Comparison of Local Branch Prediction to Correlating Branch Prediction

Juliette Regimbal Charbel El Hachem

April 12, 2019

1 Introduction

Branching is a source of delays in processors. In the instruction fetch (IF) stage of the processor pipeline, the processor must decide whether to increment the program counter (PC) as normally or to adjust by the offset specified in the branch to a different instruction. The problem arises because it is unknown which action is correct until the following instruction decode (ID) stage. If the wrong action was taken, then the instruction loaded in IF at that point is not to be executed and another cycle is required.

Speculative execution is implemented in modern pipelined processors to prevent these delays. Branch prediction is a common optimization solution where information is gathered dynamically during program execution, more specifically during branching which is then utilized to make a "best guess" of the branch outcome, ideally better than the 50% accuracy guaranteed by random chance predictions. This is because there are typically patterns in execution that can be known before ID and are related to the result of that branch condition. However the complexity and entropy of those patterns will be dependent on the program itself. There are various schemes that are congruous to certain program structures which can identify those patterns and exploit them to improve the prediction accuracy of branch instructions.

This paper focuses on two prediction schemes, local n-bit branch predictors and the (m, n) correlating branch predictors with global history. An n-bit branch predictor is an n-bit counter indexed by the address of a branch instruction. This counter has 2^n possible states, half of which represent a "branch not taken" prediction and half of which represent a "branch taken" prediction. Saturating arithmetic is used to increment or decrement the value if a branch is resolved to be taken or not taken respectively.

An (m, n) correlating predictor with global history adds extra information by tracking the last m branches performed. This creates a two-dimensional array of n-bit predictors as described previously that are indexed by both the address of the branch instruction and the m previous branches. This means that what would be a single predictor in the local n-bit predictor approach is expanded to be 2^m predictors based on the history of all branches.

2 Approach

EduMIPS64 version 0.5.3 is used as a tool to test the performance of these different branch predictors. EduMIPS64 is a Java simulator of a MIPS64 processor. It is used to run the various predictors on different benchmarks to calculate the accuracy of the predictors in each. These results are then used to make conclusions on the suitability of these different branch prediction methods.

EduMIPS64 was modified to add additional branch prediction modes instead of the default mode of "assume not-taken." Specifically modes needed to be added for local n-bit branch prediction and for (m, n) correlating branch prediction. These predictions would need to be made in the IF stage, modifying the program counter if the branch is predicted taken. Then in the ID stage the branch condition must be resolved. In the case where the branch is mispredicted, the CPU must suffer a penalty as the instruction in the IF stage at that time is incorrect.

The instruction address must be fetched in order to index the predictor to make a prediction. In the case of an (m, n) correlating branch predictor, branch history must be accessed to index the correct n-bit predictor. When the branch condition is resolved, it is required to have access to the n-bit predictor and the branch history register (if applicable) to update both with the result of the branch. For the predictor this is simply saturating arithmetic. For branch history, this is shifting the bits left, adding one if the branch was taken, and then masking so only the m least significant bits are kept.

3 Implementation

In EduMIPS64, all branch instructions extend the class `FlowControl_IType`. Two functions were added to this class: `makePrediction` and `respondToCondition`. In the IF stage, the `makePrediction` function is called by specific branch instruction object with `OFFSET_FIELD` as a parameter if it is different for that class of branch than the default of 2 in `FlowControl_IType`. This function gets the prediction result using the `getPrediction` method of the `CPU` object. If this prediction is true — that is the branch is predicted to be taken — the program counter is modified by the offset for the instruction and the old program counter value is saved in a new branch program counter register in case it is needed later. Otherwise no action is taken.

In the ID stage, the `respondToCondition` function is called by the specific branch instruction object with the result of branch resolution and the offset as parameters. The result is compared to the prediction made in IF. If the result and prediction match — the branch was predicted not taken and the branch is not to be taken, or the branch was predicted taken and is to be taken — then no action is taken. In the event the branch was predicted not taken and is supposed to be taken, the instruction then in IF is marked as incorrect by throwing a `MispredictTakenException` after applying the offset to the program counter. This is the same behaviour as taken by default in the unmodified version of EduMIPS64. In the event the branch was predicted taken and is not supposed to be taken, the program counter is reset to the value saved in the branch program counter in the IF stage so execution can restart from the correct location. A `MispredictTakenException` is also thrown here since the instruction in IF is again incorrect. In all cases, the predictor and branch history are updated through the `updatePrediction` method of the `CPU` class.

The `CPU` helper functions `getPrediction` and `updatePrediction` get the value of a prediction for a certain instruction and update the predictor information based on the result of that prediction respectively. These functions both take the instruction object itself as an argument. This is to get the instruction address, as the `Memory` class's function `getInstructionIndex` will return the index of the instruction object, and the address of the instruction is defined as being that index multiplied by four. The branch history is used to get the correct n-bit predictor so the prediction can be known. For the `getPrediction` function, this value is simply returned. For the `updatePrediction`, the second parameter of the function which is a boolean value, representing if the branch was ultimately taken or not, is used to modify the n-bit predictor. It is incremented with saturating addition in the case the branch was taken, and decremented if it was not. The branch history is shifted and the new value appended.

The branch history register is implemented as an integer. When updated as a result of `updatePrediction` being called (it is called every time a branch is resolved), it is shifted left, has one added to it if the branch was resolved to be taken, and then is masked so that only the least significant m bits remain. This implements the tracking of the last m branches. The pattern history table is implemented as a simple two dimensional array of size 12×2^m. The 2^m dimension comes from the 2^m possible states the branch history register can take. 12 comes from the the address length for instructions in EduMIPS64 as found in `CPU.java`. The maximum size is used to ensure that each branch instruction accesses a predictor not used by any other branch instruction. When collisions occur between instructions branch predictor performance suffers, and the decision of how many bits to index from the instruction address is a design decision out of the scope of this paper, however it is obvious to us that a pattern history table collision penalty would be less significant compared to a branch history table collision due to the second dimension of predictors indexed by the history register.

4 Experimental Setup

The values of m and n were varied and `CPU.java` and `GUIStatistics.java` were modified to track the total number of correct predictions and the total number of predictions made. This is done to display the percent of correct predictions made for each benchmark. Note that an n-bit local predictor is equivalent to a (0, n) correlating branch predictor. Of the provided benchmarks, `compitomin.s`, `vet20parinum.s`, and `isort.s` were used as they are programs with varying numbers of branches per run. There are 42, 134, and 163 branches in these benchmarks respectively. Additionally another benchmark — `matrixmult.s` — was developed that calculates the tenth power of a 2×2 matrix. This benchmark has a large number of branches (221) during execution and matrix multiplication is a common computer operation. Due to the structure of matrix multiplication the recurrences and correlation of the numerous branches used in the computations will be exploited by our implementation of the correlated predictors which uses a global branch history register.

After each run, the predictors were reset to soft not taken such that any branch taken with that predictor would cause it to switch to predicting soft taken. The branch history

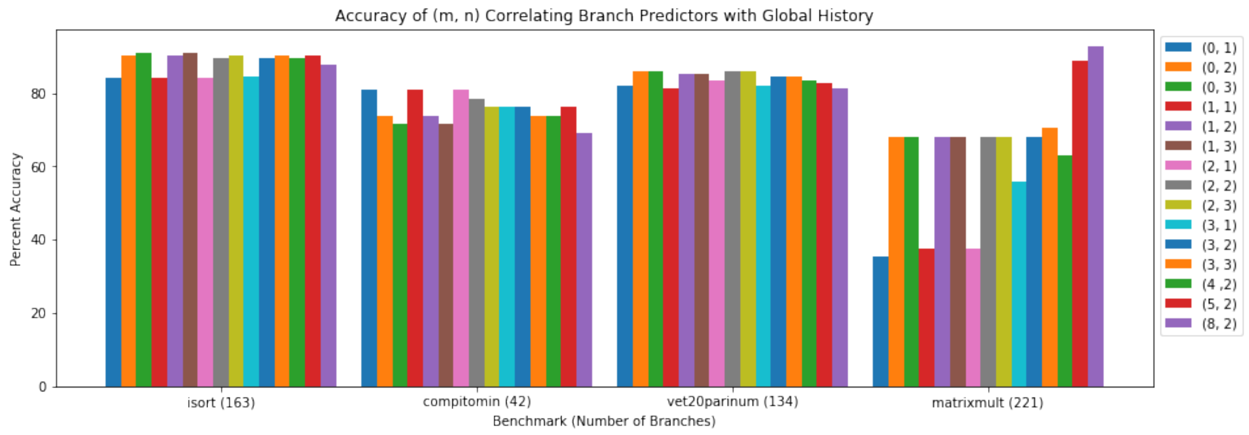

Figure 1: Branch prediction accuracy for different (m, n) correlating branch predictors grouped by four different benchmarks.

was reset to be half taken and half not taken. In total 60 trials were run with 15 predictors to evaluate and 4 benchmarks per predictor.

5 Results

All four benchmarks were run for $0 \leq n \leq 3$ and $1 \leq m \leq 3$ and additionally for three more conditions where $n = 2$ and $m = 4, 5, 8$ to further explore the effects of increasing the size of the branch history. The number of branches were compared to what was expected in each run to verify that execution occurred properly. The outcome of this experiment can be seen in Figure 1.

6 Conclusion

Across the different values of m and n we see a few general trends. First, we see that for short programs (i.e. `compitomin.s`) with fewer branch instructions a significant amount of time is spent setting up the predictors. This leads to somewhat worse performance with larger values of both m and n, although this performance appears to have a lower bound as with larger values of m and n the change in accuracy penalty decreases. Secondly we observe that with programs of longer length than this there is a significant increase in performance going from $n = 1$ to $n = 2$, but a much smaller increase (if any) from $n = 2$ to $n = 3$. This can be seen in the `isort.s`, `vet20parinum.s`, and `matrixmult.s` benchmarks. This suggests higher increases in n will have little benefit, especially for correlating predictors and not branching predictors.

Finally we see a clear benefit in (m, n) correlating branch predictors with a global history compared to local n-bit predictors as the number of branches increase and the patterns get longer, while generally any decrease in performance from larger values of m seem to be relatively small overall for programs with fewer branches. Specifically we see the (8, 2) correlating branch predictor with global history for `matrixmult.s` reaching 92.8% prediction accuracy, far higher than the 2-bit local predictor that achieved only 67.9% accuracy. Meanwhile in programs with a profile like `isort.s`, with $n = 2$ the increase from $m = 0$ to $m = 8$ causes a decrease of only 2.5 percentage points, potentially well worth the increase of 24.9 percentage points that occurs in program with a longer execution.

It is worth noting that which kind of predictor — n-bit local or (m, n) correlating with global branch history — is best heavily depends on what the common case is. This would vary from application to application. A CPU intended to run many short applications with a variety of different profiles may benefit from the shorter initialization of predictors that comes with a smaller n-bit local predictor. Meanwhile a CPU with an expected use in performing linear algebra would benefit heavily from an (m, n) correlating predictor with a large value of m.

While we were successful in implementing the correlating branch predictor we found that even relatively small changes to the pipeline could result in unexpected behaviours if we were not cautious about how we modified the control flow. We also need to keep in mind that by using an emulator we are unable to evaluate the overhead caused by implementing those predictors of varying time and space complexity into the pipeline. Implementing these predictors in hardware prototypes would be the next step taken for more accurate results.