CSc 656 Project 2: Cache simulation

due Thursday 4/26/2018, 3pm (no late submissions)
(8% of your grade)
[document updated 4/19/2018]

This is an individual project. Work on your own.

For this project, you will write a data cache simulator that uses data memory address traces as input. All code/data can be found on unixlab.sfsu.edu:

Code: `~whsu/csc656/Code/S18/P2`
Data: `~whsu/csc656/Traces/S18/P1`

We are using the same traces as for Project 1:
https://www.cis.upenn.edu/~milom/cis501-Fall12/traces/trace-format.html

The relevant fields for this project are (following the numbering on the website above):

Field 2 PC: the address of the instruction
Field 8: this is a single character
- 'L' means load (instruction reads data from memory, like a MIPS lw)
- 'S' means store (instruction writes data to memory, like a MIPS sw)
- '-' means instruction does not read/write data memory

Field 10 data memory address

The first few lines of test.trace again:

```
1 48d1de -1 -1 13 - - -        0            0       48d1e2           0 SET      ADD
2 48d1de -1 -1 13 R - -        1            0       48d1e2           0 SET      ADD_IMM
1 48d1e2 -1  5 45 - - L     -264  7fffe7ff048       48d1e9           0 CMP      LOAD
2 48d1e2 45  3 44 W - -        0            0       48d1e9           0 CMP      SUB
1 48d1e9 -1  5  3 - - L     -200  7fffe7ff088       48d1f0           0 MOV      LOAD
1 48d1f0 -1 -1  0 - - -        0            0       48d1f3           0 SET      ADD
```

Note the data memory addresses `0x7fffe7ff048` and `0x7fffe7ff088`! Make sure you use variable that are large enough to hold all the address bits.

You may assume that each data memory access will touch no more than 16 bytes.

Benchmark Traces

Use the same two traces that were provided for Project 1b: gcc and sjeng.

You should make all measurements for each cache configuration, for both benchmarks.

Cache simulator implementation

You will write a cache simulator to collect statistics on events in the memory system. Your code will allocate some arrays that represent the state of the cache, read each line of a trace file in the specified format, and update the state of the proper cache array elements accordingly.

While you are only required to show measurements for a small number of cache configurations, your code *must* work for any cache size that is a power of 2 bytes.

Your simulator tracks the behavior of a writeback data cache, as defined below.

Cache System Characteristics

For both benchmarks, study cache configurations with 16-byte blocks. Make measurements for these systems:

System 1: Direct-mapped data cache, 2KB and 4KB
System 2: k-way set associative data cache, 2KB and 4KB

Make measurements for each benchmark, for the following parameters:

System	Cache configuration
System 1a	2KB direct-mapped
System 1b	4KB direct-mapped
System 2a	2KB 2-way set associative
System 2b	2KB 4-way set associative
System 2c	4KB 2-way set associative
System 2d	4KB 4-way set associative

Assume that all read hits take 1 cycle.
Assume that the miss penalty is equal to 80 cycles.
On a write miss, assume that the requested block is always brought into the cache, and the CPU writes into the proper word in the cache.

You should collect these statistics, and display them in order (see sample executables):

- number of data reads (i.e., number of loads),
- number of data writes (i.e., number of stores),
- number of data accesses (i.e., number of loads + number of stores),
- number of total data read misses,
- number of total data write misses,
- number of data misses (i.e., number of accesses that miss the cache),
- number of dirty data read misses,
- number of dirty write misses,
- number of bytes read from memory,
- number of bytes written to memory,
- the total access time (in cycles) for reads,

- the total access time (in cycles) for writes,
- the overall data cache miss rate ((read misses + write misses) / total accesses)

A data read miss is any read access (i.e., load) that results in an access to memory.
A data write miss is any write access (i.e., store) that results in an access to memory.
A clean miss is a miss that does not replace a dirty block in the cache (i.e., the dirty bit for the block is off, or the block is not valid).
A dirty miss is a miss that replaces a dirty block in the cache.
(Note that some of these measurements give overlapping information, and it is easy to check whether your results are consistent by comparing some of the measurements.)

Follow the definitions given below for detailed operations, and how hits and misses are counted.

Detailed direct-mapped data cache operation (System 1)

Suppose an access has address A. A maps to index I in the direct-mapped data cache. Each access falls under one of these cases:

Case 1: the block containing A is found in data cache (***cache hit***)
Read: no state changes, *1 cycle*
Write: set dirty bit = 1, *1 cycle*

[Case 2a/2b cover misses. For a cache miss, index I in the data cache may contain block X (different from the block that contains address A), or may be empty.]

Case 2a: the block containing A is not found in data cache, index I in data cache either contains block X (which is clean), or is empty (***clean cache miss***)
Read: move block containing A from memory into index I in data cache,
dirty bit = 0, *(1 + miss penalty) cycles*
Write: move block containing A from memory into index I data cache, dirty bit = 1
(1 + miss penalty) cycles

Case 2b: the block containing A is not found in data cache, index I in data cache contains block X in data cache, which is dirty (***dirty cache miss***)
Read: write block X to memory
move block containing A from memory into data cache, dirty bit = 0
*(1 + 2 * miss penalty) cycles*
Write: write block X to memory
move block containing A from memory into data cache, dirty bit = 1
*(1 + 2 * miss penalty) cycles*

Detailed k-way set-associative data cache operation (System 2)

All cases are basically the same as for System 1, except the cache is k-way set-associative.

Suppose an access has address A. A maps to index I in the direct-mapped data cache. Each index I in the data cache accesses one of k blocks. Assign ids 0 to k-1 to the k blocks within the same index.

For each block in the k-way set-associative cache, in addition to the bit-fields in the direct-mapped cache, there is also a *last-used* field. This contains the current instruction count of the access that last touched that block, i.e., the order of the access in the trace file, starting from 0.

Index I in the data cache contains block X_0 to X_{k-1} , corresponding to ids 0 to k-1; each of these k blocks is either different from the block that contains address A, or is empty.

Each access falls under one of these cases:

Case 1: the block containing A is found in index I id D in the data cache (***cache hit***)
    Read: last-used for index I id D = current IC, *1 cycle*
    Write: last-used for index I id D = current IC, set dirty bit = 1, *1 cycle*

[Case 2a/2b cover misses. For a cache miss, the block containing A maps to index I. First choose one of k blocks. If there is an empty block, this is the empty block with the smallest id. If there is no empty block, this is the least recently used block, according to the *last-used* field for that block. Let the id of the chosen block be D.]

Case 2a: the block containing A is not found in data cache, the chosen block with id D is either clean, or is empty (***clean cache miss***)
    Read: move block containing A from memory into block D in data cache,
        last-used for index I id D = current IC, dirty bit = 0
        (1 + miss penalty) cycles
    Write: move block containing A from memory into index I data cache
        last-used for index I id D = current IC, dirty bit = 1
        (1 + miss penalty) cycles

Case 2b: the block containing A is not found in data cache, the chosen block with id D is dirty (***dirty cache miss***)
    Read: write block X to memory
        move block containing A from memory into block D in data cache,
        last-used for index I id D = current IC, dirty bit = 0
        *(1 + 2 * miss penalty) cycles*
    Write: write block X to memory
        move block containing A from memory into block D in data cache,
        last-used for index I id D = current IC, dirty bit = 1
        *(1 + 2 * miss penalty) cycles*

Command line format

Your simulators will follow this command line format. For sys1:

```
./sys1 tracefile cachesize [-v ic1 ic2]
```

tracefile is the name of the trace file, *cachesize* is the size of the cache in KB, and –v is the optional verbose flag. Verbose output format will be defined in the next section.

For example, to run sys1 on gcc.xac with a 4KB cache, showing references 100-200 in verbose mode:

```
./sys1 gcc.xac 4 -v 100 200
```

Sys2 has similar command line format as sys1, but with an extra argument, *set-associativity,* for set associativity k:

```
./sys2 tracefile cachesize set-associativity [-v ic1 ic2]
```

For example, to run sys2 on gcc.xac with a 4KB 8-way set-associative cache, showing references 100-200 in verbose mode:

```
./sys2 gcc.xac 4 8 -v 100 200
```

Verbose format

The –v flag is always followed by two integers ic1 and ic2. The data memory accesses in the trace file are numbered in sequential order, starting from zero. Verbose output is displayed for access ic1 to ic2, inclusive.

For System 1 (direct-mapped cache), the verbose format output comprises:

order of the load/store in address stream (starting from 0, in decimal)
data address accessed by the load/store (in hex)
cache index calculated from address (in hex)
cache tag calculated from address (in hex)
valid bit of cache block accessed
tag stored in cache block accessed (in hex; 0 if invalid)
dirty bit of cache block accessed (before current access)
hit or miss (0 miss, 1 hit)
Case number according to Detailed Operation (1, 2a, or 2b)

For example, reference 1, 5, 6, and 240 from gcc.xac for a 2KB direct-mapped cache look like this:

```
1 1888648 64 3110 0 0 0 0 2a
5 1888648 64 3110 1 3110 0 1 1
6 1888648 64 3110 1 3110 0 1 1
240 7bce40 64 64 f79 1 3110 1 0 2b
```

The original reference address is 0x1888648, which breaks up into tag = 0x3110 and index = 0x64. The cache was empty, so the valid bit is 0, the old tag and dirty bits are also 0. It's a miss (0), and this is Case 2a (clean miss with empty cache block).

Then in references 5 (load) and 6 (store), the original reference addresses are both 0x1888648. These are both hits; Case 1.

Index 64 is not touched until reference 240, with original address 0x`7bce40`. We see that while the index for this reference is 0x64, the tag is 0xf79. Reference 6 was a write, so index 0x64 is dirty. Hence, reference 240 is a dirty miss, Case 2b.

For System 2 (k-way set-associative cache), the verbose format comprises:

order of the load/store in address stream (starting from 0, in decimal)
data address accessed by the load/store (in hex)
cache index calculated from address (in hex)
cache tag calculated from address (in hex)
valid bit of cache block accessed
id of the block chosen (D, from the specification)
last-used field of the block chosen (in decimal, 0 if invalid)
tag stored in cache block accessed (in hex, 0 if invalid)
dirty bit of cache block accessed (before current access)
hit or miss (0 miss, 1 hit)
Case number according to Detailed Operation (1, 2a, or 2b)

For example, consider five references from gcc.xac for a 2KB 2-way set-associative cache (all involving index 0x33):

```
2 7fffe7fef38 33 1ffff9ffb 0 0 0 0 0 0 2a
10 7fffe7fef38 33 1ffff9ffb 1 0 2 1ffff9ffb 1 1 1
30 aae338 33 2ab8 0 1 0 0 0 0 2a
31 aae330 33 2ab8 1 1 30 2ab8 0 1 1
110 a29730 33 28a5 1 0 10 1ffff9ffb 1 0 2b
```

Reference 2 is a store to data address 0x`7fffe7fef38.` It has index 0x33, tag 0x`1ffff9ffb`. This is the first reference to index 0x33; it's a clean miss (Case 2a). The block from memory is placed in index 0x33 ID=0 in the cache block.

Reference 10 also has data address 0x`7fffe7fef38.` This is a hit. Note that the last-used field is now 2 (since this block was last touched by reference 2). The tag in the cache block is now the tag from reference 2, the dirty bit is 1 (written by reference 2).

Reference 30 is a load with data address 0xaae338. It has index 0x33, tag 0x2ab8. This is also a reference to index 0x33; since the block with ID=0 is already taken, but the block with ID=1 is invalid, the block goes into the latter. Hence, it's a clean miss (Case 2a). The block from memory is placed in index 0x33 ID=1 in the cache block. Similarly Reference 31 (same index, matching tag with block with ID 1), is a hit (Case 1). Note again that the last-used field is now 30.

References 32-109 do not touch index 0x33. At this point, index 0x33 contains two blocks:

ID	tag	valid	dirty	Last used
0	`0x1ffff9ffb`	1	1	10
1	`0x2ab8`	1	0	31

Reference 110 has data memory address 0xa29730, also mapping to index=0x33. This reference is not found in the two blocks with index=0x33. The block with ID=0 has last-used=10, and is least recently used. Hence, reference 110 is placed in the block with ID=0. This is a dirty miss, Case 2b.

Submission (2 parts):

Your code will have to run on unixlab.sfsu.edu. Again, note that the C++/Java versions on unixlab are not the latest; if you use language constructs that were not available with earlier versions of the compilers, your code will not run.

Source code submission: submit a single tar, zip or other archive file using iLearn. Your archive file should expand into a single directory tagged with your login name. The directory should contain your source files and a makefile.

I should be able to cd into the directory you submitted, type

```
make sys1
make sys2
```

to generate two executables, sys1 (for System 1) and sys2 (for System 2). I will then run each one with this command line:

```
./sys1 tracefile 2
./sys1 tracefile 4
./sys2 tracefile 2 2 etc etc
```

If you don’t provide makefiles, you *must* include a README file that includes step-by-step instructions for compiling your source code.

You may write your cache simulator in Java, with or without an IDE. However, you *must* include a README file that includes clear step-by-step instructions for compiling and running your code on the Unix command line.

If you would like to deviate from the specified command line interface, or use another language etc, you must get my permission before class Thursday 4/19.

Report submission: Submit a table (in pdf format) showing *miss rate* and *total access time* for each system, for each benchmark you are responsible for. Your table should look something like

	Benchmark 1 miss rate	Benchmark 1 tot. access time	Benchmark 2 miss rate	Benchmark 2 tot. access time
System 1a				
System 1b				
System 2a				
System 2b				
System 2c				
System 2d				