Branch Predictor On-line Evolutionary System

Karel Slaný
Faculty of Information Technology, Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
slany@fit.vutbr.cz

ABSTRACT

In this work a branch prediction system which utilizes evolutionary techniques is introduced. It allows the predictor to adapt to the executed code and thus to improve its performance on the fly. Experiments with the predictor system were performed and the results display how various parameters can impact its performance on various executed code. It is evident that a one-level predictor can be evolved whose performance is better than comparable predictors of the same class. The dynamic prediction system predicts with a relative high accuracy and outperforms any static predictor of the same class.

Categories and Subject Descriptors

I.5 [**Pattern Recognition**]: Miscellaneous

General Terms

Experimentation

Keywords

branch prediction, finite automata predictors

1. INTRODUCTION

Conditional code branching is an important part of algorithms. Whenever a branch command has to be executed the instruction parameters must be evaluated. The branch outcome depends on the evaluation of the branch parameter.

Modern scalar and super-scalar processor architectures execute multiple instructions simultaneously so code branching is a problem because of its nature. These architectures use long pipelines [9] to speed up instruction execution. Each time a conditional jump instruction enters the pipeline, the pipeline has to wait until this conditional jump instruction is evaluated and the correct branch is known. Therefore, branch prediction techniques were developed in order to reduce this pipeline stalls. A branch direction is predicted in advance in order to keep the pipeline filled. The predictor has to have a high prediction accuracy because whenever there is an incorrect prediction the pipeline has to be flushed and re-filled again.

Earlier processor designs of the Intel's Pentium family used a predictor based on Moore's automata. It has the form of a 4-state saturated counter [8]. It is a simple one-level predictor design, which means that there is no additional mechanism used and the system predicts directly the branch direction. This plain predictor design allows a good prediction at simple loops. It predicts the action which has happened more often several times in the past.

Latter designs used a bank of automata [10]. They consists of a history buffer which holds the last 4 branch outcomes. This buffer is then used for addressing a bank of 2^4 4-state predictors. One can see a two-level predictor design where the history buffer represents a decision logic. This logic determines which of the state automata will be used for prediction.

In the past, on-line evolutionary systems were proposed for using in evolvable hardware (EHW) in various applications including hash functions [1] and data classification [3]. An on-line evolutionary system, which is solving the task of noise removal in corrupted images was proposed [6].

The aim of this paper is to demonstrate that an adaptive system based on state automata evolution can perform well in the task of branch prediction.

2. STATICALLY EVOLVED PREDICTORS

In [7] an idea of creating a state predictor that would perform better on a specific type of code which it has been trained for was presented. The simple 4-state counter predictor does not perform well on all executed code. The special predictors have been evolved on sampled branch data obtained by running various programmes. The sampled data were stored in a file containing only the traces of the conditional jumps executed in the programme. The file contains information whether the branch instruction was taken or not, i.e. whether the conditional jump was or was not made.

2.1 Predictor Description

The predictor is represented as a deterministic finite state Moore's machine. Its states hold the information whether the next conditional jump will be taken or not. The predictor automata M can be described by a 6-tuple

$$M = \{Q, q_i, \Sigma_i, \Sigma_o, T, G\} \quad (1)$$

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

where Q is a finite set of states, q_i is the initial state, $q_i \in Q$. Σ_i is the input alphabet and Σ_o stands for the output alphabet. T is the transition function $T : Q \times \Sigma_i \to Q$ and G is the output function $G : Q \to \Sigma_o$. The Σ_i and Σ_o consists only of two symbols $\Sigma_i = \{-, +\}$ and $\Sigma_o = \{0, 1\}$. The symbol '$-$' stands for a not-taken conditional jump the '$+$' symbol represents a taken conditional jump. In the output alphabet Σo the symbol '0' represents the prediction of a not taken jump and '1' again stands for the prediction of a taken jump.

The data which the predictor is trained on can be represented by a string $Si \in \Sigma_i$.

$$S_i = a_0 a_1 a_2 \dots a_{n-1}, \quad a_i \in \Sigma_i, \quad 0 \leq i < n \tag{2}$$

This string (2) represents a sequence of n sampled branch outcomes.

The prediction mechanism can be seen as a transduction from the input string S_i to an output string S_o.

$$S_o = b_0 b_1 b_2 \dots b_{n-1}, \quad b_i \in \Sigma_o, \quad 0 \leq i < n \tag{3}$$

Let's assume that the transitive closure $T^+ : Q \times S \to Q$ on the string S of the transition function T defined as

$$T^+(q_i, a_0 a_1 \dots a_{n-1}) = T(T(\dots T(T(q_i, a_0), a_1) \dots), a_{n-1}) \tag{4}$$

returns the state of the automata which is determined by accepting the string S from the defined state. Starting from the initial state q_i, the first symbol b_0 in the output string (3) can be determined by using the output function G on the initial state q_i, $b_0 = G(q_i)$. Then, for all symbols in the input string a_j the symbol b_{j+1} can be determined as

$$b_{j+1} = G(T^+(q_i, a_0 a_1 \dots a_{j-1} a_j)). \tag{5}$$

The output string constructed from the input string by using (5) holds the predicted data.

The relationship between these two strings becomes visible. The symbols '$-$', '0' and again '$+$', '1' stand for the same situation in the code execution. The only difference is in the time these strings represent. The first stands for past data and the second represents the future data.

The quality of a predictor can be measured by comparing these two strings. Whenever a '$-$' occurs in the j-th place in S_i, the symbol '0' has to occur in the same j-th place in S_o in order to be a correct prediction. This goes vice-versa for symbols '$+$' and '1'. The relation can be described by a function

$$f_c(a, b) = \begin{cases} 0, & a = -, b = 1 \\ 0, & a = +, b = 0 \\ 1, & a = -, b = 0 \\ 1, & a = +, b = 1 \end{cases} \tag{6}$$

which returns the number 1 when the symbols a and b correspond. The number of correct predictions for an input string of the length n can be computed by generating the output string and computing the sum

$$C = \sum_{j=0}^{n-1} f_c(a_j, b_j) \tag{7}$$

of corresponding letters in the strings. The ratio

$$H = \frac{C}{n} \tag{8}$$

represents the success rate of the predictor in predicting future branch outcomes.

2.2 The Representation of the Predictor

The genome representation of the predictor can be much simpler than (1). The whole predictor can be described in terms of the transition function and the output function. As a matter of fact, these two functions can be described by only one table. And this table can be transformed into a single string of integers.

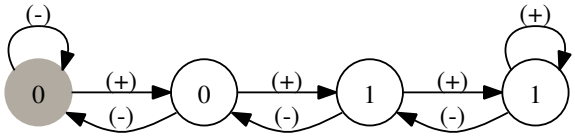

Figure 1: The graphical representation of a finite state predictor.

Table 1: The tabular representation of a state machine in the fig. (1).

State	G	$T(+)$	$T(-)$
0	0	1	0
1	0	2	0
2	1	3	1
3	1	3	2

Let's have a simple 4-state predictor shown in fig. (1) which can be described by a table shown in tab. (1). This tabular representation can be transformed into a string of integers which represents the predictor genome. The structure of the genome of a m-state predictor is described by the sequence

$$i, g_0, t{+}_0, t{-}_0, g_1, t{+}_1, t{-}_1, \dots, g_{m-1}, t{+}_{m-1}, t{-}_{m-1} \tag{9}$$

where i, $0 \leq i < m$, is the index of the initial state. The triplet $g_x, t{+}_x, t{-}_x$ describes the x-th state $g_x = G(x)$, $t{-}_x = T(x, -)$ and $t{+}_x = T(x, +)$.

An evolutionary algorithm was used in order to find a suitable predictor, using training code. Experiments with the number of states were performed. These evolved predictors showed higher or at least equal performance than a standard one-level 4-state predictor. In particular cases the experiments proved that although the total number of states a single predictor contains could not vary during the evolutionary process, the evolved predictors were able to reduce the effective number of states. This came out by making some states inaccessible from other states which were normally used.

The predictors were trained on data obtained from running a *gcc* C code compilation, data compression with *bzip2*, *gzip* and *java* virtual machine execution. The success rate results are compared with a 4-state saturated counter predictor in table (2).

3. DYNAMIC BRANCH PREDICTION SYSTEM

This section describes a prediction system that, by using evolutionary algorithm, can dynamically adapt to the currently executed code to achieve a better prediction even in the case of switching the executed code.

Table 2: Success rates comparison of a standard 4-state counter predictor with the performance of 4-state predictors specially trained for predicting branches in special programmes.

Programme	evolved predictor	4-state counter
gcc	0.55	0.54
bzip2	0.71	0.62
gzip	0.70	0.59
java	0.58	0.58

3.1 System Description

The prediction system was similarly designed as a two-level prediction system. An evolutionary core is implemented in order to change the behavior of the predictor. The best predictor is used in a separate unit. It can be replaced by a new one whenever the evolutionary unit finds a better solution.

The history of branch outcomes has to be recorded. This data are stored in a history buffer. This buffer represents a training environment for the predictor population. The buffer can be used for both the input, as well as the output string. Using the data stored in the buffer, the fitness value f_v of the evolved predictors is determined. Assume that the length of the history buffer is set to m branch outcomes. Then the fitness value $f_v = m$ stands for a 100% predictor success rate. On the other hand $f_v = 0$ means that the predictor is useless without a single success in predicting data stored in the buffer.

In order to keep the predictor system as fast as possible the evolutionary algorithm is relatively simple. The evolution runs in an infinite loop and can be described by using the following pseudo-code.

```
while (1) {
  copy_elite_to_new_population;
  while ( new_population_not_full ) {
    p1, p2 = select_two_parents;
    o1, o2 = cross_over( p1, p2 );
    mutate( o1, o2 );
    put_into_new_population( o1, o2 );
  }
  move_new_population_to_old;
  evaluate_old_population;
  send_best_to_prediction_unit;
}
```

Each time the prediction unit receives a new predictor, the previous one is replaced. The prediction unit stores the state of the used predictor. When a new predictor is being installed into the prediction unit, the state counter of the predictor is also updated to the initial state of the newly sent predictor. This is because of the cases when the predictor contains unreachable states and the state counter points to such a state.

The genome (9) is represented by using a string of integers. The mutation operator can change all the information related to the genomes transition function as well as the initial state. The ratio between the number of states predicting '0' and '1' in a single predictor is kept as close as possible to 1. This is due to the fact that the branch outcomes in the executed code can be changed by altering the conditional jump instruction by using just the negation of this condition. The crossover operator performs a single-point crossover, which can be performed at any place in the genome.

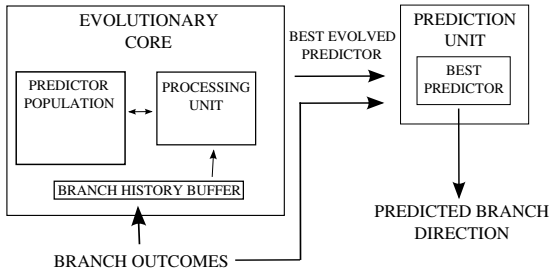

Figure 2: The structure of the prediction system. The branch outcomes are stored in the history buffer which is used for training predictors in the population. A processing unit executes the evolutionary algorithm. The best evolved predictor is sent to the prediction unit where it is used for predicting the branch outcomes.

3.2 Behaviour of the Dynamic System

The system executes the evolution in an infinite loop. In each generation cycle, the system has to adapt the individuals in the older generation to the modified environment in the branch history buffer. The progress of the best evolved fitness is shown in figure (3).

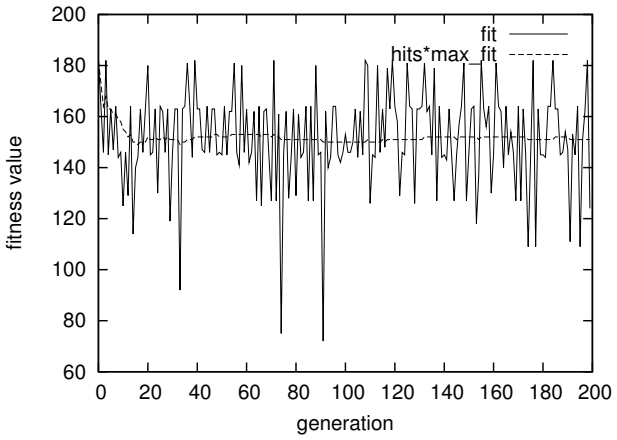

Figure 3: The progress of the best fitness value and the total predictor success rate during the first 200 generations after the system has been started. This data were recorded on the prediction of *bzip2* code. The experimental setting were: automata states 4, population size 6, history buffer size 200, burst length 200. *fit* is the best fitness value in current generation, *hits*max_fit* is the hit rate of the prediction system so far multiplied by the maximal possible fitness value, which is equal to the history buffer size. This multiplication was performed in order to achieve the correct ratio between the displayed values.

4. EXPERIMENTAL RESULTS

The experiments are performed by simulating the run of a programme by issuing the branch outcome data in regular bursts each generation cycle. For testing, three data-sets are utilized: C code compilation using *gcc*, data compression using *bzip2* and a *java* virtual machine execution. Each experiment is repeated 100-times and the experimental results are recorded. The mutation probability is set to a fixed value of 0.03. Elitism is used. One population member with the best fitness value is maintained in the population. Tournament selection is used for selecting parents. The tournament size is set to 3.

The experiments are performed with various parameter settings and its combination. The main parameters are:

population size - the number of predictors in the population

number of states - the number of states in the evolved predictors

history buffer length - number of past branch outcomes used for predictor training

branch burst length - this is the number of new branch outcomes added into the history buffer

In each generation cycle the new branch data are shifted into the history buffer and the same number of older branch outcome data are shifted out from the buffer. The number of newly added branch data is defined by the *branch burst length* parameter. This parameter defines the minimum number of branch outcomes to be predicted by a predictor in the prediction unit until it may be replaced by a newly evolved predictor.

4.1 Size of the Automata

The first set of experiments was dedicated to the determination of the optimal states number in a predictor in order to maximize the prediction success rate. Other parameters were fixed for all types of experiments. Experimental results are shown in the table (3).

Table 3: Predictor system success rate and its standard deviation in dependency on the automata size. Population size is set to 6 members. History buffer length is set to 200, while burst length is equal to 20.

program	number of states	success rate	std. dev.
gcc	4	0.57	0.00041
gcc	6	0.55	0.00027
gcc	8	0.53	0.00048
gcc	10	**0.62**	0.00038
bzip2	4	**0.72**	0.00023
bzip2	6	0.69	0.00047
bzip2	8	0.70	0.00038
bzip2	10	0.64	0.00041
java	4	**0.71**	0.00059
java	6	0.70	0.00043
java	8	0.69	0.00056
java	10	0.68	0.00031

The experimental results point out that less states display better performance in *bzip2* and *java* code prediction. However, the run of the *gcc* shows a more complex pattern where more states in the predictor are of advance.

Table 4: Predictor system success rate and its standard deviation in dependency on the population size. The number of states of evolved predictors is set to 4. History buffer length is set to 200, while burst length is equal to 20.

program	pop. size	success rate	std. dev.
gcc	4	0.55	0.00034
gcc	6	0.57	0.00055
gcc	8	0.58	0.00043
gcc	10	**0.58**	0.00029
bzip2	4	0.68	0.00020
bzip2	6	0.72	0.00051
bzip2	8	0.72	0.00025
bzip2	10	**0.74**	0.00047
java	4	0.68	0.00050
java	6	0.71	0.00043
java	8	0.73	0.00027
java	10	**0.73**	0.00013

4.2 Size of the Population

In this set of experiments the size of population was modified in order to determine how it influences the predictor performance. The results are summed up in the table (4).

This experiment set shows that a bigger population size is of advance in all measured cases.

4.3 History Buffer Size

This set of experiments was the most complex one. It was dedicated to determine the best combination of the history buffer size and the branch outcomes burst length. Results are shown in tables (5, 6, 7). The population size was set to 6, the number of states to 4.

In predicting the *gcc* and *java* code the experimental results show that the prediction system reaches the best performance when the evolution runs at a very high speed, i.e. when new predictors are issued at high speed and when the predictors are trained on few past branch outcomes. In case of predicting the *bzip2* code, the history buffer length does not have to be short. Perhaps it is because of the branch outcome patterns of the code.

5. DISCUSSION

The speed of the evolution is fundamental in cases, when the problem represented by the branch prediction changes dramatically. This can be illustrated by the C code compilation with *gcc*. The compiler has to behave according to the input file and while being dependent on the state of the compilation process, it has to change its behavior. The executed code shows different branch patterns while parsing the input file and different patterns while it is assembling the output object file. On the other hand the data compression using *bzip2* executes code with the same branch patterns which are relatively simple to learn and which are not changing very often. In such cases the speed of the evolution does not have to be very fast.

In this dynamic prediction architecture the finite state predictor in the prediction unit is replaced during every gen-

Table 5: Success rate experimental results for various combination of history buffer length and burst lengths. Experiments performed with prediction of *gcc* code.

buffer size	burst length												
	10	20	40	80	120	160	200	240	280	350	400	450	500
10	**0.70**												
20	0.59	0.66											
40	0.57	0.60	0.62										
80	0.56	0.57	0.59	0.61									
120	0.55	0.57	0.57	0.59	0.60								
160	0.55	0.57	0.58	0.59	0.58	0.59							
200	0.55	0.57	0.57	0.58	0.58	0.58	0.59						
240	0.55	0.56	0.57	0.58	0.58	0.58	0.58	0.59					
280	0.55	0.57	0.57	0.58	0.58	0.58	0.58	0.58	0.59				
350	0.55	0.56	0.57	0.58	0.57	0.58	0.58	0.58	0.58	0.58			
400	0.55	0.56	0.57	0.57	0.57	0.57	0.58	0.58	0.58	0.57	0.58		
450	0.55	0.56	0.57	0.57	0.57	0.57	0.58	0.57	0.58	0.57	0.58	0.58	
500	0.55	0.56	0.57	0.57	0.57	0.57	0.58	0.57	0.58	0.57	0.58	0.58	0.58

Table 6: Success rate experimental results for various combination of history buffer length and burst lengths. Experiments performed with prediction of *bzip2* code.

buffer size	burst length												
	10	20	40	80	120	160	200	240	280	350	400	450	500
10	0.78												
20	0.69	0.75											
40	0.69	0.72	0.75										
80	0.68	0.71	0.73	0.75									
120	0.69	0.71	0.72	0.73	**0.80**								
160	0.68	0.71	0.72	0.73	0.74	0.75							
200	0.68	0.72	0.72	0.73	0.73	0.74	0.76						
240	0.69	0.74	0.73	0.73	0.73	0.73	0.74	0.76					
280	0.68	0.71	0.73	0.73	0.73	0.74	0.73	0.74	0.75				
350	0.70	0.74	0.73	0.73	0.73	0.74	0.74	0.76	0.74	0.79			
400	0.68	0.71	0.72	0.73	0.73	0.74	0.74	0.74	0.74	0.74	0.76		
450	0.71	0.71	0.72	0.73	0.75	0.74	0.73	0.74	0.74	0.74	0.74	0.76	
500	0.68	0.71	0.73	0.74	0.73	0.74	0.73	0.74	0.75	0.74	0.74	0.74	0.76

eration cycle. Another approach to this problem can be via creating a more complex prediction unit. This unit can contain a bank of different state predictors. The prediction unit can choose the predictor with the best success rate in the past to be the predictor whose prediction might be used. Other possibility is to implement a majority function. The output from the prediction unit is the value which has appeared on most of the simultaneously working state predictors. In the last two described cases the evolutionary core can be used for evolving state automata which can replace the worst state predictor in the prediction unit.

6. CONCLUSIONS

While evolving one-level finite-state predictors, we have achieved at least the same performance as the 4-state saturated counter in predicting code the predictor was trained on. In cases the code matched the code class, which the predictor was trained on, the results were good. But when switching the code to other class, the predictor decreased its performance.

The dynamic predictor system which uses evolution for adaptive prediction of branch direction performs better in comparison with evolved one-level 4-state predictor. However, the dynamic system cannot compete in its performance and complexity with a much simpler design of a 4-state saturated counters bank. The evolutionary system is much more complicated to implement.

In comparison with the artificial neural network branch prediction systems [5, 4, 2] the evolutionary system cannot compete with its accuracy. The neural network prediction systems achieve 75% - 90% accuacy. This high accuracy was not reached with the system desribed in this paper. But there exists a possibility to improve the evolutionary dynamic prediction system performance. In future work a more complex prediction unit can be implemented. This unit will use a collection of predictors and choose the currently best predictor to be active. The evolutionary algorithm will be used to improve the performance of the worst predictor in the collection.

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* and the Research intention No. MSM 0021630528 – Security-Oriented Research in Information Technology.

Table 7: Success rate experimental results for various combination of history buffer length and burst lengths. Experiments performed with prediction of *java* code.

buffer size	burst length												
	10	20	40	80	120	160	200	240	280	350	400	450	500
10	**0.78**												
20	0.70	0.77											
40	0.70	0.72	0.76										
80	0.69	0.71	0.73	0.75									
120	0.68	0.71	0.73	0.73	0.74								
160	0.68	0.71	0.72	0.73	0.73	0.74							
200	0.68	0.71	0.72	0.73	0.73	0.73	0.74						
240	0.67	0.70	0.72	0.73	0.73	0.73	0.74	0.74					
280	0.67	0.70	0.72	0.73	0.71	0.73	0.74	0.73	0.74				
350	0.67	0.70	0.71	0.72	0.73	0.73	0.73	0.73	0.73	0.74			
400	0.67	0.70	0.71	0.72	0.72	0.73	0.73	0.73	0.74	0.73	0.74		
450	0.67	0.70	0.71	0.72	0.73	0.73	0.73	0.73	0.73	0.73	0.74	0.74	
500	0.66	0.69	0.70	0.72	0.72	0.72	0.72	0.73	0.73	0.73	0.73	0.73	0.74

7. REFERENCES

[1] E. Damiani, A. Tettamanzi, and V. Liberali. On-line evolution of fpga-based circuits: A case study on hash functions. *The First NASA/DoD Workshop on Evolvable Hardware*, 1999.

[2] C. Egan, G. Steven, P. Quick, R. Anguera, F. Steven, and L. Vintan. Two-level branch prediction using neural networks. *Journal of Systems Architecture*, 49(12):557–570, December 2003.

[3] K. Glette, J. Torresen, and M. Yasunaga. An online ehw pattern recognition system applied to sonar spectrum classification.

[4] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20:369–397, 2002.

[5] A. A. Rustan. Using artificial neural networks to improve hardware branchpredictors. *International Joint Conference on Neural Networks*, 5:3419–3424, 1999.

[6] L. Sekanina. *Evolvable Components: From Theory to Hardware*. Springer-Verlag, Berlin Heidelberg, 2004.

[7] K. Slaný and V. Dvořák. Evolutionary designed branch predictors. *13th International Conference on Soft Computing*, pages 18–23, 2007.

[8] J. E. Smith. A study of branch prediction strategies. *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, 1981.

[9] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. *Proceedings of the 29th annual international symposium on Computer architecture*, pages 25–34, 2002.

[10] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. *International Symposium on Microarchitecture*, pages 51–61, 1991.