2024 08 – CS 3853 Computer Architecture

Group Project: Cache & Virtual Memory Simulator

Final Project Due: Thu, Dec 5th, 2024 11:59 pm

NO LATE ASSIGNMENTS ACCEPTED FOR ANY REASON

1. Objectives

The goal of this project is to help you understand the internal operations of CPU caches and a simple virtual memory scheme. You are required to simulate a Level 1 cache for a 32-bit CPU and map virtual memory to physical memory. Then you will analyze the results and do a performance comparison.

2. Groups

This is a group project for teams of 3, although a team of 2 is acceptable. 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.

3. Programming Languages and Reference Systems

You may use any of the following programming languages: Python, C/C++, or Java.

4. Project Specifications

Assume a 32-bit data bus. The cache must be command line configurable.

- Cache Size: 8 KB to 8 MB in powers of 2
  - 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192
- Block Size: 8, 16, 32, 64 byte blocks
- Associativity: direct-mapped, 2-way, 4-way, 8-way, or 16-way set associative
- Replacement Policy: round-robin OR random
  - Physical Memory: 1 MB to 4096 MB in powers of 2 < 1, 2, 4, 8, … 4096>
- Virtual address space is set at 4GB (32 bits)
- Able to handle up to 3 trace files – each represents a different process

4.1. Simulator Input and Memory Trace Files

- `-s <cache size - KB>  [ 8 to 8198 KB]`
- `-b <block size>  [ 8 bytes to 64 bytes ]`
- `-a <associativity>  [ 1, 2, 4, 8, 16 ]`
- `-r <replacement policy>  [ RR or RND ]` ← Implement one of these
- `-p <physical memory - MB>  [ 1 MB to 4 GB ]`
- `-u <% phys mem used>  [ 0% to 100% ]`
- `-n <Instr / Time Slice  [ 0 to -1 ] -1 = max`
- `-f <trace file name>  [ name of text file with the trace ]`
  - `You must accept 1, 2, or 3 trace files as input`
  - `Each file will use a "-f " to specify it`

4.2. Example

Your program should be a command line/console style program. The following is an example run of the program. This is all one line, but shown below as 2 lines for clarity

```
CacheSim_v2.11.exe -s 512 -b 16 -a 4 -r rr -p 1024 -n 100 –u 75
-f Trace1.trc -f Trace2_4Evaluation.trc –f Corruption.trc
```

This command line would read the trace files named “trace1.trc”, “trace2_4Evalaution.trc”, and “Corruption.trc” configure a 512 KB cache with a block size of 16 bytes/block, 4-way set associative with a replacement policy of Round Robin. Physical memory would be set at 1 GB <equals 1024 MB> (Note: In my sample sim I only accept numbers for cache and physical memory size, so for an 8 MB cache, you would need to enter “-s 8192”). The write policy is irrelevant so just assume memory is kept in sync with the cache.

4.3. Simulator Outputs

Your simulator should output the simulation results to the screen (*standard out*, or *stdout*). The output must be formatted as follows the specifics are based on the example above. None of the trace files will use addresses greater than 0x7FFFFFFF, so our page table(s) can be 512 KB entries. Do not output what is in red.

Cache Simulator CS 3853 Fall 2024 – Group #XX (where “XX” is your group number)

MILESTONE #1: Input Parameters and Calculated Values

```
Cache Simulator - CS 3853 - Instructor Version: 2.11

Trace File(s):
        Trace1.trc
        Trace2_4Evaluation.trc
        Corruption1.trc

***** Cache Input Parameters *****

Cache Size:                     512 KB
Block Size:                     16 bytes
Associativity:                  4
Replacement Policy:             Round Robin
Physical Memory:                1024
Percent Memory Used by System:  75.0%
Instructions / Time Slice:      100

***** Cache Calculated Values *****

Total # Blocks:                 32768
Tag Size:                       15 bits
Index Size:                     13 bits
Total # Rows:                   8192
Overhead Size:                  65536  bytes
Implementation Memory Size:     576.00 KB  (589824 bytes)
Cost:                           $86.40 @ $0.15 per KB

***** Physical Memory Calculated Values *****

Number of Physical Pages:       262144
Number of Pages for System:     196608
Size of Page Table Entry:       19 bits
Total RAM for Page Table(s):    3735552 bytes  == 512K entries * 3 .trc files * 19 / 8
```

MILESTONE #2: - Simulation Results

```
***** CACHE SIMULATION RESULTS *****
***** CACHE SIMULATION RESULTS  *****

Total Cache Accesses:    343626  (312222 addresses)     // # times cache row hit
Instruction Bytes:       725273  SrcDst Bytes: 295084   // it was valid and tag matched
Cache Hits:              333593  // it was valid and tag matched
Cache Misses:            10033   // it was either not valid or tag didn't match
--- Compulsory Misses:   9998    // it was not valid
--- Conflict Misses:     35      // it was valid, tag did not match


***** *****  CACHE HIT & MISS RATE:  ***** *****

Hit  Rate:                       97.0803%    // (Hits * 100) / Total Accesses
Miss Rate:                       2.9197%     // 1 - Hit Rate
CPI:                             4.38 Cycles/Instruction  (238451) // # Cycles/# Instr
Unused Cache Space:              400.25 KB / 576.00 KB = 69.49%  Waste: $60.04
Unused Cache Blocks:             22770 / 32768

// Unused KB = ( (TotalBlocks-Compulsory Misses) * (BlockSize+OverheadSize) ) / 1024
// The 1024 KB below is the total cache size for this example
// Waste = COST/KB * Unused KB
```

MILESTONE #3: - Simulation Results

```
***** PHYSICAL MEMORY SIMULATION RESULTS *****

Physical Pages Used By SYSTEM:  196608     // -u % * total physical pages
Pages Available to User:         65536     // total pages - pages used by system

Virtual Pages Mapped:            312222
        -----------------------------
        Page Table Hits:         311563    // the virtual page is already mapped in the
                                           // page table - a hit!
        Pages from Free:         659       // # times a virtual page is mapped to a
                                           // physical page not currently in use
        Total Page Faults:       0         // # times when no physical page is available
                                           // and must be swapped with one in use

Page Table Usage Per Process:
-----------------------------

[0] Trace1.trc:
        Used Page Table Entries: 137   ( 0.03%)
        Page Table Wasted: 1244858 bytes

[1] Trace2_4Evaluation.trc:
        Used Page Table Entries: 311   ( 0.06%)
        Page Table Wasted: 1244445 bytes

[2] corruption1.trc:
        Used Page Table Entries: 211   ( 0.04%)
        Page Table Wasted: 1244682 bytes
```

5. 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 length of the instruction (number of bytes read), the 32-bit hexadecimal address, the machine code of the instruction, and the human-readable mnemonic.

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. ASSUME all data accesses are 4 bytes.

For the instruction line, you need the length and the address. For the data line, the length is 4 bytes (ALWAYS!) 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. **SPECIAL NOTE**: The destination write actually appears on the next line. If there is a mov [memory address], eax, that destination address will appear on the next line. That doesn't matter. Process each address access independently.

5.1. 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 length of that instruction. The next number, highlighted in yellow is the 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 actually 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 --------          ← IGNORE these!!!

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

The above has 4 address accesses: 4 bytes read at 0x7c809767, 7 bytes read at 0x7c80976b, 4 bytes written at 0x7ffdf034, and 4 bytes read at 0x7ffdfe2c

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 effect and treat it as an access in the same block in which you read it – this will not affect simulation results.

Here is the data that should be processed by your cache simulator for the two instructions above.

Address: 0x7c809767, length = 4. 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 and data read at 0x7ffdfe2c, length = 4.

5.2. How to Parse:

The file is very structured with the characters at 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.  In C/C++, a sscanf("%x"); can do this.
```

5.3. CPI CALCULATION:

We will calculate CPI in the following manner: The data bus is 32 bits wide which means we can access four bytes of data simultaneously. We require clock cycles for instruction fetch/decode, instruction execution, effective address/branch calculation, and memory access. For our simulation, reading memory requires 4 clock cycles while reading the cache requires only 1.

Consider the trace example below with the "AND" instruction. In reality, we have to read the memory at 0x7C809767 to fetch the instruction. Then we must read the memory at [eax+0x34] to get the data, AND it with zero, and then write the result back to memory. Also, the WRITE to memory, from the "AND" instruction is depicted on the next line as "dstM: 0x7ffdf034". We ignore that complexity - process the trace line by line.

So in this example, the "AND" instruction has no data reads or writes, but the "MOV" instruction has both a "dstM:" (destination) and a "srcM:" (source) data access. For the simulation, no need to determine that the "AND" performs a read nor that the dstM: goes with the "AND" instruction. Just read the line and process the addresses.

```
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
```

5.4. CPI determination:

A. Fetch 4 bytes at 0x7c809767
   a. cache hit : 1 cycle
   b. cache miss : (4 cycles * number of memory reads to populate cache block)
      i. number of reads == CEILING (block size / 4)
      ii. the 4 is because the data bus is 32 bits (i.e. 4 bytes)
   c. NOTE: Multiple cache rows per address possible, 'a' and 'b' apply for each cache row accessed
   d. +2 cycles to execute "AND" instruction (do NOT count effective address time here)

B. Fetch 7 bytes at 0x7c80976b
   a. cache hit: 1 cycle
   b. cache miss: (4 cycles * number of memory reads to populate cache block)
   c. SAME NOTE:
   d. +2 cycles to execute "MOV" instruction

C. Write 4 bytes at 0x7ffdf034
   a. cache hit : 1 cycle
   b. cache miss : (4 cycles * number of memory reads to populate cache block)
   c. +1 cycle to calculate effective address

D. Read 4 bytes at 0x7ffdfe2c
   a. cache hit : 1 cycle
   b. cache miss : (4 cycles * number of memory reads to populate cache block)
   c. +1 cycle to calculate effective address

NOTE: Each address is processed the same way! For an instruction, add 2 cycles. Remember, each address can result in multiple cache rows accessed. When we read 4 bytes at x9767 we REALLY access x9767, x9768, x9769, and x976A. IF we had an 8-byte block, one cache row would be accessed by x9767, and a second cache row would be accessed by x9768 – x976A.

6. Experiment Guidelines

Once the simulator is complete, you will need to simulate multiple 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 for simulation, 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 minimum 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 – WILL REQUIRE SOME TIME

ALSO, if you fail to get a working cache simulator by the Milestone #2 due date, I will let you use mine to get results and write up a detailed simulation report. We'll discuss requirements if this becomes necessary.

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 C/C++ on 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.

(40 pts) Milestone #1 – ALL Input parameters, Calculated Values, and parsing the trace file.
DUE: Sun Nov 3rd, 11:59pm.
Upload a .zip file with the following name to Canvas:

"2024_08_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 and 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 in Section 2.5 (**MILESTONE #1: Input Parameters and Calculated Values**) and be named "Team_XX_Sim_n_M#1.txt", where n = 1, 2, 3.

(70 pts) Milestone #2 – The Cache Simulator program
DUE: Sun Nov 17th, 11:59 pm.
Upload a .zip file with the following name to Canvas:

"2024_08_CS3853_Team_XX_M#2.zip"

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

The zip file should ONLY contain your source code and the output for 3 runs for the "A-9_new_1.5.pdf.trc" file showing your results as shown in Section 2.5 (**MILESTONE #2: - Simulation Results**) and be named "Team_XX_Sim_n_M#2.txt", n = 1,2,3. INCLUDE Milestone #1 Calculations as well. Do NOT include trace files in .zip file nor the entire project folder – doing so costs 10 points.

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.

(80 pts) Milestone #3 – The Virtual Memory Simulator program + Final Report.
DUE: Thu Dec 5th, 11:59 pm. --- NOTE: Use ONLY the 3 large Trace Files for Analysis
Upload a .zip file with the following name to Canvas: TEN points deducted for incorrect name.

"2024_08_CS3853_Team_XX_M#3.zip" XX is your team number.

The zip file must contain the final analysis report.

For the report, assume you are making the recommendation to management as to which cache to implement on a new chip based on these trace results. Consider the miss rate, CPI, cost, and how much of the cache is unused (number of blocks never populated). There may be a range of recommendations for different costs/performance. Also suggest an amount of physical memory that would be optimal. You need to run the simulator using multiple configurations to try to hone in on the properties that will balance cost/performance.