Valerie Kristofic, Elena DeJaco, Darren Mascioli

vak34, ekd17, dam253

Dr. Rami Melhem

COE 1541

2 October 2018

Branch Prediction and Superscalar Analysis

Overall for the modified five\_stage.c, branch prediction method 1 decreased the number of cycles needed to run a trace. This was supported by running all the various-sized trace files and noting a pattern in their results. With prediction method 0, the number of cycles needed to fully run through the trace for sample.tr is 1092. With prediction method 1, the number of cycles is 1073. For sample1.tr with prediction method 1, it required 1127743 cycles. With prediction method 0 it required 1137056 cycles. This pattern was seen throughout the rest of the medium-length traces as well. For sample\_large1 with prediction method 1, it required 102394047 cycles. For prediction method 0 the file required 105277704 cycles. The other large trace file showed a similar pattern. We believe branch prediction method 1 has superior performance because it was able to dynamically adjust to the tendencies of the code, and therefore had less missed predictions. For example, if there was a large loop in the code, the pipeline would only predict the wrong branch twice instead of every time that prediction method 0 would do.

For superscaler.c, having two pipelines greatly reduced that number of cycles required. For example, running sample.tr though the superscaler design only required 527 cycles. This is almost half as many as required in the single pipeline. The results are similar for the other traces, with almost half as many cycles required to complete the simulation of the trace.

In five\_stage.c for prediction method 1, we represented the hash table as a char array and made a function to update the result of the branch in the ID stage. This was done through a function that checked if the branch prediction was correct by looking at the next instructions in the pipeline (in the IF stage). If the prediction was incorrect, we moved the IF instruction into the prefetch queue and inserted a ‘squashed’ (no-op) instruction into the IF stage.

In the superscaler, we had basically two of everything. We also added in a packing buffer that went in between the prefetch queue and the pipeline. We did something similar with inserting squashed instructions and no-ops as we did in the five\_stage pipeline. We tried to fill the packing buffer from the prefetch queue first, because if there were instructions there it means we had inserted a no-op. If the prefetch queue was empty, we try to pack in a new super-instruction from the trace. We check for data hazards before we pack, and check for control hazards in the IF stage. We did this because we found that moving instructions back to the prefetch queue was easier than moving other instructions forward through the pipeline.

Milestone 1 write up, for reference.

Test Implementation Methods Project One

For testing this milestone, we made small trace files using the given trace\_generator.c file. The purpose of each trace was to act like a unit test - focus on one, very specific functionality. All possible combinations were tested of branch taken and not taken, and for mode 1 prediction correct or incorrect, as well as testing data hazards for each mode. The results of running our modified pipeline on the traces files is shown on page 2, with the corresponding file name in a comment above the output.

All prediction mode 0 traces consist of 2 instructions. File branch\_not\_take\_mode\_0.tr shows that when the PC of the instruction following a branch instruction is not the target address of the branch instruction, no no-op is inserted. The file branch\_taken\_mode\_0.tr shows that when the PC of the instruction following a branch instruction ***is*** the target address of the branch instruction, a no-op is inserted.

All prediction mode 1 traces consist of 4 instructions and have the combinations of branch taken/not taken, and prediction correct/not correct. File branch\_not\_taken\_mode\_1\_correct\_prediction.tr sets up the prediction hash table to have the branch not taken, and then runs the PC sequence again and no no-op is inserted. File branch\_not\_taken\_mode\_1\_incorrect\_prediction.tr sets up the prediction hash table to have the branch not taken, and then runs so that the branch is taken, and a no-op is inserted. File branch\_taken\_mode\_1\_correct\_prediction.tr sets up the prediction hash table to take the branch, a no-op is inserted, and then the branch is taken again with no no-op. File branch\_taken\_mode\_1\_incorrect\_prediction.tr sets up the prediction hash table to take the branch, a no-op is inserted and then the branch is not taken, and an additional no-op is inserted.

For the data hazard traces, the outcome is the same for each mode. The files d\_hazard.tr in mode 1 and in mode 0 have a set of instructions that will not trigger a data hazard, and therefore no no-op is inserted, and a set of instructions that trigger a data hazard and the instructions are inserted.

// branch\_not\_taken\_mode\_0.tr

[cycle 5] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 6] RTYPE: (PC: 8)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

//branch\_taken\_mode\_0.tr

[cycle 5] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 6] NOP:

[cycle 7] RTYPE: (PC: 123456789)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

//branch\_not\_taken\_mode\_1\_correct\_prediction.tr

[cycle 5] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 6] RTYPE: (PC: 8)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

[cycle 7] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 8] RTYPE: (PC: 8)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

//branch\_not\_taken\_mode\_1\_incorrect\_prediction.tr

[cycle 5] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 132456789)

[cycle 6] RTYPE: (PC: 8)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

[cycle 7] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 8] NOP:

[cycle 9] RTYPE: (PC: 123456789)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

//branch\_taken\_mode\_1\_correct\_prediction.tr

[cycle 5] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 6] NOP:

[cycle 7] RTYPE: (PC: 123456789)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

[cycle 8] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 9] RTYPE: (PC: 123456789)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

//branch\_taken\_mode\_1\_incorrect\_prediction.tr

[cycle 5] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 6] NOP:

[cycle 7] RTYPE: (PC: 123456789)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

[cycle 8] BRANCH: (PC: 4)(sReg\_a: 1)(sReg\_b: 2)(addr: 123456789)

[cycle 9] NOP:

[cycle 10] RTYPE: (PC: 8)(sReg\_a: 4)(sReg\_b: 5)(dReg: 6)

//d\_hazard.tr in mode 1

[cycle 5] RTYPE: (PC: 0)(sReg\_a: 1)(sReg\_b: 2)(dReg: 3)

[cycle 6] LOAD: (PC: 4)(sReg\_a: 3)(dReg: 1)(addr: 0)

[cycle 7] LOAD: (PC: 4)(sReg\_a: 1)(dReg: 3)(addr: 123456789)

[cycle 8] NOP:

[cycle 9] RTYPE: (PC: 8)(sReg\_a: 3)(sReg\_b: 2)(dReg: 1)

//d\_hazard.tr in mode 0

[cycle 5] RTYPE: (PC: 0)(sReg\_a: 1)(sReg\_b: 2)(dReg: 3)

[cycle 6] LOAD: (PC: 4)(sReg\_a: 3)(dReg: 1)(addr: 0)

[cycle 7] LOAD: (PC: 4)(sReg\_a: 1)(dReg: 3)(addr: 123456789)

[cycle 8] NOP:

[cycle 9] RTYPE: (PC: 8)(sReg\_a: 3)(sReg\_b: 2)(dReg: 1)