Perceptron based Branch Predictor

Strahinja Praska

June 2024

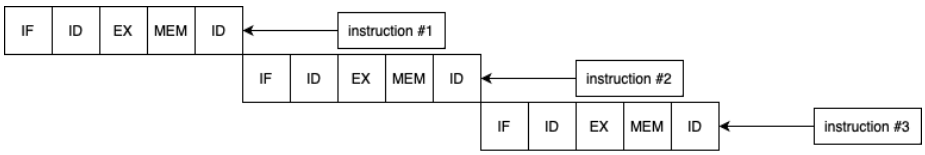
1 Instruction Pipeline

Instruction pipelining is a technique that enables us to process multiple instructions simultaneously by dividing instruction processing task into several stages.

There are 5 stages in pipelined RISC processor:

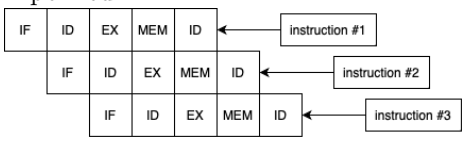
1. Instruction Fetch(IF)
2. Instruction Decode(ID)
3. Execute(EX)
4. Memory Access(MEM)
5. Writeback(WB)

In non-pipelined version we would have this situation:



It would take $5n$ cycles to run n instructions because each instruction would wait for previous to complete. The hardware that corresponds to different stage would have to wait for the remaining cycle to end.

Pipelined:



Throughput is 1 instruction per cycle now, we still deal with one instruction in 5 cycles but throughput is increased, we can process more instructions.

We could potentially extend the pipeline, and get even more benefits, but mispredictions would cost us more.

Each of stages in pipeline has it's corresponding register that passes data from one stage to another.

2 Branch prediction

Branch predictor is digital circuit that tries to guess which way will branch go before it's known definitely, it's purpose is to improve flow of instruction pipeline.

If we didn't have branch prediction in our pipeline, the processor would have to wait until the branch instruction has passed the execute stage before the next instruction can be fetched in the pipeline.

Branch predictor attempts to avoid this by trying to guess if branch will be taken or not, the branch that is guessed to be the most likely to be taken 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 flushed and the pipeline starts over with the correct branch incurring a delay.

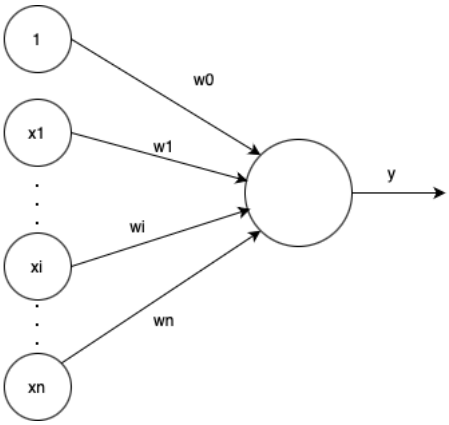
3 Perceptron based branch predictor

3.1 Motivation

We use perceptrons because of ease of implementation in hardware and easy to understand decisions that perceptrons make.

We use a single-layer perceptron consisting of one artificial neuron connecting several input units by weighted edges to one output unit.

A perceptron learns a target boolean function $t = (x_1, x_2, ..., x_n)$ of n inputs. In our case x_i will be bits of global branch history register where a perceptron keeps track of positive and negative correlations between branch outcomes and target boolean function t will be whether branch is taken or not.



3.2 Training

We represent perceptron by a vector of weights(signed integers).
The output is the dot product of weight and input vector, $x_0 = 1$ because of bias weight w_0, instead of learning a correlation with a previous branch outcome, the bias weight, learns the bias of the branch, independent of the history, each x_i is either -1 interpreted as not taken and 1 as taken.

$y = w_0 + \sum x_i w_i$
$x_i \in \{-1, 1\}, x_0 = 1, w_i \in Z$

We retrain perceptron after each known outcome, $t = -1$ not taken, $t = 1$ we would know this in execute stage where retraining would happen. We also use θ, threshold parameter to say when to stop training.

Algorithm 1 Perceptron training

```
if sign(y) ≠ t or |y| ≤ θ then
  for i:=0 to n do
    w_i := w_i + t x_i
  end for
end if
```

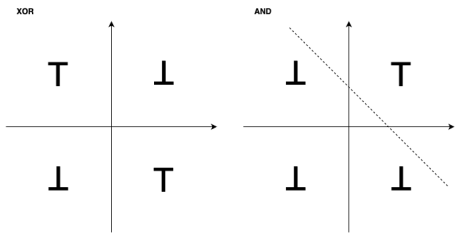
Since t and x_i are always -1 or 1, the algorithm increments the i^{th} weight when the branch outcome agrees with x_i and decrements the weight when it doesn't. When there is mostly agreement weight becomes positively large, when there is mostly disagreement, the weight becomes negatively large. When there is weak correlation, the weight is close to 0 and doesn't impact the output of perceptron.

3.3 Linear separability

A drawback of perceptron is that it can only learn linearly separable functions, but this is fine for us.

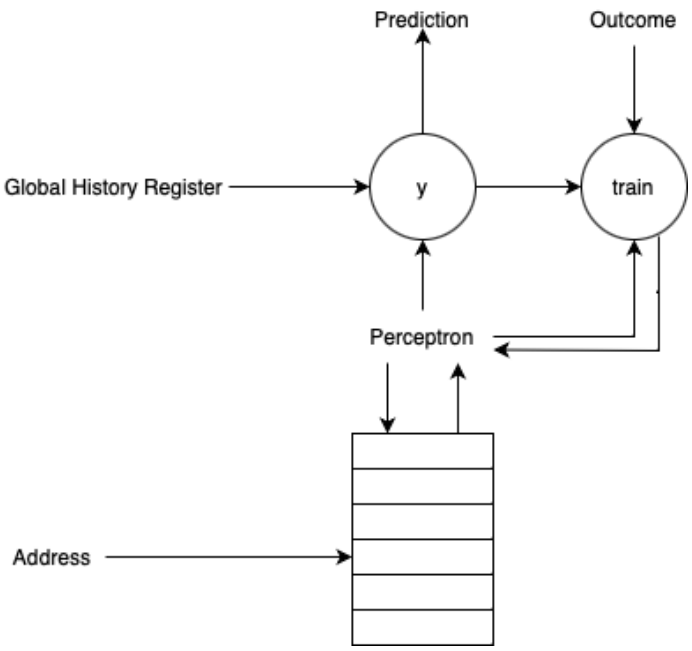
Mathematically, imagine the set of all possible inputs to a perceptron as an n-dimensional space.
The solution to the equation:
$w_0 + \sum x_i w_i = 0$
is a hyperplane(in 2-dimensional space, a line) diving the space into the set of inputs for which the perceptron will predict false and the set for which the perceptron will predict true.

Boolean function $t = (x_1, ..., x_n)$ is linearly separable iff there exist values for $w_0, ..., w_n$ such that all true instances can be separated from all the false instance by hyperplane.
Only linearly separable functions can be quintessentially, XOR cannot be separated, while AND can.



3.4 Constructing branch predictor

Construction below is branch predictor.



The CPU keeps a table of N perceptrons in fast SRAM, N is determined by number of weights that is determined by length of history.

The high level description of how predictor works:

1. We get by address corresponding perceptron P(vector of weights), from hash table.
2. The value of y is computed as the dot product of P and the global history register
3. The branch is predicted taken if $y \geq 0$, otherwise not taken
4. Once the actual outcome of the branch is known, the training algorithm uses this outcome and the value of y to update weights in vector P
5. P is written back to the hash table

4 Implementation and results

4.1 Implementation

Project is realised as simulation of instruction pipeline in Rust, that is instated with custom predictor that implements BranchPredictor trait(predict and update methods).

Something to be built upon is dealing with data hazards, but that is out of scope for this project.

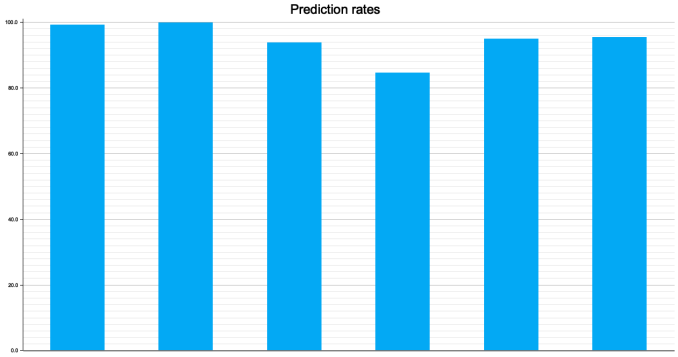
4.2 Results

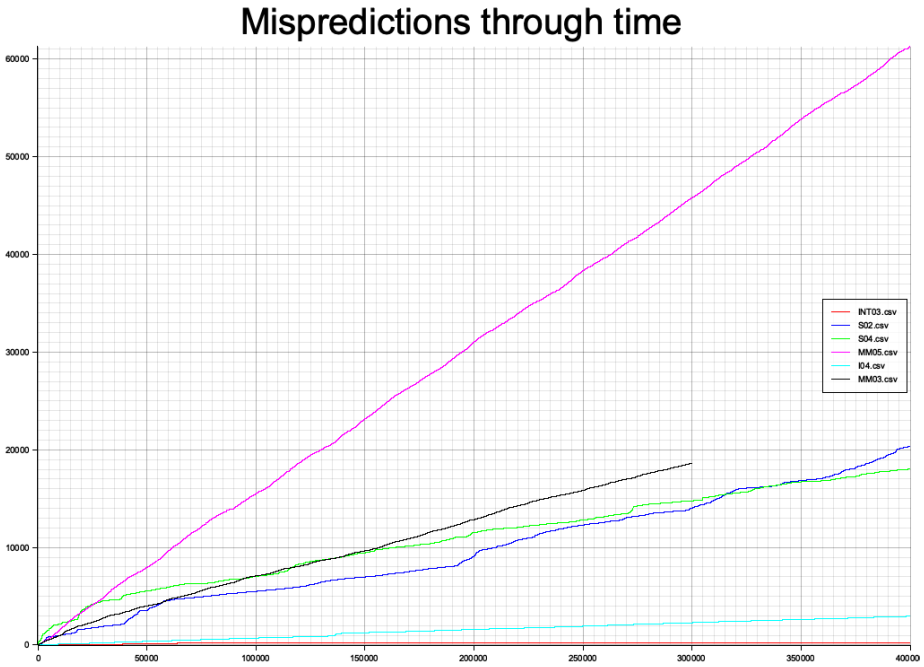
As for results, the best range for history length is proven to be 12-62 as in paper.

Prediction rates for each of datasets:

File	Prediction Rate (%)	Mispredictions
I04.csv	99.26	2951
INT03.csv	99.95	201
MM03.csv	93.79	18643
MM05.csv	84.86	61264
S02.csv	94.92	20331
S04.csv	95.49	18030

Table 1: Prediction Rates and Mispredictions for Different Files





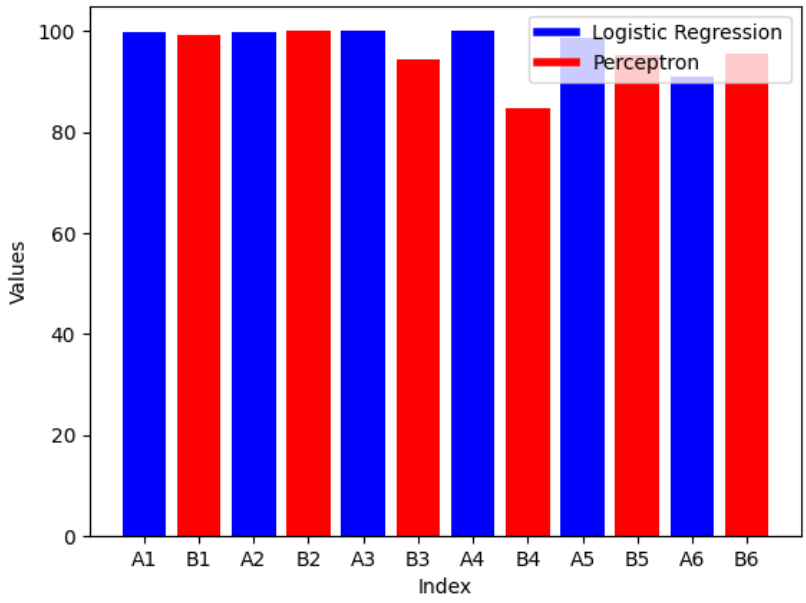
4.3 Logistic Regression

Something I've tried implementing is a logistic regression(modified for online learning) to compare the results to perceptron, designed in python.

It does outperform perceptron in terms of accuracy, but time is much longer. Although comparing rust to python is not fair, perceptron took 10 seconds to go through all datasets, while python script took 255 seconds.

File	**Prediction Rate (%)**	**Mispredictions**
I04.csv	99.88	479
INT03.csv	99.78	864
MM03.csv	99.97	91
MM05.csv	100.00	0
S02.csv	98.65	5413
S04.csv	91.00	35999

Table 2: Prediction Rates and Mispredictions for Different Files



5 Resources

Dynamic Branch Prediction with Perceptrons, Daniel A. Jimenez, Calvin Lin

Kaggle dataset

Branch predictor Wikipedia article

HIGH PERFORMANCE COMPUTING, L10, Lecture notes