CDA3101 – Project 3

Cache Simulator

Submission:

1. Create a tar file that includes your program and any additional info you may the TAs to know.
2. Make an electronic submission of your tarball file to Canvas.

Due Date:

1. Nov 30, 2018, 11:50pm for all students

Late Penalty:

1. 15% per day (round up)
2. Maximum **one** day

1. Purpose

The purpose of this project is to practice your understanding of on the organizations of caches of various sizes and configurations, as well as the write policies of write-through, write-allocate and write-back.

2. Basic Description

Your job is to create a program, contained in a single C or C++ file called `proj3.c` or `proj3.cpp`. You program should take three integer arguments with the following information:

```
<blocksize>: size of block in bytes.
<numbsets>: number of sets in the cache.
<assoc>: associativity of cache (1, 2 or 4).
```

Your submitted proj3.c or proj3.cpp must compile and run on `linprog` as in the following example:

```
$ gcc -o simCache proj3.c
$ g++ -o simCache proj3.cpp
$ ./simCache 16 64 2 < test1.trace
```

Of course, you need to use g++ if you have a C++ program, or gcc in case you have a C program. **If you use C, a partial program is provided. You are on your own if you use C++**. From this point on, all descriptions assume that the program is written in C. You should not rely on any special compilation flags or other input methods. If your program does not compile and execute as expected with the above commands, you will receive zero points on project 3.

The contents of the .trace files used as input are described below. There is no provided source code for this project so you should start by writing C code to parse the command line arguments and trace files.

After reading in the command line arguments, you should print out the number of bytes in a block, the number of sets in the cache, and the associativity of the cache. Additionally, you should print the number of bits required for each partition of a memory reference: tag, index, and offset.

Afterward, you should prepare to simulate the behavior of two caches on the sequence of references in the trace file. The first cache has a write-through, no write allocate policy. The second has a write-back, write allocate policy. For every reference, you should determine for each cache whether the reference is a hit or a miss, and whether main memory needs to be accessed. After all references have been processed, print the statistics for each cache.

For example, consider the contents of the trace file test.trace:

```
W 300
R 304
R 4404
W 4408
W 8496
R 8500
R 304
```

Let's use this trace file as input to our program and specify that our caches should have 64 sets with 2 blocks each and 16 bytes per block.

```
$./simCache 16 64 2 < test.trace

Block size: 16
Number of sets: 64
Associativity: 2
Number of offset bits: 4
Number of index bits: 6
Number of tag bits: 22
****************************************
Write-through with No Write Allocate
****************************************
Total number of references: 7
Hits: 1
Misses: 6
Memory References: 7
****************************************
Write-back with Write Allocate
****************************************
Total number of references: 7
Hits: 2
Misses: 5
Writebacks: 1
Memory References: 6
```

3. Trace File Description

The .trace files that will be used as input to your cache simulator contain up to 250 lines, where each line has the following format:

```
<access_type> <byte_address>
```

The access type can be either 'R' for reading or 'W' for writing. The byte address is simply the decimal representation of the 32-bit address of the reference. An example .trace file contents follows:

```
R 4096
R 5000
W 5000
```

Note: MIPS is word-aligned, which means that we can only legitimately access every 4th byte. All provided test files will contain word-aligned references, but you do not need to check to make sure the references are word-aligned.

4. Suggested Development Approach

In this project, you are not given any starting source code so you shall start as soon as possible to allow yourself time to plan the structure of your program.

The first thing you should work on is the parsing of the command line arguments. You may assume that the block size and number of sets provided as arguments to the program will always be powers-of-two. After you have parsed the arguments, echo the values back to the user as shown in the sample output. You should also calculate and print the number of offset bits, index bits, and tag bits, as discussed in Lecture 9. You may assume that the addresses in the test file correspond to 32-bit binary addresses.

```
Block size: 16
Number of sets: 64
Associativity: 2
Number of offset bits: 4
Number of index bits: 6
Number of tag bits: 22
```

Next, you should implement a *write-through, no write-allocate* cache and supporting functionality for reading from and writing to the cache.

There are several ways to implement the cache in your C code, but the most straightforward is probably through the use of two-dimensional arrays (or a flattened one-dimensional array). You may create a dynamically-sized array based on the input parameters using the malloc() function. You need to remember to call free() to deallocate the memory before you exit the program.

For a *write-through, no write-allocate* cache, we have the following properties:

- When a block is present in the cache (hit), a read simply grabs the data for the processor.
- When a block is present in the cache (hit), a write will update both the cache and main memory (i.e. we are ***writing through*** to main memory).
- When a block is not present in the cache (miss), a read will cause the block to be pulled from main memory into the cache, replacing the **least recently used block** if necessary.
- When a block is not present in the cache (miss), a write will update the block in main memory but we do not bring the block into the cache (this is why it is called "***no write allocate***").

After you have implemented the write-through cache, you should implement the write-back cache. One way to approach this is to have two separate caches, one for write-through and the other for write-back, which are both updated independently every time a reference is read from the trace file.

For a *write-back, write-allocate* cache, we have the following properties:

- When a block is present in the cache (hit), a read simply grabs the data for the processor.
- When a block is present in the cache (hit), a write will update only the cache block and set the dirty bit for the block. The dirty bit indicates that the cache entry is not in sync with main memory and will need to be written back to main memory when the block is evicted from the cache.
- When a block is not present in the cache (miss), a read will cause the block to be pulled from main memory into the cache, replacing the **least recently used block** if necessary. *If the block being evicted is dirty, the block's contents must be **written back** to main memory.*

- When a block is not present in the cache (miss), a write will cause the block to be pulled from main memory into the cache, replacing and evicting the **least recently used block** if necessary. When the block is pulled into the cache, it should immediately be marked as "dirty". *If the block being evicted is dirty, the block's contents must be **written back** to main memory*.

Using these properties, you must calculate the number of hits and misses, the memory references, as well as the number of writebacks (for write-back cache only) invoked by the sequence of addresses in the trace file. A memory reference is defined as an access to main memory in order to either update or read a block. You do not need to model the data or main memory. The purpose is to only simulate the effect of the references, not deal with actual data.

5. Breakdown of points

The grade breakdown for the project is as follows:

Points	Requirements
5	Conformant to the required compilation and execution commands.
5	Compile, execute and interpret instructions
10	Calculate the bits for tag, index and offset correctly
10	Implement the LRU policy correctly
Write-Through, No Write-Allocate Cache	
10	Count hits, misses and memory references correctly for direct-mapped cache
10	Count hits, misses and memory references correctly for 2-way associative cache
10	Count hits, misses and memory references correctly for 4-way associative cache
Write-Back, Write-Allocate Cache	
10	Count hits and misses correctly for direct-mapped cache
10	Count hits and misses correctly for 2-way associative cache
10	Count hits and misses correctly for 4-way associative cache
10	Count the number of writebacks correctly

6. Miscellaneous

A sample executable and a simple test case are provided for you to match the output format. You need to come up with an extensive set of test cases on your own to validate the behavior of your program. The first person to report an error on the executable earns an extra credit of 5 points. A report must provide a snapshot on the content of input and output, along with an explanation on the reported error. Automatic plagiarism detection software will be used on all submissions – any cases detected will result in a grade of 0 for those involved. One-day late submissions are allowed **at a 15-point penalty**.