# A Correct rate-based Swithing multi-thread Branch Predictor


## Name:MINGZHE ZHANG NUID:001055828


## Abstract


Studies about how to mix several branch predictors together increase nowadays. One of them is about voting by every predictor. Another is about neural branch prediction, which gives every predictor with different weight and then do a prediction. The first one may be inaccurate because it gives too much weight to predictor with low correct rate. The second one may be complex because machine learning libraries are applied most in Python libraries while c/c++ libraries are applied for basic layer more frequently. What's more, this kind of predictor needs a huge amount of data for its "learning" phrase and supposes different program's "taken" and "not taken" situations have a relationship someway. This paper designed a kind of correct rate-based swithing branch predictor and made a analysis of its prediction compared to different single algorithm predictor. It switch to different algorithm for predictor when this kind of algorithm has the max correct rate. It only use the algorithm with the highest correct rate. Also, it is based on c/c++. Moreover, it does not depend on other programs.


## introduction

This paper consists of serveral algorithms for predictor. However, the calculating one by one will cost a lot of time. So author utilize std::thread library to solve this calculation parallelly. Another problem is about data sharing. Address list , as a vital part, is only read by every algorithm/thread. So set it as a global variable is enough. Correct rate and predicts of different algorithm belong to algorithm itself, so it is not a problem either. What matters most is the final prediction. To solve this, author set a counter which is set to 0 default, and will be plus one if one algorithm is finished. This variable is set with a lock. Then the reduction thread will read if for times before it counter is full. Then the thread know it can read the result of all the threads, and then switch to the algorithm with highest correct rate.

## Background on tools and algorithms

### Taken predictor

Predict all addresses are taken.

## Not Taken predictor

Predict all addresses are not taken.

## One-bit predictor

It only record the last bit of result and do the same prediction as last do.

## Backward taken forward taken

It do the taken if it was taken and do the not taken if it was not taken.

## Bimodal predictor[1]

A bimodal predictor is a state machine with four states: Strongly not taken, Weakly not taken, Weakly taken, Strongly taken.

When a branch command is evaluated, the corresponding state machine is modified. If the branch is not adopted, the state value is decreased in the direction of "strong no selection"; if the branch is adopted, the state value is increased in the direction of "strong selection". The advantage of this method is that the conditional branch instruction must select a certain branch twice in succession to flip from the strong state, thereby changing the predicted branch.

## Delayed predictor

Similar to taken/not taken predictor, but prediction is determined by result several bit before.

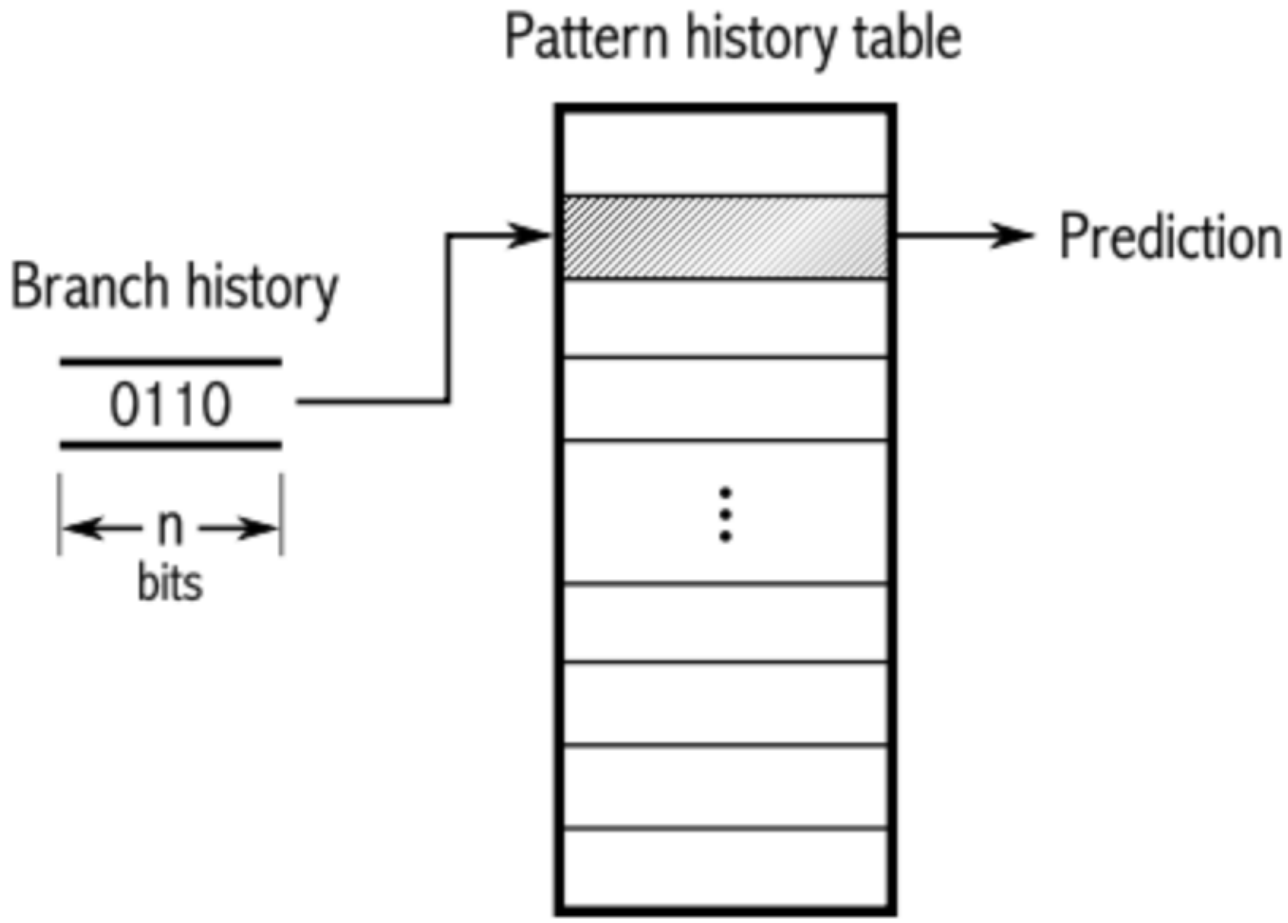
## History-based predictor

If this address has been done with taken, predict it taken again or predict it not taken.

## Two level adaptive predictor[2]

During the 1990s Two-level Adaptive Branch Predictors were developed to meet the requirement for accurate branch prediction in high-performance superscalar processors.

If an if statement is executed three times, the decision made on the third execution might depend upon whether the previous two were taken or not. In such scenarios, a two-level adaptive predictor works more efficiently than a saturation counter. Conditional jumps that are taken every second time or have some other regularly recurring pattern are not predicted well by the saturating counter. A two-level adaptive predictor remembers the history of the last n occurrences of the branch and uses one saturating counter for each of the possible 2^n history patterns.




[3]

## Pin

Pin is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools. Some tools built with Pin are Intel® VTune™ Amplifier, Intel® Inspector, Intel® Advisor and Intel® Software Development Emulator (Intel® SDE). The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications on Linux*, Windows* and macOS*. As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code[4]

## Linpack Benchmark

LINPACK is a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems. The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square. In addition, the package computes the QR and singular value decompositions of rectangular matrices and applies them to least-squares problems. LINPACK uses column-oriented algorithms to increase efficiency by preserving locality of reference.

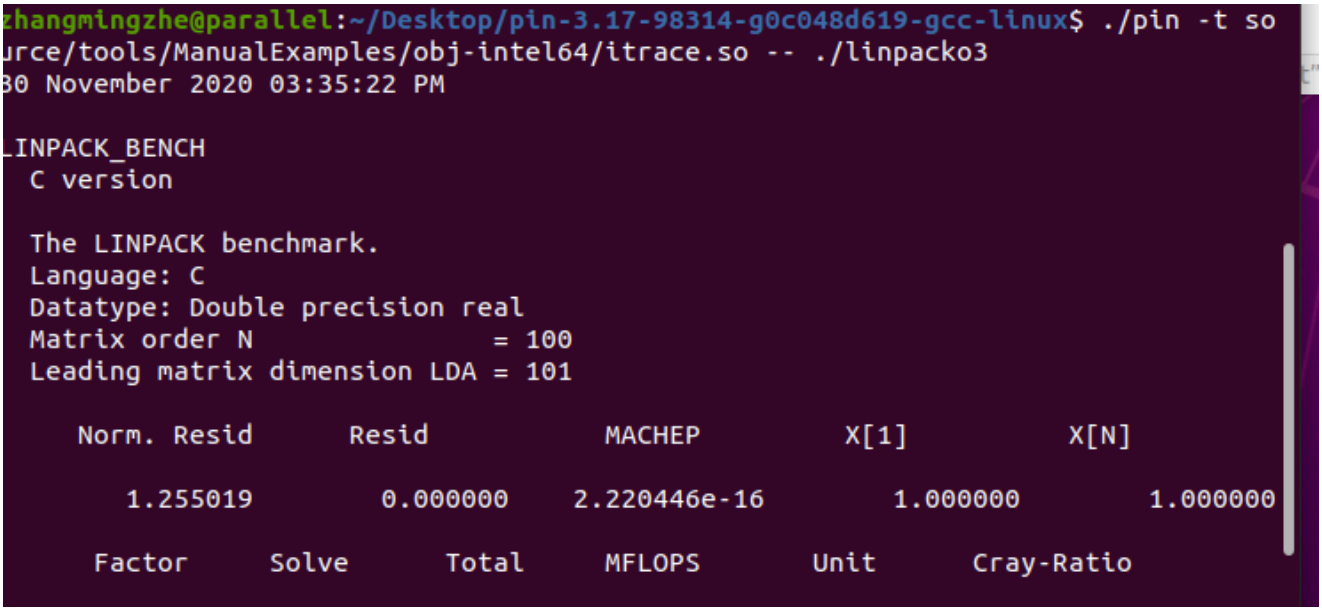
LINPACK was designed for supercomputers in use in the 1970s and early 1980s. LINPACK has been largely superceded by LAPACK, which has been designed to run efficiently on shared-memory, vector supercomputers.

Author only used pin tool to get the address trace of the program "linpack". The program is the sample program provided by source/tools/SimpleExamples/trace.cpp . Then program will calculate the differ of two address to determine it jumps or not.

Serveral algorithms were included in this predictor.

# Implementation Details

## Pin the trace of linpack address with differen N in linpack




## Read addresses into program

```
vector<long long> address(0);
ifstream fin("itrace.out");
    string str="";
    string::size_type sz;
    sz = 0;
    long long tmp;
    while(getline(fin,str)){
        if(str!="#eof"){
            tmp=stoll(str,&sz,0);
            address.push_back(tmp);
        }
    }
```

## Transfer the address to the taken/not taken record

```
    tmp=address[0];
    long long diff;
    for(int i=1;i<address.size();i++){
        diff=address[i]-tmp;
        if(diff<0||diff>15){
            jump.push_back(true);
        }
        else{
            jump.push_back(false);
        }
        tmp=address[i];
    }
```

## Start the predictor and do the reduction

```
std::thread threads[9];
    threads[0]=thread(reduction);
    threads[1]=thread(taken_predictor);
    threads[2]=thread(not_taken_predictor);
    threads[3]=thread(backward_taken_forward_not_taken_predictor);
    threads[4]=thread(one_bit_predictor);
    threads[5]=thread(bimodal_predictor);
    threads[6]=thread(two_level_adaptive_predictor);
    threads[7]=thread(delayed_predictor);
    threads[8]=thread(history_based_predictor);
    for (auto& th : threads) th.join();
```

# Result

ltrace.out is from

| file name\predictor name | Switch | Taken | Not taken | One bit | BWFN | Bimodal |
|---|---|---|---|---|---|---|
| itrace.out | 89.8603% | 10.1318% | 89.8682% | 80.5419% | 19.4581% | 89.4567% |
| itrace2.out | 97.7697% | 2.22992% | 97.7701% | 95.542% | 4.45801% | 97.7691% |
| itrace3.out | 97.5434 | 2.4559% | 97.5441% | 95.0914% | 4.90857% | 97.5424% |

| file name\predictor name | History-based | Two level adaptive | Delayed |
|---|---|---|---|
| itrace.out | 89.455% | 89.3296% | 80.9451% |
| itrace2.out | 99.8127% | 97.4797% | 96.0828% |
| itrace3.out | 99.7246% | 97.2069% | 95.7209% |

# Conclusion

The swithing predictor does not have the toppest correct rate among these predictors, but one of the toppest correct rates. The not taken predictor got very high correct rate because most of lines do not consist taken. But you can not use not-taken for the all code. Because this means no pretaking for the code which means this predictor should be out of consideration. So, for "small program" (with few repeating codes), switch is a proper predictor as well as bimodal and two level adaptive predictor. For "big program" (with repeating codes), histroy-based predictor has the toppest rate because of repeating. However, swithing predictor still has high rate of correct.

In a word, swithing predictor is very stable in different scenario with very high rate of corret.

Also, the speed is as fast as other predictor when running because of paralleling. This paper does not show the exact number of time because the different of time is under 1 second which is meaningless to compare.

# Improvement

## Swith frequency

Correct rate changes very subtly, it is useless and cost calculations to do reduction after every prediction. So the comparion can be improved with switching predicting policy every 100 addresses.

## Abandon not taken and taken policy

Both taken and not taken policy is useless, they should be abandoned.

## Default policy

Since bimodal predictor has the toppest correct rate for "small program", I will default it as a first-chosen policy.

## Add more predictor

History-based predictor with offset predictor, profile-based predictor and Function-returns-based predictor can be added into the polices of switching predictor.

## More advanced libraries

Openmp can be used for parallel solvements. A class that enumerate all the policies and use for loop with openmp comment to predict parallely.

## Utilize message queue instead of lock for sharing variable

Reduction will start when all predictions finished. Author used a counter for every thread, and reduction will start when the counter achieve the amount of predictions. However, competition may be very fierce when too many prediction and reduction threads compete together. All the other threads need to do is to let reduction threads know they are done. So choosing message queue to make model of this problem is a better choice.

# Reference


[1]"Dynamic Branch Prediction",(http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/). *web.engr.oregonstate.edu*. Retrieved 2017-11-01

[2]C. Egan, G. B. Steven, Won Shim and L. Vintan, "Applying caching to two-level adaptive branch prediction," Proceedings Euromicro Symposium on Digital Systems Design, Warsaw, 2001, pp. 186-193, doi: 10.1109/DSD.2001.952280.

[3]New Algorithm Improves Branch Prediction Better Accuracy Required for Highly Superscalar Designs, Linley Gwennap, MICROPROCESSOR REPORT

[4]Pin 3.17 User Guide,https://software.intel.com/sites/landingpage/pintool/docs/98314/Pin/html/

[5]Matlis, Jan. Sidebar: The Linpack Benchmark. ComputerWorld. 2005-05-30