# EVOLUTIONARY DESIGNED BRANCH PREDICTORS

Karel Slaný and Václav Dvořák

Faculty of Information Technology, Brno University of Technology
Božetˇchova 2, 612 66 Brno, Czech Republic
Phone: +420 54114-1176 Fax: +420 54114-1270
slany@fit.vutbr.cz, dvorak@fit.vutbr.cz

*Branch prediction techniques are commonly used for speeding-up code execution. Modern microprocessors use predictors based on a set of finite automata predictors. This paper shows that finite automata branch predictors can be created by using evolutionary algorithms. These evolved predictors have better performance in predicting the code execution, which they have been trained for, than a standard 2-bit counter predictor.*

## 1 INTRODUCTION

Branch prediction is a technique for speeding-up code execution on modern microprocessors. When a branch instruction enters the execution pipeline of the processor all instruction which are following this conditional jump instruction have to wait until this condition is evaluated and the correct branch of the executed code is chosen. This causes a problem on modern processor designs with long pipelines because a relatively long time is needed for emptying and reloading the pipeline.

Several techniques have been invented which can minimize the waiting in the pipeline. One of them is speculative code execution. Branch directions are statically or dynamically predicted in order to maintain code execution and keep the pipeline filled. Speculative execution is used to reduce the time when the pipeline is waiting.

Imagine that the processor is executing code when a vast majority of conditional jump instructions is taken, respectively not taken. A static prediction which predicts jumps, respectively no jumps, is of great advantage. In more general cases a dynamic prediction can do a better job. It uses a more complicated type of prediction where the predictor can adapt itself to the behaviour of the executed code in order to achieve better prediction accuracy.

This paper shows that evolutionary techniques can be used for predictor training thus adapting it to the executed code on the fly, during code execution, in order to increase its performance.

## 2 BRANCH PREDICTION

Modern processors used in PC architecture computers do use branch prediction techniques, but detailed information is not offered by its manufacturers. However most of the used branch predictors use the schema on the picture (1).

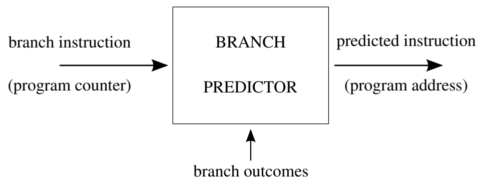

Figure 1: Simple scheme of a branch predictor.

Intels Pentium (P5) used a four-state saturated counter. Each state determined whether to take a conditional jump or not. The current state of the predictor changes in dependency whether in the executed code was a jump taken or not. Its structure is shown in the picture (2).

In further processor-designs of the Pentium family (Pentium MMX, Pentium Pro, Pentium II, . . . ) was this one-level prediction system improved by adding a four-bit shift register. This register is used as a simple jump history buffer which gives a history of 16 patterns, that are used for addressing into a bank of 2-bit sate counter, which are similar to te predictor on the picture (2). The advantage of this mechanism is that it can learn a repetitive patterns occurring in the executed code, therefore giving better performance than a simple one-level design.

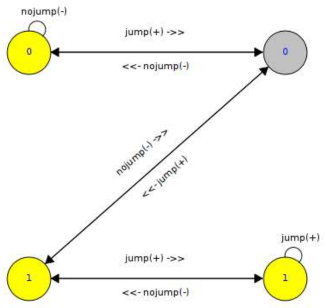

Figure 2: 2-bit counter used as a branch predictor. Zeros in the states represent predicted no jumps. Ones stand for predicted jumps.

## 3 EVOLUTION OF BRANCH PREDICTORS

The branch prediction system described in this paper uses evolutional algorithms for creating a one-level state predictor that can suit best for the currently executed code. That means that the predictors are adapted on the fly by a evolutionary core running simultaneously with the code execution. The whole predictor design was implemented in software and tested on different executed codes.

The design of the system consists of a programme execution unit an evolutionary core and a prediction unit. The design is described on the picture (3).

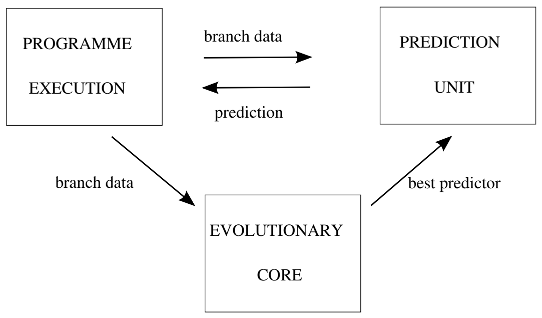

Figure 3: Structure of the evolutionary driven prediction system.

The programme execution unit sends information whether conditional jumps were taken, in that case it sends a $1$. If the branch instruction was not taken it sends a $0$. The core block runs the evolution of the predictors and passes the currently best design into the prediction unit. The prediction unit uses the last predictor it has become to predict branch instruction behavior.

### 3.1 CALCUALTING FITNESS FUNCTION

The predictor $P$ (1) is described by the finite set of it states $Q$, the finite input alphabet $\Sigma = \{0, 1\}$ which represents input and output alphabet, $0$ is no jump, $1$ represents jump. The set of edges $T$ define mapping $Q \times \Sigma \rightarrow Q$. There also exists a mapping $D : Q \rightarrow \Sigma$ which represents the meaning of the current state. An $q_i \in Q$ initial state can be also defined.

$$P = \{Q, \Sigma, T, D, q_i\} \tag{1}$$

Each predictor is represented by its chromososome $C$ (2) which is a string

$$\begin{gathered} C = (i_i)[d_0, j_{t_0}, n_{t_0}]_0[d_1, j_{t_1}, n_{t_1}]_1 \ldots [d_{n-1}, j_{t_{n-1}}, n_{t_{n-1}}]_{n-1} \\ 0 \leq i_i < n,\ n = |Q|,\ d_m \in \Sigma,\ j_{t_m} \in Q,\ n_{t_m} \in Q,\ 0 \leq m < n \end{gathered} \tag{2}$$

where as mentioned $i_i$ is the index of the initial state and the triplet $[d_m, j_{t_m}, n_{t_m}]_m$ describes the $m$-th state from the set $Q$. The $j_{t_m}$ is the target index of a state of the transition which is taken, when a jump in the code is made, and the predictor has to change its state. When there was no jump and a change of state has to be made, then the taget is described by index $n_{t_m}$. The description of the state is given by $d_m$.

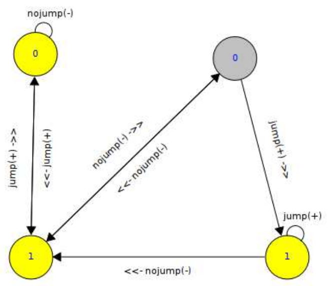

Figure 4: Structure of a predictor with the chromosome: $(0)[0,3,2][0,2,1][1,1,0][1,3,2]$

The history of branch data is stored in a buffer $H$ (3) which holds the last $k$ branch states,

$$\begin{gathered} H = h_0 h_1 h_2 \ldots h_{k-2} h_{k-1} \\ h_m \in \Sigma,\ 0 \leq m < k \end{gathered} \tag{3}$$

where again $0$ stands for a not taken jump and $1$ is a taken jump.

Let there be function $update(x)$ which gives $0$ when the prediction was false and $1$ when the prediction was true. This function holds an internal number state $q$ which is before the function is used for the first time initialized to $q_i$. Each time the function is called it compares the predicted with $x$, the current branch data, and updates its internal index. Then the fitness function $f$ (4) can be described as

$$f = \sum_{i=0}^{k-1} update(h_i) \tag{4}$$

where $0 \leq f \leq k$. Higher values of the fitness function stand for better predictor. If the valuse is equal to $k$ then the predictor can predict data in stored in $H$ with 100% accuracy.

## 3.2 EVOLUTION SETTINGS

The function of the system was simulated on branch data obtained by running warious programs. In order to keep the evolution as fast as possible and to reduce the computational load, the used branch data were sampled before the run of the predictor system. The speed in which the sampled data were issued into the system was $100$ samples per one generation cycle. The data were send in a loop. Each time the end of the sampled data was reached the data were send into the $H$ buffer from the beginning.

The algorithm can be described in pseudocode as:

```
do {
  issue sample burst;
  generate new population;
  evaluate new population;
  if (new_population_fittest_fitness > old_population_fittest_fitness)
    send new_population_fittest_chromosome to prediction unit;
} until (maximum number of generations is reached);
```

The `generate new population` command executes this steps. The fittest member of the current population is copied into new generation. Two chromosomes from the current population are selected an the rest of the entire new population is generated. Each time during the generation cycle a simple crossover of the parrental chromosomes is performed and mutation operator is applied on both offsprings. Then both offsprings are validated. This is done because of the mutation operator, which can produce invalid chromosome code. If an invalid code is found it is randomly modified in order to fix it. Both offspings are moved into new population. Mutation can preform changes in all places of the chromosome. The validation operation changes the values of $d_m$ state description parameters so that the ratio between $0$ and $1$ marked states is kept near $1$.

## 4 EXPERIMENTS

All experiments were run on sampled branch data data of the length about $1000000$ samples. The length of the buffer $H$ holding past branch data which were used for predictor training was set to $10000$ samples. Mutation probability was set to 3%. The population size varied. Larger population were used for predictors which had more states. As further mentioned elitism was used to keep the best evolved member in population. The experiments were run for $4000$ generations.

| **N**umber of precdictor states | **P**opulation size |
|---|---|
| $4$ | $6$ |
| $3$ | $6$ |
| $5$ | $10$ |

Table 1: The population sizes in dependency on the evolved predictor sizes.

The $5$-state predictors do not have as high good prediction rate as the $3$ and $4$ state predictors.

The predictor was simulated on branch data obratined from running these programmes: compilation with *g*cc, compression with *b*z2, compression with *g*zip and running *j*ava.

During the simulation of the system, predictor data send to the prediction unit were saved. Especially when training larger predictors a behavour was observed when no major improvement was made. But in those cases a set of two or three predictor designs altered themselves in the prediction unit and no other predictor was evolved.

The best evolved predictors were compared with a standard 2-bit counter scheme which is on the picture (2). The predictors were tested in predicting the bahaviou of the whole programme and their correct prediction counts were compared as a ratio. The results are in the table (2).

| **P**rogramme | **R**atio evolved predictor : 4-state predictor |
|---|---|
| *g*cc | $1$ |
| *b*z2 | $2.45$ |
| *g*zip | $2.58$ |
| *j*ava | $1$ |

Table 2: Comparing the best evolved predictor designs wit the 4-state prediesctor used in Pentium processors.

Some of the evolved 4-state predictor designs are shown in the picture (5). In many cases some states of the predictor are unaviable thus reducin the design to less states.

## 5 DISCUSSION

Predictors evolved during the code execution have been proven to have at least the same performance as a standard 4-state saturated counter predictor. In some cases the evolved predictors have significantly better performance in specific code execution. This can be a great advantage. However this system has a great disadvantage which lies in its complexity. This system is painfully slow in comparison with other branch prediction systems.

## 6 CONCLUSION

In this paper a system for creation evolutionary designed predictors was desribed. Evolutionary algorithm was used to design predictors and to adapt them to the changing behavior of the executed code. The use of evolution has proven functional. However the main disadvantage of thi system is its speed.

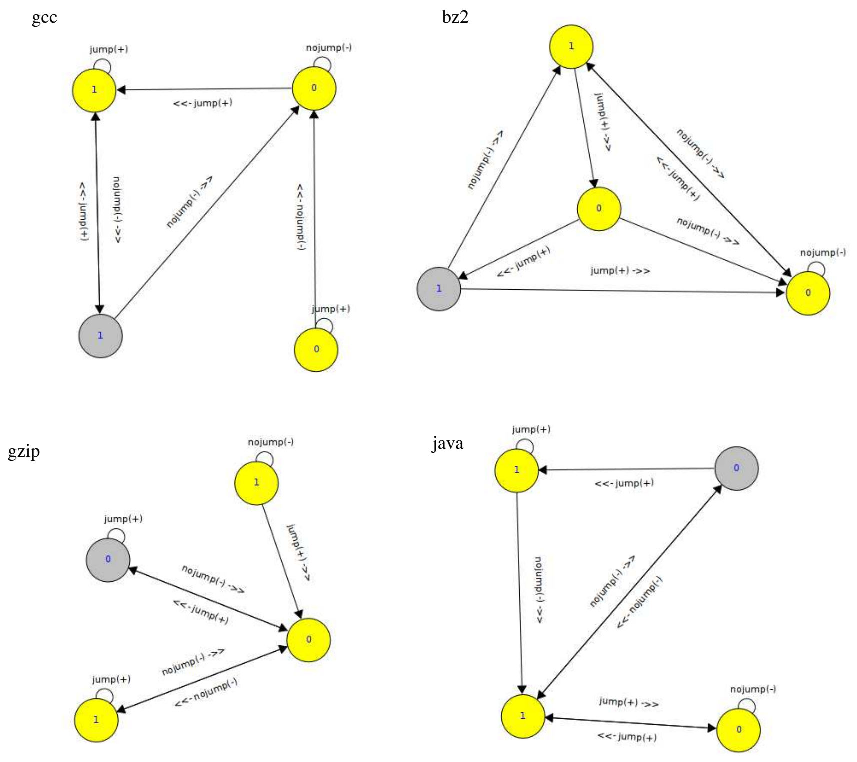

Figure 5: Structure of evolved 4-state predictors.

## ACKNOWLEDGEMENTS

This work was supported by the Grant Agency of the Czech Republic under No. 102/ 07/0850 *Design and hardware implementation of a patent-invention machine* a nd the Research intention No. MSM 0021630528 – Security-Oriented Research in In formation Technology.

## REFERENCES