Valerie Kristofic, Elena DeJaco, Darren Mascioli

vak34, ekd17, dam253

Dr. Rami Melhem

COE 1541

4 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.

To create the predictor in five\_stage.c for prediction method 1, we represented the hash table as a char array and built 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 back into the prefetch queue and inserted a ‘squashed’ (no-op) instruction into the IF stage.

For superscaler.c, having two pipelines greatly reduced that number of cycles required. This is obviously because we can do two things at once in the best possible case. For example, running sample.tr through the superscaler design only required 527 cycles. This is almost half as many as required in the single pipeline. This is due to the fact that this sample trace used a lot of Load and Store instructions. These instructions have their own pipeline in this simulation and can run simultaneously alongside non data-dependent instructions. The results are similar for the other traces, with almost half as many cycles required to complete the simulation of the trace. The performance increase is greater the higher percentage of these load and store instructions are in the trace.

In the superscaler, we have basically two of everything. This parallel execution is the basis of this huge cycle gain. To implement this structure, we added in a packing buffer that went in between the prefetch queue and the pipeline. It held one superscalar instruction. When a control hazard was found between the ID stage and this packing buffer, we would insert a NOP superscalar instruction in the packing buffer, and move the instructions that were just in the packing buffer back to the prefetch queue. This prefetch queue held two super scalar instructions for this reason. If two control hazards are found back to back in implementation, both superscalar instructions in the prefetch queue will need to be filled.

When packing instructions, 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 or were unable to pack the two previous instructions together. 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 if one is found the second trace instruction will be written to the priority queue. Before moving this packing buffer to the IF stage, we 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.