CDA 3101: Spring 2019
Project 3 - Cache Simulator

Total Points: 100
Due: Saturday, 04/20/2019, 11:59 PM

1 Objective

The objective for this assignment is to make sure

- You have a thorough understanding of caching and the various methods of implementing caching.
- You have an understanding of the concepts of tags, indices and offsets.
- You have an understanding of cache replacement and writeback policies.
- You can create arrays and iterate through them in C.

2 Project Description

This project requires you to read in a bunch of memory references and simulate the operation of a write-through, no-write-allocate cache and a write-back, write-allocate cache.

2.1 Suggested Approach

For this project, it is suggested you just maintain the references in parallel arrays of chars and ints. You will have at most 100 references. Once the references are saved, build the cache either using parallel arrays or an array of structures. Then, simulate the following two caches.

2.1.1 Write-through, No write allocate Cache

A write-through, no write allocate cache has 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).

2.1.2 Write-back, Write allocate Cache

A write-back, write allocate cache has 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 blocks 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 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 blocks contents must be written back to main memory.

2.2 Input

Your input will be redirected through `stdin`. The firs 3 lines of your input file would be the block size, the number of sets and the set associativity. That will then be followed by several lines (at most 100) of memory references, where each line is of the format:

```
R/W<tab>address
```

Here, R/W will be a single uppercase character (either 'R' or 'W') and the address would be a 32 bit positive integer. You do not need an unsigned int. The largest address you would get is 1073741824 (2^{30}).

Sample Input

```
16
64
2
W 300
R 304
R 4404
W 4408
W 8496
R 8500
R 304
```

2.3 Output

Your output should print to `stdout`. The output should first print the block size, the number of sets and associativity, followed by the number of bytes used for the tag, index and offset. Then, the program should print the statistics for the two types of caches we are simulating, as shown below:

```
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
Memory References: 6
```

3 Dynamic Allocation in C

For this homework, you will be required to allocate dynamic arrays of integers/structures dynamically, since we do not know the size of the caches until we have read in the parameters. For the sake of not being raked over the coals for the C requirement again, here are the ONLY things you need to know about dynamic arrays in C.

3.1 Declaring

You need to use the `malloc` function in the `stdlib` library for this. The syntax for an array of integers is:

```
int * arr = (int *) malloc (size_of_array * sizeof(int));
```

For an array of the `line` structure, which has been `typedef`'ed is:

```
line * arr = (line*) malloc (size_of_array * sizeof(line));
```

You can now use these just like regular arrays, using the `[]` operators.

3.2 Deallocating

To deallocate the memory after you're done using it,

```
free(pointer);
```

Besides this, the only C concepts you will need are: declaring all variables at the beginning of the function (including loop counters), `printf, scanf , fgets` and `atoi`. These projects do not have as steep as a C learning curve as some students seem to believe. You can use almost ALL the C++ concepts you learned in COP 3014 here. C does not have any advanced object oriented concepts you learned in COP 3330, but you do not need any of those for these projects.

4 Testing

Part of the process of developing a good software artifact is thoroughly testing the artifact. The sample test cases, while extensive, do not cover all the possible combination of instructions your program could encounter.

For this project, you are required to develop your own suite of test cases. You can write anywhere between 1 (very long) and 10 (somewhat small) test cases, testing out different cache and block sizes, different levels of associativity, different write policies, etc. The quality of the test suite will be determined by how many of scenarios are tested.

5 Submission and Grading

You are required to turn in the following:

- You C program, called "proj3_LastName.c"
- A suite of test cases that you have generated, beyond the test cases provided, that demonstrate your rigorous testing of your solution.
- A README file listing all the requirements your program can handle, and all known issues with your program (parts not implemented, parts implemented with issues, etc).

Please tar your C program and your test cases into one tarball. Please make sure your tarball does not contain any other files, especially executables.

Submissions may be made through Canvas in the Assignments section. You must submit before 11:59 PM on April 20 to receive full credit. Late submissions will be accepted for 10% off on April 21. The first person to report any errors in the provided materials will be given 5% extra credit. Automatic plagiarism detection software will be used on all submissions any cases detected will result in a grade of 0, reduction of the overall grade by one letter grade, and a report will be sent to Academic Affairs.

Your submitted C program will be compiled and run on `linprog` with the following commands, where `test.in` is a cache trace file, as described below:

```
$ gcc proj3.c -lm
$ ./a.out < test.in
```

You should not rely on any special compilation flags or other input methods, except for including the math library. The grade breakdown for the project is as follows:

- Reading and parsing - 10 points
- Calculating the bits for tag, index and offset - 10 points
- Setting up the arrays for the cache - 10 points
- Implementing Least Recently Used Policy - 10 points
- Managing Associativity - 10 points
- Simulating a write-through, no-write allocate cache - 20 points
- Simulating a write-back, write allocate cache - 20 points
- Quality of the test suite - 10 points

Please do NOT write header files / break your code into several files. This makes it harder to run through plagiarism detection software.

Programs that do not compile will automatically receive a grade of 0.

Prgrams that crash with a segmentation fault will receive a 10% penalty for every test case that crashes.

You are responsible for making sure you turn in the right file and that you haven't tarred over your files. We will not accept any excuses in this regard, this this project is likely to be graded very close to finals.