CSCI-2500 Group Project: Instruction Pipeline and Cache (IPLC) Simulator

Christopher D. Carothers
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street
Troy, New York U.S.A. 12180-3590

November 20, 2020

DUE DATE: 11:59 p.m., Wednesday, December 9th, 2020

1 Overview

For this GROUP assignment (upto 4 students per team as before) you will be implementing and comparing various implementations of a combined instruction pipeline and cache (IPLC) simulator. In particular, your group will be determining what design yields the best overall performance (i.e., fewest cycles). For the pipeline/cache simulation part, you will be given a template which you'll need to extend, that contains the following set of files.

- **mm-swap-64.trace:** This is an instruction trace file. The format is described below.
- **iplc-sim.c:** template for the simulator code.
- **Makefile:** builds the iplc simulator.
- **taken-2-2-2.out:** trace-file output for the case described below. Used to help you debug your implementation.
- **taken-2-4-4.out:** additional trace-file output used to help you debug your implementation.

NOTE: You must develop and run your code on the CompOrg Linux machine, `csci-2500.cs.rpi.edu`**!**. The template is located on `kratos.cs.rpi.edu` under:
`/tupper/csci_2500/CLASS_MATERIALS/group-project-template.tar.gz`.

```
0x00400000  lw $4, 0($29) 0x7ffff308
0x00400004  addiu $5, $29, 4
0x00400008  addiu $6, $5, 4
0x0040000c  sll $2, $4, 2
0x00400010  addu $6, $6, $2
0x00400014  jal 0x00400024
0x00400024  addi $29, $29, -4
0x00400028  sw $31, 0($29) 0x7ffff304
0x0040002c  lui $4, 4097
0x00400030  lui $1, 4097
0x00400034  ori $5, $1, 512
0x00400038  ori $6, $0, 8
0x0040003c  jal 0x00400264
0x00400264  ori $8, $0, 4
0x00400268  add $9, $0, $0
0x0040026c  beq $9, $8, 180
0x00400270  ori $10, $0, 2
0x00400274  beq $10, $8, 164
0x00400278  add $11, $9, $9
0x0040027c  add $11, $11, $11
0x00400280  add $11, $11, $11
0x00400284  add $11, $11, $11
0x00400288  add $12, $10, $10
0x0040028c  add $12, $12, $12
0x00400290  add $11, $11, $12
0x00400294  add $11, $11, $4
0x00400298  lw $13, 0($11) 0x10010008
```

Figure 1: Example instruction trace from the MIPS 2-D Matrix Swap and Multiple Program. There are 150,518 instructions in this trace!

1.1 Instruction Trace Format

Figure 1, provides a example of the nearly 35,000 plus MIPS instruction trace. This trace is from a 2-D Matrix Swap and Multiple MIPS code.

The format for each instruction is as follows. First, each instruction occupies a single line of text. The first field is the memory address at which the instruction is located. Next, follows the MIPS instruction name (e.g., `lw` for Load Word followed by the registers and or offset that might be used as part of the instruction. After the instruction has ended and the address that follows is the absolute memory DATA address that was read or written by that instruction if it accessed memory. Instructions that do not access memory will not have a DATA address following.

NOTE: I have provided you with the code necessary to parse this instruction trace file. So you don't need to worry about that part.

1.2 Cache Details

Using this cache simulator and INSTRUCTION and DATA address trace (which the cache simulator skeleton code reads in for you), your overall goal is to find the optimal values for (1) block size, and (2) associativity such that the miss rate of the cache is minimal, yet the total size of the cache must **be less than 10240 bits**. Notice the total size of the cache is expressed in "bits", NOT "bytes"!!). Possible cache configurations are the following:

- **Possible Block Sizes:**
  - 1 word (32 bits)
  - 2 words (64 bits)
  - 4 words (128 bits)
  - 8 words (128 bits)
- **Levels of Associativity**
  - direct mapped (1-way set associative)
  - 2-way set associative with LRU replacement
  - 4-way set associative with LRU replacement
  - 8-way set associative with LRU replacement

Cache Size Function: You can express the size of the cache in terms of 3 factors:

- BlockSize
- Associativity
- IndexBits

$$CacheSize = Associativity*(2^{IndexBits}*(32*BlockSize+33-IndexBits-BlockOffSetBits))$$

where $BlockOffSetBits$ is the Log_2 of the BlockSize, which can be computed as follows:
$BlockOffSetBits = NaturalLog(4*BlockSize)/NaturalLog(2)$

Using the above equation (which has been coded for you in the cache skeleton), you need to find the largest value for the index such that it results in a total cache size that is less than or equal to maximum cache size of 10,240 (10k) bits FOR EACH combination of blocks size and level of associativity. Thus, there will be 16 value combinations of index size, block size and level of associativity. Additionally, your simulator will consider if the branch predictor should predict "taken" or "not taken". Thus, overall, you will have 32 different cache and branch prediction scenarios your cache simulator should compute and determine which one provides the best overall performance in terms of the fewest number of cycles to execute the instruction trace provided.

Given the above scenarios, build a "cache" data structure that is dynamic (i.e., use malloc). Meaning, I do not want to see 16 hard coded cache data structures that correspond to the above 16 different configurations.

You are to assume that on reads and writes, the data is brought into the cache – i.e., to simplify things, you will have a "write/miss" in this cache.

In order for you to check yourself, I have provided the pipeline trace and cache performance output for the 2 bits of cache index, 2 word block size, 2-way set associative, branches taken case. Note, this case is not the largest cache size you could have had and is solely to help you debug your instruction trace simulator.

As you will see, for this very small cache (only 736 bits) the cache miss rate is high (i.e., > 13%). You are allowed to have a MAX cache size of 10240 bits. So, you need to determine the 16 possible cache configurations using the template such that each configuration is the maximum index (number of cache lines) for a given block size and level of associativity. Equally, if your cache simulator does not get this precise answer, then your cache simulator is incorrect and you will not get the performance in cycles correct.

1.3 Pipeline Details

The pipeline is the standard 5-stage design consisting of FETCH, DECODE, ALU, MEM and WRITEBACK stages. The design features you are to consider are:

- One a cache instruction MISS, the pipeline FETCH stage incurs a **stall penalty** of 10 clock cycles.
- One a data word MISS (LW/SW instruction) is also a 10 clock cycle **stall penalty** in the MEM stage for LW and WRITEBACK stage for SW.
- Determine which branch prediction method yields better performance, TAKEN or NOT-TAKEN?
- Last, you are to assume "forwarding" of registers across stages to avoid pipeline stalls.

Note, you will know if a branch is taken because you can see if the address for the next instruction is 4 bytes larger or not. That is, if $PC + 4$ was not the next instruction, then you know the branch was TAKEN!!. Thus, if your predictor guesses wrong, you just need to insert the NOP instruction into the pipeline but do not count that as a real instruction but count the cycles.

The forwarding logic can be found in the textbook Section 4.7. However, your logic is much easier. You will know what instruction you are processing as opposed to particular control signal lines. For example, if either `Reg1` or `Reg2` being used in the DECODE stage are the `Dest_Reg` in the ALU, or MEM stages, you can FORWARD either `Reg1` or `Reg2` to avoid the stall. That is when you detect this condition, you'll allow the pipeline to continue without stalling (e.g., adding delay cycles to overall cycle count). Also note that if a `Dest_Reg` in the WRITEBACK stage is used in the DECODE stage, then no forwarding is needed since the write occurs in the first half of the clock cycle and the read occurs in the last half of the clock cycle for that pipeline stage.

The code you will write for this functionality are detailed in comments provided in the template code `iplcsim.c` which I will cover in class.

2 Performance Evaluation

You will run your simulator 32 times – 16 cache configurations plus branch predictor configured TAKEN and 16 cache configurations plus branch predictor configured NOT-TAKEN.

Show the summary output for all your runs and indicate which configuration performed best for this instruction trace file.

3 Grading Criteria

The following is how these projects will be graded.

1. The project is viewed as 3 major parts: Cache Simulator, Pipeline Simulator, Performance Evaluation. Each of these count 30 points. You should provide good comments in your code so we can follow what you are doing – this is especially true if something does not work right. You can at least explain what you where trying to accomplish with your comments.
2. Write-up and documentation (detailed below) is worth 10 points.
3. If you have a logic error with your cache or pipeline simulator, you will only be deducted points from that part. So, suppose your cache simulator does not work well at all, then you could obtain a score of 70 points if all other parts are done well.
4. **If your simulator seg faults or dumps core due to an error, the most points you can obtain is 10**. Having code written that does not work correctly hurts you way more than having much less code that actual does useful operations.

4 Report Write-up Instructions

Provide a write-up that summarizes your implementation. Describe how each team member contributed to the overall project. Please note, that while not all team members need to be involved in the final project write-up process, it is expected that ALL team members participate and contribute to the design and implementation process. This part of the project document should come first. Then you should describe the output findings for each cache/branch predictor configuration and then present which configuration has the best performance (e.g., fewest cycles).

Turn in a hard copy of your project report and your iplc-sim.c to Submitty using the Project gradable by the appointed deadline. Submitty will not be used to auto-grade your cache simulators.