trace [25 pts]

This trace must have a 100% miss rate in a 1KB 2-way set-associative cache with 32B blocks. It must have at least 5 of each type of miss (compulsory, capacity, conflict). This trace may have no more than 100 accesses.

9 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.