# Project 1: Branch Prediction

Maxim Kovalev (mkovalev@andrew.cmu.edu),
Mridula Chappalli Srinivasa (mchappal@andrew.cmu.edu)
Department of Electrical and Computer Engineering
Carnegie Mellon University Silicon Valley Campus
Moffett Field, CA 94043

February 2014

**Abstract**

In this project we have implemented the algorithm for branch predicting used in Alpha 21264 CPU and employed several techniques to improve its performance. Overall, the misprediction rate of 4.1662% has been achieved.

## Contents

## 1 Introduction

As modern superscalar processors are moving towards deeper pipelines to increase performance, branch prediction has become an important problem. Branch predictor is a circuit that predicts the branch outcome. Branch predictors improve performance in a piplelined processor design as they try to avoid pipeline stalls.

Without branch prediction, the processor would have to wait until the branch instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline. The branch predictor attempts to avoid this waste of time by trying to guess whether the conditional jump is most likely to be taken or not taken. The branch that is guessed to be the most likely is then fetched and speculatively executed. If it is later detected that the guess was wrong then the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay.

The delay caused by branch misprediction is equal to number of stages in the pipeline from fetch to execution stage.This could significantly affect performance in processors having deeper pipelines.Hence, there is a need to minimize the branch misprediction rates. Several branch prediction techniques have been devised which include static and dynamic prediction techniques. In this project, we implement a branch prediction algorithm based on the tournament branch prediction algorithm.

## 2 Runscript

In this project we need to calculate the performance over the whole set of traces, stored in separate files. As it turns out, `zcat` adds `EOF` in the end of each file dump, which is why `zcat ../traces/* | ./predictor` runs on the first file only. In addition, the simulator (`main.o`) does not make use of multiple processor cores, which significantly increases the time needed to test the predictor.

To address these two problems, we have developed a Python script that reads traces, feeds them the the predictor, and then parses its output. All this is done in multiple parallel processes, thus working as fast as the host system allows.

To evaluate the overall performance, it multiplies the misprediction rate by the number or branches in the corresponding trace. Then it sums up these numbers for all the traces and divides by the total number of branches.

# 3 Branch Prediction Algorithm

In this work, in all the variations implemented, we use the concept of tournament predictor, where two different predictors operate simultaneously, and the one performing better so far is chosen to give the final prediction. These two predictors are the local predictor, that bases its predictions upon the history for each specific branch separately, and the global predictor, that uses the overall history of branch taking indiscriminately. Within this framework we then employed several approaches to fine tuning of different parameters in order to achieve better performance than that of the baseline design.

## 3.1 Baseline Design

### 3.1.1 Global Predictor

Global predictor bases its predictions upon the history of whether the last 12 times branches were taken or not. To store this history it uses a 12-bit global history register, where each bit signifies corresponds to one taken/not taken event. Upon each new branch, the value in this register is shifted to the left by one bit, and if the branch was actually taken, the least bit is set to 1; otherwise it's set to 0.

These 12 bits are used to address an array of 4096 values, that are 2-bit saturation counters. Each time the branch is taken, the corresponding (according to the global history register) counter is incremented if it's less than $11_2$. Each time the branch was not taken, it is decremented if it is more than $00_2$.

Then, to make the prediction, the value of current saturation counter is evaluated. If it is $10_2$ or $11_2$ the branch is predicted to be taken; otherwise – not taken.

### 3.1.2 Local Predictor

Local predictor, instead of putting all the history in a single register, has a separate register for each specific branch, using instruction address (or program counter – PC) to distinguish them. This set of history registers is called local history table.

Local history table has only 1024 entries, so it is impossible to store the history for all the memory addresses possible. Instead, it uses 10 least significant bits of PC. Thus, if two branches in the same program are separated by a multiple of 1024 bits, they will point to the same entry in the local history table. However, when the execution is local enough, different branches correspond to different entries in the local history table.

Local history table stores 10-bit histories, that are then used to address the local prediction table, that works similarly to the global prediction table. Unlike global history table though, the saturation counters are 3-bit wide.

To make a prediction, the local predictor fetches history from the entry in local history table corresponding to the current value of the PC, and then fetches the corresponding saturation counter value from the local prediction table. If its value is $100_2$ or more, the branch is predicted to be taken. The general principle is that when the counter is half or more its maximum possible value, the branch is predicted to be taken.

#### 3.1.3 Chooser

Chooser stores the global history of whether the local or global predictor was performing better last time. To do it, it uses an array of 4096 2-bit saturation counters that are decremented when the local predictor gave the correct prediction and incremented if the global predictor was right. Thus, when they disagree on a particular prediction, these counters are used to decide what prediction to use.

#### 3.1.4 Structures

Since minimal addressable unit in the RAM of a modern computer is 1 byte, it is impossible to directly implement any register whose width is not a multiple of 8. As a workaround, these variables were encapsulated in structures as bit Fields, thus making it possible to work with them as if they had any specified width in bits.

#### 3.1.5 Performance

The overall performance over all the set of traces supplied, this design shows the misprediction rate of %5.4777. Several techniques were employed to lower this number, not all of whose turned out to be useful, but which in general allowed us to make it as low as 4.1662%.

## 3.2 Predictions for unconditional branches

Since unconditional jumps, calls and returns are always taken, there is no point to employ any algorithm to predict their outcomes. Ignoring the state of prediction tables and always predicting "taken" for them decreases the misprediction rate to 4.87%.

## 3.3 4-bit counters in local prediction table

Within the memory limit of 33792 bits, it is possible to use as much as 6 bits for the each entry in the local prediction table. Using 4 bits decreases the misprediction rate to 4.833%. 5 bits, however, give only a marginal improvement, and 6 bits even increase the misprediction rate. Thus, we've chosen 4 as the optimal value for the counter bit width.

## 3.4 Discarding unconditional branches

Even if unconditional branches are always predicted to be taken, the information about them still takes place in prediction tables. Skipping all the steps of

Table 1: Memory usage of the final design

| Name | Number of items | Size of an item (bits) | Total space used (bits) |
|---|---|---|---|
| Local history table | 1024 | 10 | 10240 |
| Local prediction table | 1024 | 4 | 4096 |
| Global history register | 1 | 12 | 12 |
| Global prediction table | 4096 | 2 | 8192 |
| Chooser prediction table | 4096 | 2 | 8192 |
| Total | | | 30732 |

predictor teaching for those branches allowed us to decrease the misprediction rate to 4.1879%.

## 3.5 G-Share

A good practice in branch predicting is to hash global history register before using it to address the global prediction table. Specifically, 12 most significant bits of the PC are XORed with the value of global history register. This value is then used to address the global prediction table and the chooser prediction table. This method lowered the misprediction rate to 4.1662%.

## 3.6 Memory usage of the final design

Our final design uses the total of 30732 bits of storage space. The exact breakdown is shown in Table 1.

## 3.7 Unhelpful techniques

Several other approaches were employed, but turned out to be unhelpful.

### 3.7.1 Saving states between traces

For each new trace the predictor is executed separately, and essentially initialized by garbage values from the memory. Alternative possible approach is to save its state from the previous trace, and use it to initialize the new predictor. However, not only it did not decrease the misprediction rate, but even slightly increase it up to 4.8348%.

### 3.7.2 Discarding least significant bits of the PC

In the baseline design, the 10 least significant bits of the PC are used to address the local history table. However, we hypothesized that it is unlikely for two conditional branches to be adjacent. In addition one Alpha's opcode is allegedly 4-bytes wide. Thus, there should be no possible way for two branches to differ only in two least significant bits. With this in mind, we attempted to right shift the PC before taking 10 least significant bits to address the local history table. Supposedly, that should have allowed us to store the information about the larger number of different branches without increasing the memory size. However, it turned out that right shifting the PC only increases the misprediction rate, and the more it is shifted the higher is the misprediction rate.

#### 3.7.3 Fully associative local history table

The assumption here is that the local history table is used inefficiently. It is not guaranteed that for all possible 10 bits there is a branch so that its address has these bits as its least significant bits. In other words, some entries in the local history table may be never accessed, while others are used for several different branches. Thus, we decided to exceed the memory limit and implement a fully associative local history table, just to see if it is possible to achieve better results this way. It was rather challenging to test the predictor, since fully associative array is much slower that that with direct mapping. We tested it with only one trace that initially had the misprediction rate of 9%. For this trace it increased up to 14%.

## 4 Questions

1. State three reasons why your trace-driven simulator would not perfectly reflect how your predictor would perform if implemented in a real processor pipeline.

    - Local history table is essentially a directly mapped cache without any check for the consistency of data. That can lead to false hits where one value of the PC corresponds to the entry created for another instruction whose address has exactly the same 10 least significant bits. While it is tolerable with trace-driven simulation, where all the instructions are branches, in the actual execution most of the instructions are ALU or memory instructions. Thus, where would be many false hits with the instructions that are not branches at all. Without specifically addressing this problem, it can cause significant decrease of the overall performance.
    - Traces that are generated for one set of machine characteristics are used to simulate a machine with a different set of characteristics. However, the execution path of a multiprocessor workload may depend on the ordering of events on different processors, which in turn depends on machine characteristics such as memory system timings. Trace-driven simulations of multiprocessor workloads are inaccurate unless the timing-dependencies are eliminated from the traces.[3]
    - Most of the trace-driven simulators may not model the instruction and cycle count accurately.There could be additional cycle counts

if the kernel-mode instruction execution is not modelled appropriately. In multiprocessor systems, some workloads introduce timing dependencies in the traces due to asynchronous interaction between processes which include unsynchronized access to shared data. Some trace-driven simulators attempt to reduce both the time and space requirements of trace-driven performance modeling. These techniques attempt to capture the behavior of the original program trace in a smaller, shorter trace that takes less space and time to store and process which may lead to inaccuracies in comprehending the results[2].

2. What is aliasing? Describe cases where aliasing is desirable and undesirable.

   - Aliasing occurs when more than one branch uses a single entry in the history tables(the same saturating counters)
   - Aliasing could be beneficial when it corrects a prediction that could have been incorrect.This is called constructive aliasing. In constructive aliasing, the branches have similar behaviors and they re-inforce each others predictions. The counter stays saturated at one end or the other. Even when one branch is suffering more transient behavior, the other branch tends to steady the predictions by remaining consistent. In destructive aliasing, the branches have opposing behaviors and cause constant misprediction[1].

3. Does increasing the number of bits in a saturation counter always improve performance? Give an example to support your position.

   - As shown in subsection 3.3, from some point increasing the number of bits in a saturating counter not only fails to improve the performance, but it also makes it worse.
   - Aliasing increases due to increase in number of bits.

## 5 References

[1] http://people.cs.umass.edu/ weems/CmpSci535/Discussion18.html

[2] Stephen. R. Goldschmidt. John L. Hennessey *The accuracy of trace driven simulations of multiprocessors*

[3] Bryan Black, Andrew S. Huang, Mikko H. Lipasti and John Paul Shen *Can Trace-Driven Simulators Accurately Predict Superscalar Performance?*

[4] P. Shen, John; Lipasti, Mikko *Modern processor design: fundamentals of superscalar processors.* Boston: McGraw-Hill Higher Education. 2005