Final Project

Due Dec 7 by 9:30am **Points** 200

ECE/CS 472/572 Final Exam Project

Submit your final project to TEACH (https://teach.engr.oregonstate.edu) by Fri. 12/07/2018, at 9:30am

Late submissions will not be accepted.

***Your submission should be comprised of two items:** a **.pdf** file containing your written report and a **.tar** file containing a directory structure with your C or C++ source code. Your grade will be reduced if you do not follow the submission instructions.*

All written reports (for both 472 and 572 students) must be composed in MS Word, LaTeX, or some other word processor and submitted as a PDF file.

Please take the time to read this entire document. If you have questions there is a high likelihood that another section of the document provides answers.

Introduction

In this final project you will implement a cache simulator. Your simulator will be configurable and will be able to handle caches with varying capacities, block sizes, levels of associativity, replacement policies, and write policies. The simulator will operate on trace files that indicate memory access properties. All input files to your simulator will follow a specific structure so that you can parse the contents and use the information to set the properties of your simulator.

After execution is finished, your simulator will generate one or more output file(s) containing information on the number of cache misses, hits, and miss evictions (i.e. the number of block replacements). The file will also provide the total number of clock cycles used during the situation.

It is important to note that your simulator is required to make a couple significant assumptions for the sake of simplicity.

1. You do not have to simulate the actual data contents. We simply pretend that we copied data from main memory and keep track of the hypothetical time that would have elapsed.
2. When a block is modified in a cache or in main memory, we always assume that the entire block is read or written. This means that you don't have to deal with the situation where only part of a block needs to be updated in main memory.

3. Assume that all memory accesses occur only within a single block at a time. In other words, we don't worry about the effects of a memory access overlapping two blocks, we just pretend the second block was not affected.

Basic-Mode Usage (472 & 572 students)

Input Information

Your cache simulator will accept two arguments on the command line: the file path of a configuration file and the file path of a trace file containing a sequence of memory operations. The cache simulator will generate an output file containing the simulation results. The output filename will have ".out" appended to the input filename. Additional details are provided below.

```
# example invocation of cache simulator
cache_sim ./cache_config ./tracefile1
Output file written to ./tracefile1.out
```

The first command line argument will be the path to the configuration file. This file contains information about the cache design. The file will contain only numeric values, each of which is on a separate line.

Example contents of a configuration file:

```
8 <-- number of sets in cache (will be a non-negative power of 2)
16 <-- block size in bytes (will be a non-negative power of 2)
3 <-- level of associativity (number of blocks per set)
1 <-- replacement policy (will be 0 for random replacement, 1 for LRU)
1 <-- write policy (will be 0 for write-through, 1 for write-back)
13 <-- number of cycles required to write or read a block from the cache
230 <-- number of cycles required to write or read a block from main memory
0 <-- cache coherence protocol (0 for simple implementation, 1 for MESI, only used for 572 projects)
```

Here is another example configuration file specifying a direct-mapped cache with 64 entries, a 32 byte block size, associativity level of 1 (direct-mapped), least recently used (LRU) replacement policy, write-through operation, 26 cycles to read or write data to the cache, and 1402 cycles to read or write data to the main memory. CS/ECE472 projects can ignore the last line.

```
64
32
1
1
0
26
1402
0
```

The second command line argument indicates the path to a trace file. This trace file will follow the format used by Valgrind (a memory debugging tool). The file consists of comments and memory access information. Any line beginning with the ‘=’ character should be treated as a comment and ignored.

```
==This is a comment and can safely be ignored.
==An example snippet of a Valgrind trace file
I  04010173,3
I  04010176,6
 S 04222cac,1
I  0401017c,7
 L 04222caf,8
I  04010186,6
I  040101fd,7
 L 1ffefffd78,8
 M 04222ca8,4
I  04010204,4
```

Memory access entries will use the following format in the trace file:

```
[space]operation address,size
```

- Lines beginning with an ‘I’ character represent an instruction load. For this assignment, you can ignore instruction read requests and assume that they are handled by a separate instruction cache.
- Lines with a space followed by an ‘S’ indicate a data store operation. This means that data needs to be written from the CPU into the cache or main memory (possibly both) depending on the write policy.
- Lines with a space followed by an ‘L’ indicate a data load operation. Data is loaded from the cache into the CPU.
- Lines with a space followed by an ‘M’ indicate a data modify operation (which implies a special case of a data load, followed immediately by a data store).

The address is represented as a hexadecimal number (up to 64 bits). The size of the memory operation is indicated in bytes (as a decimal number).

If you are curious about the trace file, you may generate your own trace file by running Valgrind on arbitrary executable files:

```
valgrind --log-fd=1 --log-file=./tracefile.txt --tool=lackey --trace-mem=yes name_of_executable_to_trace
```

Cache Simulator Output

Your simulator will write output to a text file. The output filename will be derived from the trace filename with “.out” appended to the original filename. E.g. if your program was called using the invocation “cache_sim ./dm_config ./memtrace” then the output file would be written to “./memtrace.out”

(S)tore, (L)oad, and (M)odify operations will each be printed to the output file (in the exact order that they were read from the Valgrind trace file). Lines beginning with “I” should not appear in the output since they do not affect the operation of your simulator.

Each line will have a copy of the original trace file instruction. There will then be a space, followed by the number of cycles used to complete the operation. Lastly, each line will have one or more statements indicating the impact on the cache. This could be one or more of the following: **miss**, **hit**, or **eviction**.

Note that an eviction is what happens when a cache block needs to be removed in order to make space in the cache for another block. It is simply a way of indicating that a block was replaced. In our simulation, an eviction means that the next instruction cannot be executed until after the existing cache block is written to main memory. An eviction is an expensive cache operation.

It is possible that a single memory access has multiple impacts on the cache. For example, if a particular cache index is already full, a (M)odify operation might **miss** the cache, **evict** an existing block, and then **hit** the cache when the result is written to the cache.

The format of each output line is as follows (and can list up to 3 cache impacts):

```
operation address,size <number_of_cycles> <cache_impact1> <cache_impact2>
```

The next-to-last line of the output file will indicate the number of hits, misses, and evictions. The final line will indicate the total number of simulated cycles that were necessary to simulate the trace file.
These lines should exactly match the following format (with values given in decimal):

```
Hits:<hits> Misses:<misses> Evictions:<evictions>
Cycles:<number of total simulated cycles>
```

In order to illustrate the output file format let's look at an example. Suppose we are simulating a direct-mapped cache operating in write-through mode. Note that the replacement policy does not have any effect on the operation of a direct-mapped cache. Assume that the configuration file told us that it takes 13 cycles to access the cache and 230 cycles to access main memory. Keep in mind that a hit during a **load** operation only accesses the cache while a miss must access **both the cache and the main memory**.

In this example the cache is operating in write-through mode so a standalone (S)tore operation takes 243 cycles, **even if it is a hit**, because we always write the block into both the cache and into main memory. If this particular cache was operating in write-back mode, a (S)tore operation would take only 13 cycles if it was a hit (since the block would not be written into main memory until it was evicted).

The exact details of whether an access is a hit or a miss is entirely dependent on the specific cache design (block size, level of associativity, number of sets, etc). Your program will implement the code to see if each access is a hit, miss, eviction, or some combination.

```
==For this example we assume that addresses 04222cac, 04222caf, and 04222ca8 are all in the same block at index 2
==Assume that addresses 047ef249 and 047ef24d share a block that also falls at index 2.
==Since the cache is direct-mapped, only one of those blocks can be in the cache at a time.
==Fortunately, address 1ffefffd78 happens to fall in a different block index (in our hypothetical example).
==The output file for our hypothetical example:
S 04222cac,1 243 miss
L 04222caf,8 13 hit
M 1ffefffd78,8 486 miss hit <-- notice that this (M)odify has a miss for the load and a hit for the store
M 04222ca8,4 256 hit hit <-- notice that this (M)odify has two hits (one for the load, one for the store)
```

```
S 047ef249,4 473 miss eviction <-- 243 cycles for miss, additional 230 cycles for eviction
L 04222caf,8 473 miss eviction
M 047ef24d,2 716 miss eviction hit <-- notice that this (M)odify initially misses, evicts the block, and then hit
s
L 1ffefffd78,8 13 hit
M 047ef249,4 256 hit hit
Hits:8 Misses:5 Evictions:3
Cycles: 2929 <-- total sum of simulated cycles (from above)
```

Implementation Details

You may use either the C or the C++ programming language. Graduate students will have an additional component to this project.

Your code should make the assumption that memory reads and writes never simultaneously occur across multiple cache blocks. In other words, your simulator can ignore the number of bytes specified as part of the trace file. This makes your work easier since each memory access will either be entirely in the cache or entirely outside of the cache.

In our simplified simulator, increasing the level of associativity has no impact on the cache access time. Furthermore, you may assume that it does not take any clock cycles to access non-data bits such as Valid bits, Tags, Dirty Bits, LRU counters, etc.

Your code must support the LRU replacement scheme and the random replacement scheme. For the LRU behavior, a block is considered to be the Least Recently Used if every other block in the cache has been read or written after the block in question. In other words, your simulator must implement a true LRU scheme, not an approximation.

You must implement the write-through cache mode. You will receive extra credit if your code correctly supports the write-back cache mode (specified in the configuration file).

Acceptable Compiler Versions

The flip server provides GCC 4.8.5 for compiling your work. Unfortunately, this version is from 2015 and may not support newer C and C++ features (especially related to parallel programming). If you call the program using “gcc” (or “g++”) this is the version you will be using by default.

If you wish to use a newer compiler version, I have compiled a copy of GCC 8.2 (released July 26, 2018). You may write your code using this compiler and you’re allowed to use any of the compiler features that are available. The compiler binaries are available in the path:

```
/nfs/farm/classes/eecs/fall2018/cs472/public/gcc/bin
```

For example, in order to compile a multithreaded C++ program with GCC 8.2, you could use the following command (on a single terminal line):

```
/nfs/farm/classes/eecs/fall2018/cs472/public/gcc/bin/g++ -ocache_sim -lpthread -Wl,-rpath,/nfs/farm/classes/eecs/
fall2018/cs472/public/gcc/lib64 my_source_code.cpp
```

Parallel Implementation (required for CS/ECE 572 students)

Implement your cache simulator so that it can support up to 4 simulated processors operating in parallel. Each processor will have its own personal cache. These caches will all share a common bus that is connected to a shared main memory. For our simulations, you can assume that the main memory uses a pipelined implementation and can deliver cache blocks to each processor with the same latency as the single core version.

Cache Operation

Since we are working with multiple processors we now have to consider how we can keep the data consistent across multiple caches. This could be problematic if multiple threads happen to need access to the same data in memory. There are multiple techniques for dealing with this problem. Please read the Wikipedia page for an overview of the issue: **https://en.wikipedia.org/wiki/Cache_coherence (https://en.wikipedia.org/wiki/Cache_coherence)**

In this project you will use a bus-snooping protocol to address the challenge of cache coherence.

You will implement a basic write-invalidate protocol. We will use a simple (inefficient) technique to deal with cache coherence. Basically, each cache is able to monitor the shared bus. If any cache sees write activity on the bus corresponding to a currently cached block, that block is immediately marked as invalid. The next request for that cache block will be a miss (for the snooping caches).

If a snooping cache sees a read request for a cached block, no action is taken (since the read did not invalidate our copy of the data).

With the addition of the coherence challenge, the multi-thread cache simulator provides the same functionality as the single-core version.

572 Extra Credit

If you want to earn extra credit, also implement the MESI cache coherence protocol.
https://en.wikipedia.org/wiki/MESI_protocol (https://en.wikipedia.org/wiki/MESI_protocol)

Note that your cache should support write-back operation in order to implement MESI.

Recall that the final line of the input configuration file indicates the coherence protocol (0 = basic write-invalidate protocol, 1 = MESI protocol). Your code will select the appropriate coherence policy based on this input.

You may assume that communication from cache-to-cache takes the same number of clock cycles as a standard cache hit.

If you implement the MESI protocol make sure that you incorporate the effects of the protocol on the hit and miss times in your output files. In some cases your code might have to receive data from another cache (rather than main memory) if the data has not been written back yet to main memory.

Simulator Operation

Your cache simulator will use a similar implementation as the single-core version but will accept multiple trace files on the command line. Each trace file will be assigned to a specific simulated core. You do not ever have to simulate context switching where a program is moved to a different processor during execution.

This example command line output demonstrates the command line operation:

```
# example invocation of parallel cache simulator
cache_sim ./cache_config ./tracefile1 ./tracefile2 ./tracefile3 ./tracefile4
Output file(s) written to ./tracefile1.out, ./tracefile2.out, ./tracefile3.out, ./tracefile4.out
```

The number of simulated cores is dictated by the number of command line arguments. You will simulate one core for each trace file that is provided.

The output files will contain the same information as in the single-core version. However, if you chose to implement the MESI algorithm the hit and miss times might be impacted (depending on the cache block states). We expect that the hit and miss rates will certainly be affected by the behavior of the parallel threads.

Code Implementation

Since your cache simulator is simulating multiple processors, it seems logical that this is a perfect application for parallel programming in C or C++. Your program must split its tasks into threads. Each hypothetical cache must operate on a separate thread. You may use additional threads to coordinate the simulator's operation.

Even if a single trace file is provided, your code must use multithreading to allocate the simulation to a separate thread. Flip supports 24 virtual cores, feel free to use them!

If you are programming in C, you may use pthreads or other libraries (as long as you can include the libraries in your project directory and it compiles on flip.engr.oregonstate.edu).

If you are programming in C++, I suggest using the threads support that is natively incorporated into C++14. Again, you may use alternate libraries if you are able to incorporate all dependencies into the project directory (and the code compiles on flip.engr.oregonstate.edu)

Project Write-Up

Note: Any chart or graphs in your written report must have labels for both the vertical and horizontal axis.

Undergraduates (CS/ECE 472)

Part 1: Summarize your work in a well-written report. The report should be formatted as a single-spaced document with 12pt text. Use images, charts, diagrams or other visual techniques to help convey your information to the reader.

Explain how you implemented your cache simulator. You should provide enough information that a knowledgeable programmer would be able to draw a reasonably accurate block diagram of your program.

- What data structures did you use to implement your design?
- What were the primary challenges that you encountered while working on the project?
- Is there anything you would implement differently if you were to re-implement this project?
- How do you track the number of clock cycles needed to execute memory access instructions?

Part 2: There is a general rule of thumb that a direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2.

Your task is to use your cache simulator to conclude whether this rule of thumb is actually worth using. You may test your simulator using instructor-provided trace files or you may generate your own trace files from Linux executables (“wget oregonstate.edu”, “ls”, “hostid”, “cat /etc/motd”, etc). Simulate at least three trace files and compare the miss rates for a direct-mapped cache versus a 2-way set associative cache of the same size. For these cache simulations, choose a block size and number of indices so that the cache contains 32KiB of data. Put your simulation results into a graphical plot and explain whether you agree with the aforementioned rule of thumb. Include this information in your written report.

Part 3: If you chose to implement any extra credit tasks, be sure to include a thorough description of this work in the report.

Graduate Students (CS/ECE 572)

Part 1: Summarize your work in a well-written report. The report should be formatted as a single-spaced document with 12pt text. Use images, charts, diagrams or other visual techniques to help convey your information to the reader.

Explain how you implemented your cache simulator. You should provide enough information that a knowledgeable programmer would be able to draw a reasonably accurate block diagram of your program.

- What data structures did you use to implement your parallel cache simulator?
- How did you implement the parallel programming aspect of the project?
- What were the primary challenges that you encountered while working on the project?
- Is there anything you would implement differently if you were to re-implement this project?
- How do you track the number of clock cycles needed to execute memory access instructions?

Part 2:

Using trace files provided by the instructor, how does the miss rate and average memory access time (in cycles) vary when the simulated program is operating in parallel? Note that you can compute the average memory access time by considering the number of simulated hits and misses, along with the total number of simulated cycles. What is the difference between 1, 2, or 4 threads? Compare this information using a chart or plot.

Part 3: If you chose to implement any extra credit tasks, be sure to include a thorough description of this work in the report.

Submission Guidelines

You will submit both your source code and a PDF file containing the typed report.
Any chart or graphs in your written report must have labels for both the vertical and horizontal axis!

For the source code, you must organize your source code/header files into a logical folder structure and create a tar file that contains the directory structure. Your code must be able to compile on flip.engr.oregonstate.edu. If your code does not compile on the engineering servers you will receive a 0 grade for all implementation portions of the grade.

You need to create a Makefile that can be used to compile your project from source code. If you need a refresher, please see **this helpful page (https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html)** . If the Makefile is written correctly, the grader should be able to download your TAR file, extract it, and run the “make” command to compile your program. The resulting executable file should be named: “cache_sim”.

Grading and Evaluation

CS/CE 472 students can complete the 572 project if they prefer (and must complete the 572 write-up, rather than the undergraduate version). Extra credit will be awarded to 472 students who choose to complete this task.

Your source code and the final project report will both be graded. Your code will be tested for proper functionality. All aspects of the code (cleanliness, correctness) and report (quality of writing, clarity, supporting evidence) will be considered in the grade. In short, you should be submitting professional quality work.

Errata

This section of the assignment will be updated as changes and clarifications are made. Each entry here should have a date and brief description of the change so that you can look over the errata and easily see if any updates have been made since your last review.