2019 01 – CS 3853 Computer Architecture

Group Project: Cache Simulator

Final Project Due: Thu, May 2, 2019 7:30 pm

NO LATE ACCEPTED

1 Objectives

The goal of this project is to help you understand the internal operations of CPU caches. You are required to simulate a Level 1 cache for a 32-bit CPU. The cache must be command line configurable to be direct-mapped, 2-way, 4-way, 8-way, or 16-way set associative and implement both round-robin and random replacement policies for performance comparisons.

2 General Project Descriptions

2.1 Groups

You may work in groups of 2 to 3(preferred) students. This project requires coding, testing, documenting results and writing the final report. Everyone must contribute to receive credit. Anyone not contributing will either have to finish their own project by the due date or receive a zero. Note: NO WORK OF ANY KIND IS ACCEPTED AFTER May 2, 2019 at 7:30pm – except the final exam.

2.2 Programming Languages and Reference Systems

You are allowed to use any of the following programming languages: Python, C/C++ and Java.

2.3 Simulator Input and Memory Trace Files

Your simulator will have the following input parameters:

1.	–f <trace file name>	[name of text file with the trace]
2.	–s <cache size in KB>	[1 KB to 8 MB]
3.	–b <block size>	[4 bytes to 64 bytes]
4.	–a <associativity>	[1, 2, 4, 8, 16]
5.	–r <replacement policy>	[RR or RND or LRU for bonus points]

Sample command lines:

```
Sim.exe –f trace1.txt –s 1024 –b 16 –a 2 –r RR
```

That would read the trace file named “trace1.txt”, configure a total cache size of 1 MB with a block size of 16 bytes/block. It would be 2-way set associative and use a replacement policy of Round Robin. We will assume a write-through policy.

2.5 Simulator Outputs

Your simulator should output the simulation results to the screen (*standard out*, or *stdout*). The output should have a short header formatted as follows:

Cache Simulator CS 3853 Spring 2019 – Group #* (where *** is your group number)**

Cmd Line: Reprint the command line used and the parsed parameters:
Trace File: <name of trace file>
Cache Size: <size typed in KB>
Block Size: <size typed in bytes>
Associativity: <direct, 2-way, 4-way, etc.>
R-Policy: <characters typed in>

Generic:	**Example:**	
Cache Size: *** KB	Cache Size: 1024 KB	**← This means 1 MB**
Block Size: *** bytes	Block Size: 16 bytes	
Associativity: ***	Associativity: 2	
Policy: RR or RND or LRU	Policy: RR	
----- Calculated Values -----		
Total #Blocks: ***	Total #Blocks: 64 KB (2^ 16)	
Tag Size: *** bits	Tag Size: 13 bits	
Index Size: *** bits, Total Indices: ***	Index Size: 15 bits, Total Indices: 32 KB	
Overhead Memory Size: ***	Overhead Memory Size: 114,688 bytes (or 112 KB)	
Implementation Memory Size: ***	Implementation Memory Size: 1,163,264 bytes (or 1136 KB)	
----- Results -----		
Cache Hit Rate: *** %	Cache Hit Rate: 96.2 %	

4 Trace Files

I will provide several trace files for testing which will be formatted as shown below. The trace file contains an execution trace from a real program execution. At least one trace file will be very short so that you can manually determine the miss rate.

The trace files provided contain two lines – the instruction fetch line and the data access line. The instruction fetch line contains the data length (number of bytes read), the address (a 32-bit hexadecimal number) and data (the machine code of the instruction).

The data access line shows addresses for destination memory "dstM" (i.e. the write address) and source memory "srcM" (i.e. the read address) and the data read. There is NO indication of the length of data read and thus you should ASSUME all data accesses are 4 bytes.

For the instruction line, you care about the length and the address. For the data line, the length is given as 4 bytes and you need the dst/src addresses. Additionally, if the src/dst address is zero, then ignore it – that means there was no data access.

4.1 Sample Trace File Format:

Below is an example of what two blocks in the trace file look like. EIP is the Extended Instruction Pointer – this identifies the memory that is read containing the instruction. The number in parenthesis (highlighted in green) is the number of bytes read (i.e. the length of that instruction.) The next number, highlighted in yellow is the numeric address containing the instruction.

The next set of hex digits are the actual machine code for this particular instruction – in other words, this is the data read. For our cache simulator, we do not care about the data so it should be ignored.

```
EIP (04): 7c809767 83 60 34 00  and dword [eax+0x34],0x0
dstM: 00000000 --------    srcM: 00000000 --------

EIP (07): 7c80976b 8b 84 88 10 0e 00 00  mov eax,[eax+ecx*4+0xe10]
dstM: 7ffdf034 00000000    srcM: 7ffdfe2c 901e8b00
```

The second line contains "dstM:" followed by an address. If that address is all zeros, then ignore it, otherwise it is the address of a data write. Assume 4 bytes and ignore the data. Then srcM shows the address read by this instruction. Again, if the address is zero, ignore it – no data read occurred and assume 4 bytes and ignore the data.

Note that when an instruction is executed, the dstM is not yet written, so the dstM shown is actually the destination from the prior instruction. For our purposes, ignore that and treat it as an access in the same block in which you read it.

When all said and done, here is the data that should be processed by your cache simulator for the two instructions above.

Address: 0x7c809767, length = 3. No data writes/reads occurred. <There is a data write by the and instruction BUT it is not seen until the next block.>

Address: 0x7c80976b, length = 7. Data write at 0x7ffdf034, length = 4 bytes, data read at 0x7ffdfe2c, length = 4 bytes.

4.2 How to Parse:

The file should be very uniform with the characters are the same line offset for each line. In C/C++, use fgets to read a line into a char array.

```
Line[] = "EIP (04): 7c809767", so line[0] = 'E', line[5,6] = "04", and line[10,17]
= "7c809767". You will have to convert the character representation of the
address into a hex value.
```

5 Experiment Guidelines

Once the simulator is complete, you will need to try different cache parameters and compare results. You may graph them in Excel or some similar software. Below is the minimum required comparisons, but you may do additional ones as desired.

For each trace file provided, apply each associativity with the following parameters:

Cache Sizes: 8 KB, 64 KB, 256 KB, 1024 KB, Block Sizes: 4 bytes, 16 bytes, 64 bytes, Replacement Policy: RR and RND

The total number of simulation runs will be: #trace files * 4 cache sizes * 3 block sizes * 2 replacement policies.

So 24 simulation runs for each trace file. You may automate the execution of your simulator and/or the collation of results. In the report, document the various simulation runs and any conclusions you can draw from it.

THIS IS AN IMPORTANT PART OF THE PROJECT – MAKE SURE TO SPEND SOME TIME ON IT. ANALYSIS WAS VERY WEAK LAST TIME SO I AM GIVING YOU A HEADS-UP.

ALSO, if you fail to get a working cache simulator by the due date, I will let you use mine to get results and write up the report.

6 Deliverables and Submission Guidelines

Hard Copy: Final report detailing the Experimentation Results

Soft Copy: Final Report, All Simulator Source Code, Test files you used

7 Grading

For the code, I will execute your simulator. It's in your best interest to make this as easy as possible for me. For C/C++ projects in Linux, include a makefile. For windows, MAKE SURE to set the multi-threaded debug option in Visual Studio.

I will execute it with several small memory traces to test if it can produce the correct cache miss rates. The memory trace files used in grading have the exact same format as the provided trace files.

The report will be graded based on the quality of implementation, the complexity and thoroughness of the experiments, and the quality of the writing. Individual grade will be negatively affected if a student does not exhibit a fair share of contribution.

8 Submission Schedule

This project is divided into 3 parts with 3 due dates. Each ***due date is very strict*** and a late turn-in results in a ***ZERO*** for that portion.

(30 pts) Milestone #1 – Input parameters and parsing the trace file.

DUE: March 26, 11:59pm. Blackboard.

Upload a .zip file with the following name to blackboard:

"2019_01_CS3853_Team_XX_M#1.zip"

TEN points deducted for incorrect name. XX is your team number.

The .zip file should include a copy of your source code, output files from 3 different runs on Trace1.trc using different parameters each time. The output files should have all the header information printed as described above including a place for the cache miss rate.
Additionally, the first 20 of the addresses and lengths printed in a list formatted like this:

```
0xhhhhhhhh: (xxxx)
```

Where the "hhhhhhhh:" is the hexadecimal address and the (xxxx) contains the length of the read in decimal.

(70 pts) Milestone #2 – The Cache Simulator program.

DUE: April 17, 11:59 pm. Blackboard

Upload a .zip file with the following name to blackboard:

"2019_01_CS3853_Team_XX_M#2.zip"

TEN points deducted for incorrect name. XX is your team number.

The zip file should contain your source code and 5 runs for the "A-9_new 1.5.pdf.trc" file showing your results (the cache miss rate for each run). There is NO REQUIREMENT to output the addresses and lengths at this point. If you want to more easily graph your results, you are welcome to output a second file that has only the results for importation into Excel. If you do that, make sure to include samples of those output files as well.

(50 pts) Milestone #3 – Analysis.

DUE: May 2, 7:30 pm. Blackboard for softcopy. 7:31pm is LATE for the softcopy. ZERO!

Upload a .zip file with the following name to blackboard:

"2019_01_CS3853_Team_XX_M#3.zip"

TEN points deducted for incorrect name. XX is your team number.

The zip file should contain your final source code, all result output files used in the report, all softcopies of sources, and the final analysis report.