Automatic Synthesis of Branch Prediction Schemes through Genetic Programming

Ciprian Cândea, MSE
Artificial Intelligence Research Group
Reconstructiei Nr. 2A Sibiu, Romania
40 69 22 41 68, ciprianc@airg.verena.ro
Marius Staicu, BEng
Artificial Intelligence Research Group
Reconstructiei Nr. 2A Sibiu, Romania
40 69 22 41 68, mariuss@airg.verena.ro
Lucian N. Vinţan, Professor
University "L. Blaga", Department of Computer Science
Emil Cioran 4 Sibiu, Romania
40 69 21 27 16, vintan@jupiter.sibiu.ro

Abstract

As processor architectures have increased their reliance on speculative execution to improve performance, the importance of accurate prediction of what to execute speculatively has increased. Our approach was to use a simple algebraic-style notation that allows one to describe branch predictors and also the feedback-process necessary to improve the prediction. This notation, developed by Joel Emer and Nikolas Gloy at Harvard University, allows the description of a wide variety of branch predictors in a uniform way. Also this notation facilitates the use of an efficient search technique called Genetic Programming, which is loosely modeled on the natural evolutionary process, to explore the design space. We expanded the initial notation to allow a more detailed description of modern branch predictors. In this paper we describe our notation version, the modeling system used and the first results of the application of genetic programming to the design of branch predictors.

Key Words: Branch Prediction, Genetic Programming, Simulation

1. Introduction

Every year it has been ascertained a growth of power for processing with about 50 % starting with 1990 [1]. However, for current utilization of processors it has been found a growth of performance's programs less then reported by producer. This phenomenon appears due to the problems that limit the execution's speed of programs and, up to this moment, aren't completely removed from the new processors (branches penalties, data hazards, cache misses etc). For solving this kind of problems (jumps) in this moment the dynamic branch prediction technique is the most largely used [8].

The first efficient approach in hardware (dynamic) branch prediction consist in Branch Target Buffer (BTB) structures. BTB is a small associative memory, integrated on chip, that retains the addresses of the most recently executed branches, their targets and optionally other information (e.g. target opcode). Due to some intrinsic limitations, BTB's accuracies are limited on some benchmarks having unpropitious characteristics (e.g. correlated branches).

In order to improve BTB's efficiency, Yeh and Patt (1992) and independently Pan et al (1992) generalized it through a new approach called Two Level Adaptive Branch Prediction. According to [6], The Two Level Adaptive Branch Prediction uses two distinct levels of branch history information to make predictions. The first level consists in the History Register (HR), that contains the last k branches encountered (taken/ not taken) or the last k occurrences of the same branch instruction. The second level consists in the branch behavior of the last l occurrences of the specific pattern of these branches. It is implemented by a Pattern History Table (PHT), that contains essentially the branch prediction automaton (usually 2 bit saturating counters) [5,6,8].

2. Research methodology

Our approach in order to solve this problem was to search for new branch predictors using a genetic search method. This idea was taken from Ermer and Gloy work, in this sense we used a modified BP language proposed by them [1]. This language (BP) is an algebraic-style branch predictor description based on the observation that every predictor can be divided in several elementary blocks such – memories, function and numbers (PC etc.).

The actual predictors can be divided into two distinct classes: **static** and **dynamic**. In the case of a static predictor the prediction is always the same logical function. On the other hand, **dynamic** predictors learn to make better predictions using information that is only available after the prediction is made. Dynamic predictors thus use feedback to learn from past behavior and hence make better predictions in the future. At this moment we can find new predictor structures, which contains ideas from AI fields like neural predictor [8].

In order for such a feedback control system to learn, it needs some sort of memory. To provide this memory was defined as a primitive, the structure in Figure 1. This primitive is basically a memory that is **w** bits wide and **d** entries deep. This memory has to type of operations: **predict** and **update**. The **predict** phase consist in the moment when the memory is accessed at address index I, and the value read is used as the prediction P. Some time later, when the prediction resolves, an update value U is delivered to the predictor and written into the same location indexed by I. In this moment we have implemented and tested two types of memories: P – a linear memory and A – a full associative memory.

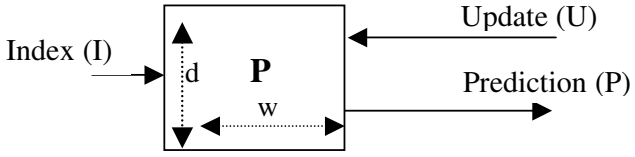

Figure 1: Primitive predictor

In Figure 1 we presented schematic our memory so we can write that like an algebraic expression:

$$P\,[w, d]\,(I; U)$$

where,

Name	Description	Type
w	Width	Static
d	Depth	Static
i	Index for prediction and update	Dynamic
u	Update value	Dynamic

Table 1: Notation used

The static parameters allow us to describe a class of predictors of various sizes with one definition. The dynamic parameters is turn partitioned between the input arguments, listed first, and the update arguments, listed after the semicolons (;). Use of this predictors can be thought of as inputting a series of index (I) and update (U) values and generating a series of predictions (P). By using specific values or expression as the input to the predictor, we can generate a variety of predictors. The classical 1 bit predictor can be represented as:

Onebit[d](PC;T) = P[1,d](PC;T)

where,

PC = current Program Counter (index value) T = branch resolution (update value) (0-> not taken, 1 -> taken)

We define the predictor "Onebit" with parameter d, input PC and update values T this predictor can be parameterized by its depth, d. For example if we choose d, 2048 (2K) we can write:

Onebit[2048](PC;T) = P[1,2048](PC;T)

In the same manner we can build more complex structures. Thus we can define an array of n-bit saturating counters each of which counts up or down based on their update value.

Counter[n,d](I; T) = P[n,d](I; if T then P+1 else P-1)

The adding and subtraction operations are in this case saturating operations, but we have tested also the arithmetic one. If we combined this counter with function MSB – which return the most significant bit of a value we can write:

Twobit(d)[PC; T) = MSB(Counter[2,d](PC; T)) and respectively

Easily now we can describe other very used primitives: global history.

Hist[w, d](I; V) = P[w, d](I; P CAT V)

CAT represents the concatening function, and we concated the current history value with the update value. In this moment to describe the well-known branch predictors proposed by Yeh and Patt [6], so called GAg, PAg and PAp is straightforward.

Gag[n](; T) = Twobit[2^n](Hist[n,1](0; T); T) | PAg[n, d](PC; T) = Twobit[2^n](Hist[n, d](PC; T);T)

and with a simple modification the PAp scheme

PAp[n,m,d](PC; T) = Twobit[2^m](PC CAT Hist[n,d](PC; T); T)

Our versions of BP language have defined function, memories and terminals. In the next two tables we presented our functions and terminals.

Function Name	Description
MSB	Most significant bits
CAT	Concatenation
XOR	Xor function
MASKHI	Most n significant bits
MASKLO	Most less n significant bits
SUB	Saturating subtraction (0 val. min)
ADD	Saturation adding
EQU	Equality of tow values
IF	If
THEN	Then
PLUS	Adding
MINUS	Subtraction

Table 2: Functions

Terminal Name	Description
T	Taken/Not Taken value read from trace file (1 bit value)
PC	Current program counter
A	Refer first A type memory definition
P	Refer first P type memory definition
0	0
1	1

Table 3: Terminals

In our approach we based our automatic search for predictors on Genetic Programming [2,3]. Genetic programming is derived from Genetic Algorithms that are a method for efficiently searching extremely large problems spaces.

A genetic algorithm encodes potential solutions to a given problem as fixed - length bits. In our representation each bit represents a language atom (function, memory, terminal) from BP language. To generate the initial population is necessary to create the individual. Each individual is created using a random algorithm. Individuals are represented by a tree structure, which is easily translated into a corresponding expression in the BP language.

We evaluate the **fitness** – prediction accuracy in this case - of each individual by computing a metric that reflects how well the solution encoded solves problem. This metric might be the cost of solution, or a measure of how close an individual gets to achieve a particular task. Next step is to create next generation in the evolutionary process is create new individuals from old ones by applying **genetic operators** that recombine the components of the old individuals in different ways. This process of combining pieces of solutions to form new solution is one of the key features of genetic algorithms. The other key feature is the way in which the fitness of an individual influences its propagation in future generations. The individuals that serve as input to the genetic operations are chosen with a probability based on their fitness value. Individuals with a higher fitness value have a higher probability of begin chosen, so that they may appear many more times than individuals of lower fitness value. This means that the next generation will contain many individuals that contain one or more components from successful individuals that contain one or more components of the previous generations, which makes it likely that the average fitness of the new generation will be better than that of the previous generation. By repeating this process many times, we produce a sequence of successive generations. Our genetic program mainly consists in the following steps:

1. Create initial population of randomly generated individuals
2. Rank fitness of individuals in the population by simulation

3. Apply genetic operations to create new generation
4. If no stop condition (e.g. if user send a stop command) go to step 2

Genetic operations that we used to populate a new generation are:

1. *Integrity check*

   With this operator we check the integrity (semantically and syntactically) of each individuals. The most important constraints are:
   - The deep of the tree is limited to an initial value (this value can be modified by user before starting the simulation). Generally, we work with a deep of 10 because our experience shown that many of the predictors that are created do not use their size efficiently.
   - Semantically and syntactically check.
2. *Replication*

   To ensure that the very best individuals of a generation are not lost or destroyed by other operation, we copy the selected (with *fitness* rate) individuals to the next generation.
3. *Crossover*

   Is the most operation it combines the components of two predictors in different ways to form two new predictors. To perform a crossover operation on two individuals our program randomly chooses a node in each of the two, and exchange the sub-trees defined by the two nodes.
4. *Mutation*

   For a node mutation program randomly choose a node within the expression tree of the individual and modify the node in the next manner: If it is a function node will be replaced with a different function. If it is a terminal node will be changed with other terminal.

In this phase of research as a fitness rate we use the prediction accuracy divided to the number of instruction simulated obtained by predictors for each simulation trace. For simulation we use the Stanford HSA (Hatfield Superscalar Architecture) traces, developed at the University of Hertfordshire, UK [7].

In the next section we describe our tools and few implementation idea of search technique used by us.

3. Implementation structures

The Branch Prediction Language (BPL) was developed in order to allow the generic description of any branch predictor. At AIRG Ltd. ("Artificial Intelligence Research Group") Sibiu, RO, we extended this language to make it more flexible. The apparition of a new memory type, an associative one, represents the biggest modification. The original BPL has only an operative memory but much of the modern branch predictors are using associative memories to store their data. This memory type can be simulated using the linear memory, the available operators (IF-THEN-ELSE, EQU) and an iteration operator, but the performances would be lowest than using a genuine one. At this moment, this memory is 'full-associative' but in the future, if required, it can be made 'n-way set associative'.

We have designed and implemented several tools to automate as much as possible of the simulation and the analyze process. These tools and the relationships between them are showed in the figure 2.

The *Branch Predictor Generator* is the heart of the whole system. It generates the genetic predictor schemes, passes them to the simulator, receives the prediction rates and stores the results in the Data Storage module for future analyses.

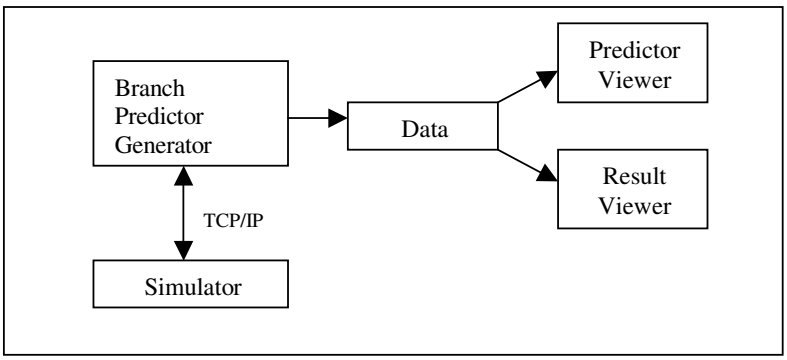

Figure 2: General System Structure

In the first generation we include some classic Two Level Adaptive Branch Predictors that were taken from configuration file [5,6]. The graphical interface allows us to customize the simulators IP addresses (see below), generation numbers, and predictors' number in a generation, the fitness rate, the predictor's general complexity (it is represented as a binary tree). Once started, the simulation (which take place in parallel on several computers) can be paused saved and resumed later or canceled.

The *Simulator* is used to verify the predictor behavior. It receives a predictor from the generator and the corresponding HSA trace file used in simulation and returns the prediction rate. If the predictor can't be simulated it will return 0. As an implementation detail, the predictor is received (TCP/IP) as a text string and is parsed in a binary tree. Each node represents an operator (MSB, MINUS, XOR…), a memory (P or A) or a terminal (T, PC or number). The node value is used in its parents as input arguments or, in the case of the root; it represents the predicted value (taken or not taken). If the node codes an operator the two children are the operator arguments and their value are obtained by evaluating the two branches. If the node codes a memory (linear or associative) the left branch represents the address in the memory space (the direct mapped address for linear memory, or the tag for associative memory) and the right branch represents the value that will update the memory. For each line in the trace files the simulator performs two basic operations: the evaluation and the update of the predictor. In the evaluation phase the predictor's memory address is obtained and the memory value is used to make a prediction; in the update phase the value at the previous calculated address is modified according with the update value. The prediction (taken/not taken) is compared with the effective value obtained from the trace file and we can see if the prediction is correct or wrong. The total number of correct predicted jumps divided by the total jumps represents the prediction rate (*fitness* rate) that is returned to the predictor generator.

The *Data Storage* is used to memorize the generated predictors and their prediction rate for future analyses. The Predictor Viewer and Result Viewer use them.

The *Predictor Viewer* is a tool that converts the predictor from a string, as generated by the predictor generator module, to a graphical form showed as a binary tree. It can be used as a stand-alone application that take the predictor from the user in a dialog box, or as an integrated tool that take the predictor from the Data Storage module. For now the viewer

shows only a raw structure of the predictor. In the future we think it would be possible to make it able to recognize standard portions (i.e. a local history memory or a circular register) and replace them with symbols in the general picture. This will led to a major simplification of the graphic and will facilitate a better predictor structure understanding.

The *Result Viewer* module is used to show general information about the generated predictors and simulation results. There are several options (you can select the generation, the prediction rate, trace file used etc.) for filtering the information. Also, you can select individual predictors and display them in the predictor viewer.

Because the genetic search is a slow process we use a parallel search (simulation) in a distributed environment across several computers (client-server architecture). The Branch Prediction Generator acts as a client for the servers (the simulator module) that run on different computers. The communication between the client and the servers use an in-house protocol build over the TCP/IP. For now the client must know in advance the server number in the system and their IP address. In the future it may be possible to allow a server to attach/detach dynamically to/from the system. Another idea is to have two-three clients that can share information and use the Internet to communicate. In our simulation we used 3 computers PIII at 550Mhz 128Mb RAM with Windows 2000.

4. Obtained Results

For moment we tested only the branch prediction case. In our tests we try to find some new predictors type or new useful ideas. This is a very difficult slowly process because for many years the branch predictors were designed by human researchers. We started with a population of 400 chromosomes. In these populations 6 were classic predictors like those presented in Table 4 (right) and the rest where randomly generated. For each generation we kept all the predictors, which have a prediction rate greater than a user defined threshold and use them to derive the next generation.

In Table 4 we published the results obtained by simulating 5 classical predictors, which are usually found in modern processors. The prediction rate is computed as the arithmetical mean of the rates obtained from the simulation of predictors on all available traces. Table 5 shows the results from the simulation of the best predictors founded by our system. These predictors where founded starting with the 5th generation.

As it can be observed, the rates in the two tables are about the same. This means that the automatic generated predictors are quite similar from the performance point of view with the human-designed ones. We should add that our crossover operator isn't so efficient yet. We have noticed that, in our opinion, only after at least 10 generation it begins to appear chromosomes with a somewhat improved prediction rate [1]. Until now, because it takes a long time to simulate 10 full generations we didn't pass this boundary. On our systems the simulation of one chromosome take between 15 and 100 seconds, depending on its complexity and on the trace dimensions, with an average of 60 seconds. It takes around 12 hours to simulate a population of 400 chromosomes. The available resources don't allow us at this moment to continue the simulations for more generations and with a larger population. We'll focus on improving the genetic operators and the simulation techniques in order to improve the simulator performance and decrease the simulation time.

Based on the obtained results we can conclude that our developed method is feasible. We think it is necessary to grow the analyzed generation numbers and to use other, more complex, trace files (our trace files have an instruction address space of about 300 32-bit words).

Predictor	Rate (mean)
MSB(MASKHI(ADD(A[23,64](0;T);A[1,128](PC;PC));0))	0.711
MSB(EQU(MSB(MASKHI(ADD(IF(XOR(0;1);THEN(PC;PC));SUB(MSB(A[22,64](1;1));MASKLO(0;EQU(PC;PC))));PC));0))	0.759
MSB(EQU(XOR(SUB(MINUS(0;1);IF(ADD(A[31,64](A;PC);1);THEN(1;PC)));EQU(PC;PC));ADD(A[24,32](0;1);MASKHI(MSB(1);PC))))	0.811

Predictor	Rate (mean)
Onebit[1,2K]	0.664
Twobit[2,64K]	0.746
GAg[2]	0.849
PAg[18,8K]	0.825
PAp[9,18,8K]	0.771

Table 4: Branch Predictor Performance

5. Conclusion and Further Work

In this paper, we have presented our work based on some improvements of the BP language proposed by Ermer and Gloy [1] and the related developed tools used. Using this language we can describe a variety of predictors and also manipulate them very easy. We also repeated the experiments based on automatic search for branch prediction schemes using Genetic Programming concepts. The first results point out that the predictors created automatically are directly comparable with the "hand made" branch predictors used today, but they are more complex and, as a consequence, probably difficult for being practical implemented. On the other hand, it would be possible to find some interesting new ideas useful in human designed branch predictors. In the nearest future we intend to extend our tools but also we'll try to extend the language with other components. These search techniques we'll try to adapt in order to find other types of predictors like predictors for indirect jumps. Anyway, we think it would be possible to discover some very efficient genetic predictors.

These first positive results obtained let us think that opportunity to find the new viable structures is very height, besides improvement and adding of other genetic operators and, eventual the development of an algorithm for reduce the complexity.

6. References

[1] Ermer, J. and Gloy N.: *"A Language for Describing Predictors and its Application to Automatic Synthesis"*, Int'l Symp. On Comp. Architecture, ISCA '97, 1997.

[2] Holland, J.H.: "Adaption in Natural and Artificial Systems", University of Michigan Press, 1975.

[3] Koza, J.R.: *"Genetic Programming",* MIT Press, 1992.

[4] Sechrest, S., Lee, C-C and Mudge, T.: *"The Role of Adaptivity in Two-level Adaptive Branch Prediction",* 28th ACM / IEEE International Symposium on Microarchitecture, November, 1995.

[5] Yeh T.-Y. and Patt Y.N.: *"Two-Level Adaptive Branch Prediction",* 24th ACM / IEEE International Symposium on Microarchitecture, November, 1991.

[6] Yeh T.-Y. and Patt Y.N.*: "Alternative Implementation of Two-Level Adaptive Branch Prediction",* 19th Annual International Symposium on Computer Science, May 1995.

[7] Steven G.: *"A Superscalar Architecture to Exploit Instruction Level Parallelism",* Microprocessors and Microsystems, No 7, 1997.

[8] Vintan, L.: *"Instruction Level Parallel Architectures"* (in Romanian*)*, ISBN 973-27-0734-8, Romanian Academy Publishing House, Bucharest, 2000.