A Language for Describing Predictors and its Application to Automatic Synthesis

Joel Emer
Digital Equipment
Hudson, MA 01749
emer@vssad.hlo.dec.com

Nikolas Gloy
Harvard University
Cambridge, MA 02138
ng@eecs.harvard.edu

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. Furthermore, the types of values predicted have expanded from the ubiquitous branch and call/return targets to the prediction of indirect jump targets, cache ways and data values. In general, the prediction process is one of identifying the current state of the system, and making a prediction for some as yet uncomputed value based on that state. Prediction accuracy is improved by learning what is a good prediction for that state using a feedback process at the time the predicted value is actually computed. While there have been a number of efforts to formally characterize this process, we have taken the approach of providing a simple algebraic-style notation that allows one to express this state identification and feedback process. This notation allows one to describe a wide variety of predictors in a uniform way. It also 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. In this paper we describe our notation and the results of the application of genetic programming to the design of branch and indirect jump predictors.

1. Introduction

In the quest for more CPU performance, there is ever greater use of strategies to increase instruction-level parallelism, including deep pipelines, super-scalar issue, and out-of-order issue. In order to achieve a performance benefit, these techniques have resulted in an increasing reliance on **speculative execution**, that is, executing operations before all the input values are known.

To appear in the Proceedings of the 24th Annual International Symposium on Computer Architecture, Denver, Colorado, June 2-4, 1997.

The standard technique for coping with unknown input values is guessing the value, using the guessed value in the speculative operation, and eventually resolving whether the guess was right or not. If the guess was right, then the computation can proceed. If the guess was wrong, however, the processor needs to reset its state back to the point before the guess, and resume with the correct input values. This process of generating a guess is more formally called **prediction**.

Branch prediction is the most commonly seen form of prediction [14,17,18,11]. In this case, the value being predicted is the instruction to execute after a branch instruction, e.g., either the target of the branch or the next sequential instruction. This information is needed very early in the pipeline, so that instruction fetch can be directed to fetch the correct instruction. On the other hand, the branch result will not be determined until the branch executes far down the pipeline. Thus, accurate branch prediction can substantially improve performance by allowing for the speculative execution of instructions following the branch before the branch resolves.

While branch prediction is probably the best known form of prediction, there are many other cases where one can predict either architectural or implementation values. Some examples are: indirect jump target prediction [6,8], return instruction target prediction [5], cache way prediction [2], cache miss prediction, and data value prediction [9]. In each of these cases, a value is predicted and used speculatively pending actual computation of the value.

Previous work on categorizing and characterizing predictors has been based on the high-level constructs from earlier research on branch prediction mechanisms, such as global-history components or tables of saturating counters [12,13,18,19]. This seems logical, particularly for an automated search, because a predictor composed from such constructs is quite likely to resemble a known good predictor, whereas a predictor built arbitrarily from lower-level primitives is less likely to be useful.

Using higher-level constructs, however, means that the only predictors we are ever going to see are made from

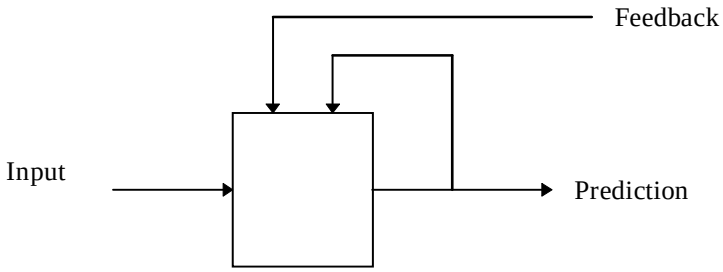

Figure 1 - Basic Prediction Feedback Loop

these familiar building blocks. We are therefore losing the chance to find interesting new components of predictors that we had not previously thought of. Another drawback is that while we know many good building blocks for branch predictors, this may not be the case for other prediction problems. So for those problems, we would likely be more successful starting from lower-level primitives. Therefore, rather than base a model of prediction on the various existing predictors, we define a simple primitive predictor.

The model we have developed allows a large variety of predictors to be described using a language with a simple algebraic-style syntax. Our aim is to formalize the process of specification of predictors in part to facilitate the search of the design space. Previously, this has been essentially a manual process, i.e., dependent on the creativity of the researcher. We aim to provide automatic help.

Thus, given this algebraic-style description for predictors, we can employ automated search techniques in order to find new predictors. In particular, the set of techniques known as genetic programming allows one to search a general expression design space [7]. It is a stochastic technique that is well suited to situations where we do not know much about what solutions might look like, since it finds increasingly better solutions from a starting point that can be completely randomly generated.

In Section 2 we provide a framework that can be used to describe a wide variety of predictors, and a corresponding algebraic notation that can be used to simply describe these predictors. In Section 3 we provide an overview of genetic programming, and in Section 4 we describe how we apply it to the task of finding predictors. In section 5 we describe the results of our genetic programming experiments to find branch predictors and indirect jump predictors. Then finally in Section 6 we conclude with some observations.

2. Predictor Notation

The principal function of any predictor is to take an input, corresponding to the current state of the system, and generate an output that predicts some as yet uncomputed value. Note that this input can be as all-encompassing of system state as desired. Thus, while most branch predictor studies have been based on just the current program counter, PC, as the input for the predictor, there is no reason not to use other information, such as the opcode or sign of the branch offset (since backward branches are likely to be part of loops, and hence more likely taken). So in general one should consider using all the information available in earlier pipe stages of the current or later instructions as well as information available from earlier instructions in later pipe stages [2].

Predictors can be divided into two classes: **static** and **dynamic.** In the case of a **static** predictor the prediction is always the same logical function of the input to the predictor. On the other hand, **dynamic** predictors *learn* to make better predictions from 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.

The natural, but not necessarily required, time to feed information back to the predictor is when it is determined whether the prediction was correct or not. This is referred to as the time the prediction **resolves.** Thus, frequently, the information fed back to the predictor is simply the accuracy of the prediction. So, in the case of a branch predictor, whether the branch was taken or not can be fed back to the predictor to help it improve its predictions. In addition, however, other information might also be provided. This overall dynamic prediction scheme can be viewed as a feedback control system as illustrated in Figure 1.

In order for such a feedback control system to learn, it needs some sort of memory. To provide this memory we define, as a primitive, the structure in Figure 2. This primitive is basically a memory that is *w* bits wide and *d* entries deep. As with a typical memory, it has two operations. For our purposes, however, rather than read and write, the two operations are called **predict** and **update**, and furthermore these two operations are always used as a pair. Thus, in our example, the operation of the predictor consists of a **predict** step in which 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

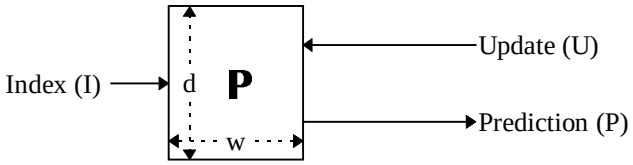

Figure 2 - Primitive Predictor

and written into the same location indexed by I. A trivial extension might involve writing back to a different location, but we will not consider that case. Similarly, we do not consider the case where there might be multiple predictions before there is an update.

We will represent the primitive predictor in Figure 2 as an algebraic expression:

P[w, d](I; U)

where,
w = width
d = depth
I = index for prediction and update
U = update value

Note that we use a notation in which the static configuration parameters of the predictor are enclosed in square brackets ([]). These parameters allow us to describe a class of predictors of various sizes with one definition. A specific instance of a predictor has constant values specified for these parameters.

Following the configuration parameters are the dynamic arguments used to generate predictions. These are enclosed in parentheses (()). These arguments are in turn partitioned between the input arguments, listed first, and the update arguments, listed after the semicolon (;). Other predictors defined in terms of this primitive predictor will also follow this notation.

Use of this predictor 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 expressions as the inputs to the predictor, we can create a variety of predictors. Thus a simple 1-bit branch predictor that predicts a branch will behave the same this time as it did last time can be represented as:

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

where,
PC = current program counter
T = branch resolution
(0 -> not taken, 1 -> taken)

This expression defines the predictor "Onebit" with parameter d, input PC and update expression T as the expression on the right of the equal sign (=). The expression on the right describes the size and how to "wire up" the primitive predictor P to create "Onebit." Note that this predictor is parameterized by its depth, d. Therefore, to specify a specific predictor, like that in the DEC Alpha 21064, we can use Onebit as follows:

A21064[](PC; T) = Onebit[2K](PC; T);

We can also build up more complex components out of the simple predictor that can be used to build even more sophisticated predictors. Thus we can define an array of n-bit saturating counters each of which counts up or down based on their update value, such as described in [14] as:

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

Note that in this case we use the value generated by the predictor, P in this example, as part of the expression for the update expression. We also have assumed that our addition and subtraction operations are saturating operations. Using this predictor and a function MSB that returns the most-significant bit of a value, we can easily create the ubiquitous 2-bit counter predictor as:

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

Another useful primitive is one that keeps a history of some value by concatenating the current history value with the update value. The following expression describes a predictor that maintains an array of histories of values:

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

Using this primitive one can create a variety of predictors such as these global and local history or two-level adaptive branch predictors from Yeh and Patt [18]:

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);

A simple modification of the index expression leads to the PAp scheme from Pan, So and Rameh [11]:

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

Similarly, McFarling's GShare predictor [10] can be expressed as:

GShare[m](PC; T) =
Twobit[2^m](PC ⊕ Hist[m, 1](0; T); T);

Finally, what may be the most complex commercially implemented predictor, the choosing predictor [10] used in the DEC Alpha 21264. This predictor consists of a local history predictor and a global history predictor that are selected between by a global history based chooser:

Lhist[](PC;T) = Threebit[1K](Hist[10,1K](PC;T); T);
Ghist[](;T) = Twobit[4K](Hist[12,1](0;T); T);

A21264[](PC; T) =
if (MSB(P[2,4K](Hist[12,1](0;T);
P + (Lhist==T) - (Ghist == T)))
then Lhist[](PC;T)
else Ghist[](;T);

Where Threebit is the obvious extension of Twobit to three bit counters.

Through the composition of predictors and various logic expressions a large variety of predictors can be created. Note, furthermore, that the predictors need not be restricted to generating single bit predictions. For example, one can specify a predictor for indirect jumps. Following is an expression that builds a table for indirect jumps that predicts that each indirect jump instruction will jump to the same target it jumped to last time:

Jump[d](PC; Target) = P[32, d](PC; Target);

We have developed a parser for a version of this notation called the **BP language.** This BP language parser understands the predictor primitive and a variety of functions. It also understands another primitive, which we do not use here, that represents a set-associative tag store.

The parser translates BP language expressions into a set of subroutines that simulate the predictor that the expression describes. From there it is easy to link the predictor subroutines to a trace reader to simulate the performance of the specified predictor.

In the next section, we describe a search technique that can automatically generate predictors.

3. Genetic Programming

We base our automatic search for predictors on genetic programming [7]. Genetic programming is derived from genetic algorithms [4], so we will describe those first. Genetic algorithms are a method for efficiently searching extremely large problem spaces. Their behavior has some similarities to the way in which natural selection enables evolution to produce species that are adapted to their environment.

A genetic algorithm encodes potential solutions to a given problem as fixed-length bit strings. Initially, we generate a set of random bit strings, each of which is called an **individual** by analogy to the evolution paradigm. This set is our initial **population** or **generation**. We evaluate the **fitness** of each individual by computing a metric that reflects how well the solution encoded by its bit string solves the problem. This metric might be the cost of the solution, or a measure of how close an individual gets to achieving a particular task.

To create the next generation in the evolutionary process, we create new individuals from old ones by applying **genetic operations** that recombine the components of the old individuals in different ways. Thus, structures that are part of a good solution that have developed in some individuals can be combined with structures developed in other individuals. The resulting offspring might be an individual combining several good components and potentially achieving a higher fitness value.

This process of combining pieces of solutions to form new solutions 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 being 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 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.

The first generation, being a set of random points in the search space, will usually not contain any reasonable solutions. The fitness-based selection method will, however, try out many modifications and combinations of the slightly better individuals, which generally leads to much improved solutions within a few generations.

Genetic algorithms often work very well for problems where solutions can be encoded in fixed-length strings. They are too constraining for problems, such as prediction, where the solutions are general algebraic expressions or programs. Genetic programming, which is closely related to genetic algorithms, is better suited to these problems. The only change is that the individuals are not encoded as fixed-length strings, but as tree structures, and the genetic operations are adapted to perform analogous operations on tree structures. This allows us to easily represent algebraic expressions or parse-tree representations of programs, and it allows the individuals to grow as needed. In our application of this method, the individuals are BP language expressions that describe predictors.

4. Genetic Programming Search

To apply genetic programming to automatically synthesize predictors, we created a set of programs and scripts to perform the genetic programming search as follows:

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. Repeat steps 2 and 3

More specifically, however, to apply genetic programming to predictors one needs to map the predictor problem into the appropriate structure. First, we will describe how we represent expressions in the BP language as tree data structures. Next, we present the genetic operations that are used to produce new individuals from old ones. Then, we describe how these operations are applied to a generation of individuals to create the next generation. We also need to place some constraints on the expressions that are produced by genetic operations to ensure that the results are valid BP expressions, that they do not grow unreasonably large, and that they satisfy certain other constraints that simplify the implementation of some aspects of the operations. Finally, we present the method we use to determine the fitness of the individuals in the population.

4.1 Representation of expressions

Individuals are represented by a tree structure which is easily translated into a corresponding expression in the BP language. The tree nodes are divided into the following categories:

Predictors

A predictor node represents a primitive predictor of the BP language. It contains the width and height of the predictor, and has two descendants: The first one corresponds to the expression used to compute the index of the predictor, and the second one corresponds to the expression used to update the state of the predictor.

Functions

We currently use the following functions: XOR, CAT (concatenation), MASKHI/MASKLO (which return a given number of the high or low bits), MSB (returns the highest bit), SATUR (performs a saturating add of a given width), IF (selects one of two inputs depending on the value of a third input). If desired, it is very easy to extend this set, since the BP language contains many additional functions.

Terminals

Input values: For each class of prediction problem, there is a list of the arguments to a predictor. Arguments are separated into two classes: the **input** values that are available immediately, and the **update** values that are available after the value being predicted has been computed. For branch prediction, the **inputs** usually include the PC of the branch instruction, but other processor state can also be useful, such as the branch direction (the sign bit of the branch offset). The **update** value for branch prediction is typically the branch outcome.

Value of a predictor: An expression can be a reference to the value of another predictor node. To simplify the implementation of other parts of the system, it was expedient to allow only references to the nearest enclosing predictor. This restricts the class of predictors that can be generated, and we plan to remove this restriction in the future.

Integer constants: We allow small integer constants to be available in expressions.

4.2 Genetic Operations

In this section, we describe the genetic operations that are used to populate a new generation.

Replication

To ensure that the very best individuals of a generation are not lost or destroyed by other operations, we copy a certain number of the best individuals to the next generation.

Crossover

The most important operation is one that combines the components of two predictors in a different way to form two new predictors. To perform a crossover operation on two individuals, we randomly choose a node in each of the two, and exchange the subtrees defined by the two nodes.

Mutation

There are two kinds of mutation operations; their purpose is to make small changes to individuals, which can sometimes be needed to prevent the entire generation from converging to identical individuals. We apply mutation operations to the offspring generated by crossover operations.

For a node mutation, we randomly choose a node within the expression tree of the individual, and modify the node as follows: If it is a function node, we replace it with a different function; if that function needs more arguments than the original function, we create random expression trees as needed. If it is a terminal node, we replace it with another terminal. If it is a predictor node, we change the width and/or height of the predictor.

For a subtree mutation, we randomly choose a node and replace the subtree defined by the node with a randomly generated subtree of identical height.

Encapsulation

Crossover operations are necessary to combine useful components of individuals, but they can also be destructive: They may break up a useful component of an individual into parts that are meaningless by themselves. An encapsulation operation makes a randomly chosen subtree of an individual into an indivisible unit that cannot be broken apart by a crossover operation or mutation, although it can still be moved in its entirety [1]. For a randomly chosen encapsulation point, there is no way to know that this encapsulated component is actually useful. However, if it is useful, then individuals that receive this component through crossover will tend to improve, and the fitness-based selection and replication will make those individuals more frequent in the next generation. On the other hand, if the component is useless, then individuals incorporating it will tend not to be successful, and the component will not propagate and eventually die out.

There is an opposite operation, called expansion, that turns an encapsulated subtree back into a regular subtree. This way, encapsulated expressions still have a chance to improve by taking part in other operations.

For each of these operations, the individuals serving as inputs are chosen using a method called **tournament selection**. To choose one individual by this method, we first choose a set of individuals randomly and uniformly from all the individuals of the current generation; the size of this set is a parameter called the **tournament size**. Then we find the individual in that set that has the best fitness value, and discard all the other individuals. The result is that individuals are chosen according to a probability distribution that gives higher probability to individuals with higher fitness values, while less fit individuals still have some chance to be chosen occasionally. How much this probability distribution is biased towards better individuals is determined by the tournament size parameter. For a generation size of 400 individuals, we typically use a tournament size of 8.

4.3 Constraints

In order to produce legal and usable individuals, we need to impose certain constraints on the results of genetic operations. In most cases, we achieve this by first allowing the operations to proceed oblivious to the constraints. We then check the result for compliance with these constraints, and modify the individuals where necessary.

Our first constraint is to avoid generating predictors of excessive implementation size (i.e. the number of storage bits it takes to implement the predictor). To keep the predictors from growing indefinitely, we limit the implementation size of any predictor to 512K bits of storage. When a predictor exceeds this bound, we reduce its size until it falls below this limit: we randomly choose a predictor node within the expression tree, and reduce either its width or its height by one step. Experience has shown that many of the predictors that are created do not use their implementation size efficiently; for example, many predictors contain subexpressions that contain a large predictor table whose value does not affect the output of the entire expression. Therefore, the somewhat generous limit of 512K bits was deemed appropriate.

Our second constraint is making sure that the expression is a legal BP expression. For example, during crossover a terminal node that refers to the value of an enclosing predictor may be moved outside of the update subtree of that predictor. In this case, if there is another predictor enclosing the new location, the node will now automatically refer to the nearest enclosing predictor in its new context. If there is no such predictor, the node is changed to a constant value of 1. Another problem occurs when a terminal node that represents an update value is in a context where that value is not available, e.g., the branch outcome cannot be used in an indexing expression of a branch predictor.

Because the widths of values in the BP system are important, when generating predictors the system must explicitly specify the widths of each expression. As a simplification, we currently only allow expressions whose width can be statically determined. In most cases, these can be determined from the widths of the subexpressions, except where there is a recursion. In those cases, we pick a width at random.

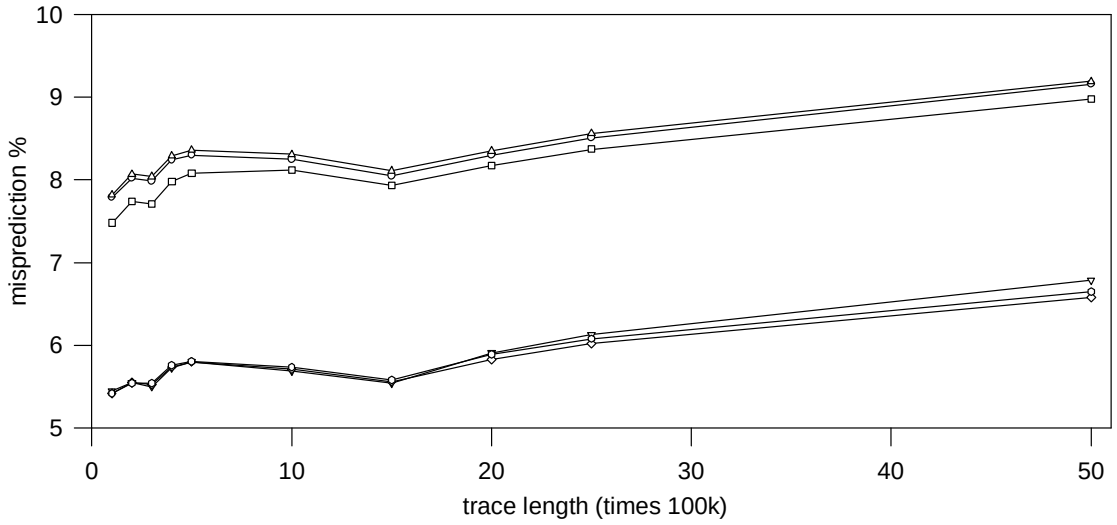

Figure 3- Performance of Branch Predictors vs. Trace Length

As Koza [7] notes, one often needs to coerce results into a particular form for the problem being studied. We found this to be true, so we developed a way of using templates for the individuals of the initial generation, to provide a reasonable starting point. For example, one could use a template that is a predictor with a random indexing expression and 2-bit saturating counter as the update expression. It is possible to have templates where the user-provided components are forever fixed, or the entire individual may be changed by genetic operations. Usually, generating the more complex forms of branch predictors depended on starting with more advanced structures in the template.

4.4 Fitness

For the predictors we study, we use the accuracy of the predictor, expressed as a misprediction rate, as the fitness metric. To calculate the misprediction rate, we use the BP language parser to create simulators of predictors from these BP language expressions. We run those simulators over instruction traces with a parameterizable training period to avoid startup effects. In our experiments, we used Atom [3] generated traces from some SPEC92 and SPEC95 [15] integer benchmarks (compress, eqntott, gcc, go, m88ksim, xlisp) compiled for the DEC Alpha on Digital Unix with the switches set as submitted to SPEC. We also used some traces from the IBS benchmark suite [16]. The simulation output consists of a prediction accuracy and fitness ranking for each predictor.

Simulating the execution of an entire benchmark every time we need to evaluate the fitness of the hundreds of predictors in a generation would make it unbearably slow to produce the 20-30 generations that are usually needed to obtain good results. Thus, we use shortened runs to generate fitness values.

Fortunately, the fitness measure is used only as an input for the tournament selection process, which needs only the relative ranking of the individuals and furthermore is a probabilistic process that is unlikely to be affected by slight inaccuracies in the rankings. Therefore, we do not need a perfectly accurate fitness value, but only a roughly accurate ranking of the individuals in a generation. To generate this ranking, we simulate until we find a ranking order that is reasonably stable. Typically, this requires simulating about one million branches.

Note, however, that if we use a single trace for an entire benchmark, we would repeatedly use only the first million events from the trace. Instead, we use several shorter samples from the entire trace. Each sample contains 5 million predicted events, and for our experiments we had 2 or 3 samples from each benchmark.

In our experiments, we used two different methods for ranking predictors. In the first method, we ranked the individuals in a generation based on just one randomly selected benchmark, i.e., one benchmark per generation. This method was relatively fast, and might correspond to looking for individuals that can survive though successive eras of drought, flood and temperate weather.

In our second method, we determine the rankings for each benchmark in a set of benchmarks and then combine the rankings for each benchmark into a single average ranking. This second approach is much more time-consuming, but it avoids the problem of only one benchmark influencing the outcome of the next generation. In either case, final evaluations of the predictors are based on longer runs of the SPEC benchmarks and IBS traces.

It remains to be shown that the relative fitness determined using short traces is actually a good measure

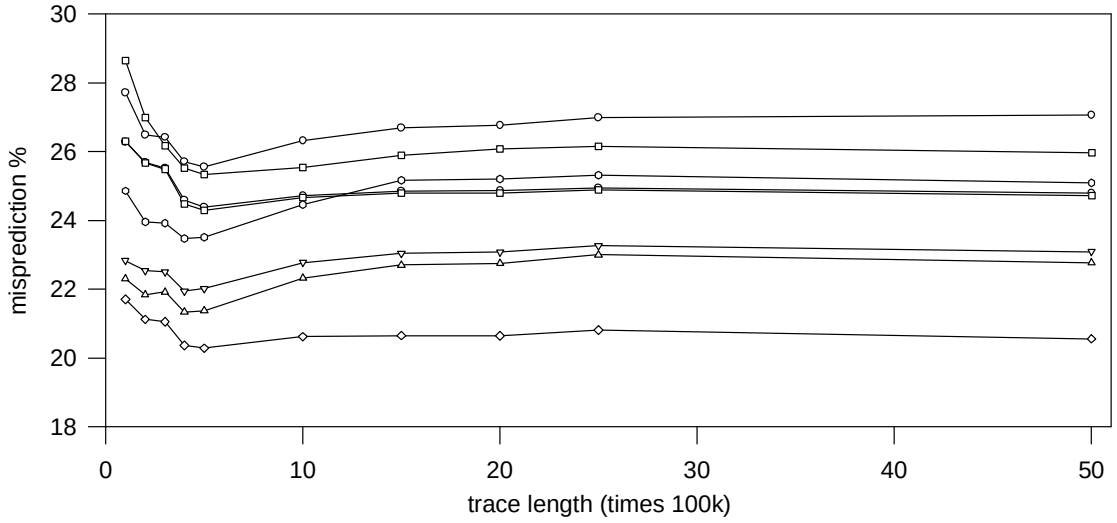

Figure 4 - Performance of Indirect Jump Predictors vs. Trace Length

of the relative fitness for longer traces of benchmark execution. We verified this by selecting some of the better predictors from our experiments, and evaluating their performance over traces of varying length.

The results of these experiments are presented in Figures 3 and 4. These figures show the absolute and relative performance for several branch and indirect jump predictors as a function of the number of predicted events in the trace. Each line represents a different predictor, where each point is calculated by averaging the results of 4 benchmarks: compress, gcc, go, xlisp.

These curves show that over time the relative ranking of the predictors is quite stable (or indistinguishably close), even though the absolute performance varies due to the differing behavior of the benchmark programs over time. Thus, we believe that rankings generated from the short traces form an adequate input to the tournament selection process.

5. Results

5.1 Branch Predictors

We chose branch prediction as the first test case for automated synthesis of predictors. Because we had previous knowledge of branch predictors, we knew what kinds of structures are promising and what kind of prediction accuracy can be achieved by a good predictor. This gave us the opportunity to tune the system to get generation to generation improvements. On the other hand, given the many years of research on human designed branch predictors, the task of finding new ones is more difficult.

Figure 5 shows the development of the best and average individuals over 16 generations of one experiment; fitness calculations were based on the average of 8 SPEC traces, and the resulting average misprediction percentages are shown for each generation. Each generation contains 400 individuals.

The graph shows a steady improvement in the performance of the best predictor. As in this example, somewhere between 15 and 30 generations the experiments usually converge to a few distinct predictors, and the subsequent generations do not achieve any significant improvement. Typically, this indicates that the predictors that can be created from the population using crossover and mutation do not show any improvement over the best predictors in the population. Note that the average performance of a population is more variable and can even get worse if crossover and mutation generate a number of useless predictors.

To see how well our automatically generated predictors fared, we have taken 6 of the better predictors produced in the course of our experiments, called GP1 through GP6, and 6 well-known human-discovered predictors. The configurations for the human-made predictors are specified using the predictor definitions from Section 2. In all cases we tried to size the predictors to about 512K bits. Table 1 shows the average performance of each predictor for the SPEC benchmarks we studied (either full runs or 50M branches whichever came first), and the average for the IBS traces.

The results show that the automatically generated predictors compared favorably to the human-generated versions. In fact, three of the predictors were better than all but the GShare predictor. Note, however, that we were not yet able to create any choosing style predictors, due to the current limitations of our crossover operation.

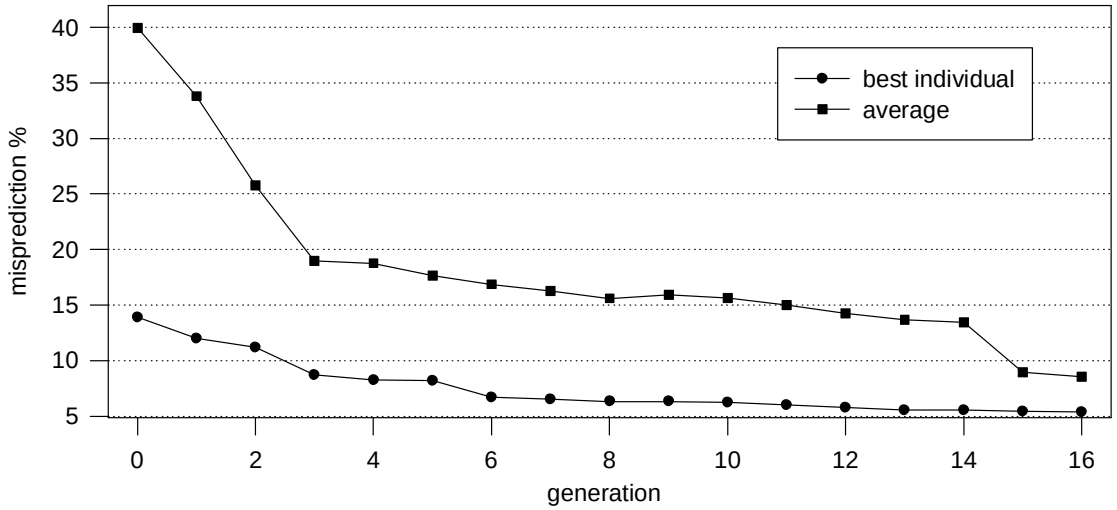

Figure 5 - Fitness vs. Generation

We have not shown the predictor expressions for GP1 to GP6, because the predictors that are generated tended to grow to the point where they consist of very deep tree structures. Thus, even though in size these predictors are comparable to the human-designed predictors they are logically much more complex, and probably not directly implementable.

The complexity of these automatically-created predictors is actually a natural consequence of the genetic programming process. Genetically created solutions are typically not very efficient, consisting of many apparently useless expressions. These expressions, called "introns" in the genetic programming literature, appear to be needed to protect the developing expression from crossovers. Thus, defining a fitness function biased against this sort of complexity may not work. In any case, we believe that there is value in just manually identifying the pieces of the solution that might be contributing to their effectiveness, and manually creating better predictors.

In fact, we were able to find numerous interesting subcomponents of these synthesized predictors. Over a variety of runs of the experiment, we found the system "invented" many familiar components of branch predictors: saturating counters, global and local branch histories as well as indirect histories.

There were also some apparently new structures that could form the basis for practical predictors. Invariably these were used as part of something like a GShare-style index (a global history xor'ed with the PC [10]) for a set of two or three bit counters. Below are some examples of such predictor components.

First is a global history indexed by a global history, which keeps around a portion of the older global history when there is a series of taken or not taken branches.

P0[14,16384](P1[14,1](0; P1 || T); P0 || T)

Next is a global history biased by the direction of the branch. If one assumes that backwards branches are normally taken, and forward branches normally not taken, then this keeps a history of "unexpected" branch actions:

P[14,1](0; (P || (DIR ⊕ T))

Next is a predictor that keeps separate global histories for forward and backward branches, maybe indicating that it is useful to separate the behaviors of loop branches from non-loop branches:

P[5,2](DIR; P || T)

Finally, we found a predictor that keeps a short history of PC values, which could be useful for tracking the history of the program:

P0[11,32](P1[5,1024](PC<10:0>; PC<5:0>);
P0 || PC<4:0>)

5.2 Jump Predictors

In contrast to branch prediction, there has been little research on predicting the target addresses of jump instructions that use a register as the target address. We therefore have little knowledge of what a good jump predictor looks like. For the same reason, this is a more interesting search space because we can expect to find

Table 1 - Branch Predictor Performance					
Predictor	Mispredict Rate (SPEC)	Mispredict Rate (IBS)	Predictor	Mispredict Rate (SPEC)	Mispredict Rate (IBS)
Onebit[1, 512K]	17.7	10.0	GP1	9.7	5.7
Twobit[2, 256K]	13.1	6.7	GP2	9.5	5.0
GShare[18]	6.7	2.7	GP3	9.7	5.7
GAg[18]	7.9	4.0	GP4	7.2	3.0
PAg[18, 8K]	7.9	4.5	GP5	7.0	2.9
PAp[9, 18, 8K]	11.2	5.5	GP6	7.1	2.9

predictors that are better than the ones that we already know.

The inputs that are available to a jump predictor are: PC, SP (stack pointer), TREG (number of register containing the jump target), and the target itself (after the address has been resolved). The benchmarks we used for this study were: gcc, go, xlisp and m88ksim.

Table 2 below shows some of the jump prediction mechanisms that one might think of; note that we only need to predict the lower 12 bits of the word address because only targets that are in the instruction cache need to be predicted.

The best predictors seen during our genetic programming experiments achieve average misprediction percentages as low as 15% on long runs of the same set of programs; this represents a significant improvement over the three predictors in Table 2. Since the genetic programming system had already been developed and refined using branch prediction as a test case, producing these jump predictors required very little effort.

One of the simplest predictors that gets significantly better performance than the simple predictors above is shown below. The average misprediction percentage of this predictor for full runs of the 4 SPEC traces of 1 to 21 million jumps is 33.4%.

```
P0[16, 16384](PC<3:0> ⊕ SP
              P1[12,16](TREG; TARGET);
              TARGET);
```

This expression contains a predictor that stores a previous target, indexed by the target register; this is XORed with the PC and the SP to form an index into a table of previous targets.

Another example is the following more complicated predictor; its misprediction rate is 30.0%.

```
P0[16, 16384](
        SP ⊕ PC ⊕
        P1[12,32](TREG; TARGET<11:2>) ⊕
        P2[15,32](P3[5,2](1; PC<4:0>);
                  SP ⊕ TARGET ⊕
                  P4[12,32](P3;TARGET));
        TARGET);
```

While maybe too complex to implement, this expression shows some interesting subcomponents that might be useful for creating more practical predictors.

6. Conclusions

In this paper, we have presented a new language for describing predictors. This language provides a concise and unambiguous way for describing a large variety of predictors. Furthermore, it allows for their automatic manipulation, including generating simulators and automated synthesis. In particular, we have used the search technique called genetic programming to search the design space for branch and jump predictors.

The result of these experiments has been the creation of branch predictors comparable to the best non-choosing

Table 2 - Indirect Jump Predictor Performance		
Description	BP expression	Misprediction percentage: average of 4 traces, 35M jumps
use the target of the previous jump	p[12,1](1: target)	63%
table of previous targets, indexed by PC	p[12,4096](PC : target)	47%
table of previous targets, indexed by PC and SP	p[12,4096](PC[9..0] || SP[4..0] : target)	54%

predictors that have been published. Although these predictors are logically complex, analysis of their structure revealed a number of interesting subcomponents that might be used to develop implementable predictors. In addition, we created relatively simple indirect jump target predictors significantly better than those typically used today.

Given the base system developed for branch prediction, generating the indirect jump predictors was a straightforward one week task. The positive results for these predictors, and the ease of adaptation of the system, make trying this technique for other types of predictors promising.

While we have been encouraged by our results, the genetic programming search often generates rather verbose predictors with portions that contribute little. While this is a necessary characteristic of genetic programming, the opportunity still remains for the automatic reduction of expressions to reduce complexity. This can include eliminating unused memory as well extracting the useful subcomponents of the more accurate predictors and applying them in a more cost-effective manner.

Acknowledgments

The authors would like to thank Simon Steely for his initial encouragement to consider genetic programming for our search. Thanks also to Addison Chen who implemented the initial version of the BP language parser. We also owe thanks to several of our colleagues who reviewed this paper, especially Rebecca Stamm and Geoff Lowney, as well as the anonymous referees. Finally, special thanks go to one of our managers, Tryggve Fossum, for his ever present support and encouragement of this work.

Bibliography

[1] P.J. Angeline and J.B. Pollack, "The Evolutionary Induction of Subroutines", *14th Annual Conference of the Cognitive Science Society*, 1992.

[2] B. Calder, D. Grunwald, J. Emer, "Predictive Associative Cache", *2nd International Symposium on High-Performance Computer Architecture*, Feb 1996.

[3] A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools", *Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems*, Jan 1995.

[4] J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

[5] D.R. Kaeli and P.G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns", *18nd International Symposium on Computer Architecture*, May 1991.

[6] Q. Jacobson, S. Bennett, N. Sharma and J. Smith, "Control Flow Speculation in Multiscalar Processors", 3rd International Symposium on High-Performance Computer Architecture, Feb 1997.

[7] J.R. Koza, *Genetic Programming*, MIT Press, 1992.

[8] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *Computer*, 17(1), Jan 1984.

[9] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction", *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

[10] S. McFarling, "Combining Branch Predictors", WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[11] S. Pan, K. So, and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1992.

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

[13] S. Sechrest, C-C Lee and T. Mudge, "Correlation and Aliasing in Dynamic Branch Predictors", *23nd International Symposium on Computer Architecture*, May 1996.

[14] J.E. Smith, "A Study of Branch Prediction Strategies", *8th International Symposium on Computer Architecture*, June 1981.

[15] SPEC. "Spec Benchmark Specifications", SPEC95 Benchmarks Release, 1995.

[16] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest and J. Emer, "Instruction Fetching: Coping with Code Bloat", *22nd International Symposium on Computer Architecture*, Jun 1995.

[17] T.-Y. Yeh and Y.N. Patt, "Two-level Adaptive Branch Prediction", *24th ACM/IEEE International Symposium on Microarchitecture*, Nov 1991.

[18] T.-Y. Yeh and Y.N. Patt, "Alternative Implementation of Two-level Adaptive Branch Prediction", *19th Annual International Symposium on Computer Architecture*, May 1992.

[19] C. Young, N. Gloy and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction", *22nd International Symposium on Computer Architecture*, Jun 1995.