Genetic Algorithms for Synthesizing Data Value Predictors

Scott R Lenser *Desney S Tan*

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
{slenser, desney}@cs.cmu.edu

ABSTRACT
As processor architectures increase their reliance on speculative parallel execution of sequential programs, the importance of not only what instructions to execute, but also how to resolve data dependences has increased. Data dependences present a major hurdle to the amount of instruction-level parallelism that can be exploited. Data value prediction is a technique that bypasses these dependences by speculating on the outcomes of producer instructions, allowing consumer instructions to execute in parallel. The goal of our project is to explore the application of genetic algorithms (GAs) to the design of value prediction hardware.

KEYWORDS: Genetic programming, genetic algorithm, data value prediction.

1. INTRODUCTION

Modern processors use speculation extensively to avoid stalling the pipeline and thereby to maximize instruction level parallelism. There are two fundamental restrictions that limit the amount of instruction level parallelism that can be exploited in sequential programs: control flow and data flow. Control flow imposes serialization constraints at forks or branches in a program. Data flow imposes serialization constraints on pairs of instructions that are data dependent (ie. one instruction relies on the output of the other for its input). Although a large amount of research has been done in examining control flow limits and methods to overcome restrictions, much less attention has been paid to reducing or eliminating data flow dependences.

In our research, we use a method called genetic programming to explore the design space associated with the creation of novel prediction strategies. Genetic programming, derived from genetic algorithms, is efficient in searching extremely large problem spaces. Its behavior is based on the concepts of natural selection and genetics. In this approach, groups of individuals (each prediction scheme in our case) undergo genetic operations such as recombination (with crossover) and mutation to yield individuals with better evaluation function values (or measures of how well the individual performs the specified task). By applying the genetic operations on large groups of individuals over several generations, we are able to produce individuals that lead to better and better solutions.

The rest of this paper is organized as follows. In section 2, we will briefly discuss related work. In section 3 we present the implementation scheme and methodology used to test the results. In section 4 we present the results along with an analysis and discussion of them. In section 5 we suggest future directions and work and in section 6 we conclude.

2. RELATED WORK

Researchers have tried various schemes for reducing data flow dependences. Initial work was done in predicting recurring values. Harbison's *Tree Machine* uses a value cache to store and look up the results of recurring arithmetic expressions to eliminate redundant computation [10]. Lipasti et al. introduced value locality, a concept related to redundant computation, and utilized a technique they called Load Value Prediction (LVP) that exploits the affinity between load instruction addresses and the values that loads produce. They go on to extend this approach to Last Value Prediction, a technique that applies to all instructions that write to registers [11]. Along similar lines, Tullsen explored a storageless method called register value prediction (RVP) for instructions that typically produced values already in the register file [18].

More recent work has been done in predicting patterns from which values are generated. For example, Gonzalez uses stride predictors to keep track not only of the last value produced by an instruction, but also the difference (or stride) between that value and the previous one [8]. Sazeides extends this method with the two-delta stride predictor that only replaces the stride with a new stride if it has been seen twice in a row (and thus is confident of the change) [13]. Sazeides also explores the use of context predictors that base their prediction on the last several values seen, thus capturing reference patterns that are not reflected in the simple stride prediction scheme. Much research has also been done on the evaluation of combinations, or hybrids, of the predictors described here [2, 12, 19].

3. IMPLEMENTION

We utilized several packages in implementing our algorithms:

- GAlib: a set of genetic algorithm objects implemented in C++ and developed at MIT. The library includes tools for using genetic algorithms to do optimization in any C++ program.
- ATOM: program analysis tool developed and maintained by the Digital Western Research Lab. The tool, which provided an interface to flexible code instrumentation, allowed us to produce traces of the Spec95 benchmarks.

There were several steps in running our genetic algorithm. Although most of steps were based upon a generic genetic algorithm, they had to be customized and tweaked in order for it to function correctly. In this section, we will discuss the decisions made and implementation details in the representation of individuals, the generation of the initial population, the crossover and mutation operations, and the evaluation (or fitness) function with its predict (speculative value generation) and update (feedback with actual value) phases.

Representation of Individuals

We originally considered having individuals organized as directed acyclic graphs since this most closely represents the hardware that a human would generate. We had planned on doing crossover of sub-graphs by selecting sub-graphs with exactly one output and either zero or one input from each individual and swapping the sub-graphs. We feel this would produce the highest quality individuals and is most similar to what a human innovator might try. Unfortunately, many complications are introduced by this crossover operation, including:

- more complex constraint calculation
- complex graph operations to find appropriate sub-graphs
- redundant graph elimination for removing multiple predictions that can result from this scheme
- garbage collection on the crossed over individual to remove unused sections of the graph

Given the time constraints, we had to abandon this idea in favor of tree-based individuals. Each individual is composed of two trees: one that generates a 64-bit value to predict and the other that generates a single-bit value that indicates whether or not to make the prediction. In order to better understand the basic building blocks used, we decomposed existing value predictors (simple stride, 2-delta stride, last value, 2-level context, 2-level stored, etc.) into promitives. The primitives that we found essential can be seen (in no particular order) in Figure 1.

A problem encountered with our initial naïve implementation was that our primitives do not allow for tagged associative lookup tables that might produce better results. In tree based individuals, a single lookup table can only produce a single value. We were, for example, unable to find a way to easily incorporate tables that produce a valid bit and a value into a tree structure. We simulate this structure, however, by encouraging the genetic algorithm to

Construct	Input lengths	Output lengths	Description of Construct
CONST	--	n	Generate constant bit string
ADD	m, n	m	Add two bit strings
SUB	m, n	n	Subtract two bit strings
SATADD	m, n	m	Saturating add of two bit strings
SATSUB	m, n	m	Saturating subtract of two bit strings
JOIN	m, n	n + m	Join two bit strings
CAT	m, n	n	Concatenation
LUT	m, n	m	Look up table
BLUT	m, n	m	Look up table for branch updates
REF	--	m	Reference to output of a lookup table
PC	--	m (<=64)	Lowest m-bits of PC
PQ	--	1	Whether we attempted a prediction or not
PV	--	m (<=64)	Lowest m-bits of predicted value
CV	--	m (<=64)	Lowest m-bits of correct value (available in nodes involved in updates)
BT	--	1	Whether last branch was taken
P	--	1	Whether the previous prediction was correct
PN	--	1	Whether the previous prediction was incorrect
EQ	m, m	1	Equality
XOR	m, m	m	Exclusive or
MUX	n, m, m	m	Multiplexing
MSB	m	1	Most significant bit

Figure 1: Primitives used to construct representations of individual predictors

produce individuals with multiple lookup tables (perhaps one with the tag and the other with the value) indexed by the same expression. We have not provided primitives to support schemes that require LRU state machines due to the added complexity.

An implementation problem with the tree-based representation is that some nodes require more information than other nodes. For instance, CONST nodes need the constant value, REF nodes need the referent ID, LUT nodes need a ID value, but ADD, OR, etc. nodes need no additional information. We handle this by defining a extra data member in the storage for each node which is either NULL or points to additional data needed by that node.

Initial Population

We generate the random trees in a top down fashion using a recursive procedure. We randomly select a primitive to use to produce a value. We use rejection sampling to remove primitives that are illegal for any reason. The following are the reasons that a primitive might be rejected:

- the primitive has children but the maximum tree depth is 1
- a bit width above 64 bits is required but the maximum tree depth is 2 and this primitive will require us to generate a value with more than 64 bits in one node which is impossible
- the primitive cannot produce the required bit width as output (e.g. EQ can not produce a 64 bit value)
- the primitive can only be used in the update phase but this part of the tree will be evaluated during the predict phase (before any feedback has returned)
- the reference (REF) primitive was selected but there was nothing to refer to

We propagate the location in the tree, the bit width of output required, the phase of processing, the maximum depth of the tree, and the bit widths for which references are available through parameters to the random tree generation function. When necessary, we select bit widths for children of a node randomly with biases towards bit widths of 1 and 64 over other bit widths

Operators

There are two main types of operators applied to individuals in the creation of the subsequent generation, crossover and mutation. Crossover takes two individuals from a population and combines parts of them to produce new individuals. Crossover is good at reusing building blocks from individuals to produce new structures. Mutation takes one individual and modifies the individual in some way. Mutation is good at doing local exploration of the solution space around an already known good solution. Mutation is also important to ensure that all possible genetic material is potentially available at all times. Without mutation, it is possible for legal individuals to never be considered simply because the genes that compose them got eliminated in a previous generation. Mutation allows this genetic material to reappear in the population.

The normal way of doing crossover for tree based individuals is to use sub-tree swapping. Sub-tree swapping consists of copying both parents, selecting a sub-tree from each parent, and then swapping the two sub-trees, producing two new and different offspring. Simple sub-tree swapping is not applicable directly in this domain since sub-trees in our individuals may have incompatible environments in which they work. Sub-trees must produce results of a particular bit width in order for the trees that they are a part of to be able to be evaluated. For instance, EQ can only compare values of equal bit width so replacing one of EQ's children with a new sub-tree of different bit width would make it impossible to evaluate the individual. Also, more primitives are available in sub-trees that are only executed during lookup table update phases since more information is available in hardware at this time. Allowing these sub-trees to move from update areas of the tree to normal areas of the tree would result in the individual using information that could never actually be used in hardware.

There are three possible solutions to this problem:

1) allow the generation of individuals that can not be evaluated
2) allow the generation of individuals that can not be evaluated but fix them up before using them
3) prevent the creation of invalid individuals

Solution 1 requires the evaluation function to assign fitness values to individuals that can not be evaluated normally. These fitness values must be higher for individuals that are more legal to ensure that the genetic algorithm makes progress. This approach adds complications in trying to ensure that the algorithm makes progress, selecting a population size (since many individuals will be illegal), and additional run time in producing individuals since most of the individuals produced are useless. Solution 2 can be very effective if it is easy to fix up individuals. However, in many of our cases, it was not clear how this could be done. To work well, the individuals after fix up should be as similar as possible to the individuals before fix up so that the sub-block reuse of the genetic algorithm can be effective. We did not feel this was feasible for many of the individuals we were interested in. Solution 3 limits the possible number of different individuals that can be created by crossover. This is good if all the useful individuals can be created but bad if some useful individuals cannot. We felt a combination approach between solutions 2 and 3 would give the best results. We try to generate correct individuals except in the few cases where the errors introduced are easily remedied. We employ a robust crossover strategy to ensure that as many useful individuals as possible can be generated by the crossover.

Because we want crossovers that make sense, we must ensure that the bit widths used by the individuals at the crossover point are compatible. We must also ensure that the sub-trees are compatible in terms of information available during evaluation. For example, the correct value (CV) can not be used in the prediction itself since it is not yet available. To solve this, we could compute the legal range of values for the output of each sub-tree along with the legal range of values that can be accepted by the node's parent. These ranges could then be compared to find all possible places to crossover that did not involve changing primitives or constant values to produce new bit widths. We describe our simpler approach later.

Additional constraints were imposed on the possible locations of some of the primitives. For instance, the CV (correct value) primitive and the REF primitive can only be used during the update phases, and the BT (branch taken) primitive can only be taken during the branch update phase. This means these primitives can only appear in branches of the tree that are evaluated during these phases (see evaluation below for a description of the evaluation phases). The most general way of ensuring that primitives remain in the correct evaluation phase is to test each possible sub-tree to see if it has any phase specific primitives and crossover appropriately. We simplified this process at the expense of some potentially good crossovers by enforcing the rule that no sub-tree ever moves from an update phase to a predict phase or from a branch update phase to any other phase.

We also ensure that references can only refer to lookup table nodes (LUTs and BLUTs) that are evaluated during the predict phase, or those that the sub-tree with the reference provides the update value for. The most general solution would be to allow reference nodes to refer to update nodes (and branch update reference nodes to branch update nodes) which do not create cycles of references in updating. This would be incredibly complex to implement, however, and unlikely to be particularly useful.

Initially we had planned to have MASKHI and MASKLO primitives that would select some of the most significant or least significant bits off a value. This would have introduced constraints on bit values that are inequalities. Checking to see if two sets of constraints from two individuals were compatible turned out to be too complex since it involves solving sets of linear inequalities. We found that all the schemes proposed by people that we saw only use the MASKLO operation and only use it to extract bits off of the PC or the correct or predicted value. We were able to remove the inequality constraints by adding additional primitives that correspond to different numbers of bits from the PC, CV, or predicted value.
Despite removing the inequality constraints the remaining constraints are still complex. The JOIN primitive makes the constraints complex by adding an addition operation to the equality operation introduced by the other primitives. This results in bit width constraints that are sets of linear equalities. Fortunately, since we are using trees that result in a single cut point, the constraints are often easily determined by local variable values. For most cases, the constraint ends up just enforcing that the bit widths are equal. This is not true for the index to a lookup table since the index can be any width compared to the output. We ignore the special case introduced by the MSB (most significant bit) operator and the additional special case of CAT, which has similar constraints to lookup tables. We handle these two possible cases by having two kinds of crossovers, normal crossovers that preserve bit width and lookup table crossovers which only crossover at lookup table indices. To simplify implementation, crossover was applied only to corresponding trees in individuals (ie. the predict tree from one individual would only crossover with the predict tree from another).

To perform normal crossovers, it is necessary to find the bit widths of all possible sub-trees along with which stage of processing that are processed in (normal [prediction], update, or branch behavior update). We do this in a preprocessing step where we traverse the tree saving this information. The LUT crossover involves finding all the LUT and BLUT nodes in the tree and storing their location and update stage. We do this during the same tree traversal. We randomly select between LUT crossover and normal crossover if both are possible. If neither type of crossover is possible, we simply copy one of the parents.

Additional complications are introduced during crossover by references. One problem with references is that since they refer to another part of the tree, the lookup table they refer to may not exist within the sub-tree that was selected for crossover. This results in references that do not refer to anything. We solve this problem by finding all references that refer to lookup tables not present in the individual. This is done in a two step process. First a tree traversal is done to find all possible referents (i.e. lookup tables to refer to). Then, a second traversal is done to make sure that all reference nodes refer to one of these referents. References that are invalid are fixed by choosing a new referent if possible. The set of referents that can be chosen from includes all referents that are evaluated in the update phase and all referents that are on a path from the reference node to the root of the tree which it is legal for the reference node to refer too. A referent on the path to the root is legal if the value of the referent is guaranteed to be available during the evaluation of the reference node, i.e. the path came from the update node for the lookup table and not the index node.

Another problem with references during crossover is duplication of reference IDs. Consider the following scenario. A highly fit individual is chosen for crossover with two different individuals producing two different

offspring that include the same LUT. The two offspring are themselves highly fit and are selected to crossover with each other. The crossover happens in such a way that the LUT from each offspring is present in this new descendant. Because this descendant has two copies of the LUT, both with the same ID number, any references to the value of this LUT are now ambiguous. To fix this, we rename all the references during crossover after copying from the destination parent and before swapping a sub-tree with the source parent. We do this by performing a tree traversal during which we build up a map from old IDs to new IDs while replacing every ID we find.

Evaluation

Evaluation is the process by which individuals in a given generation are measured to determine whether or not they should be allowed to survive and propagate to the next generation. The evaluation step consists of two phases: the predict phase and the update phase. In the predict phase, the individuals are given information that would normally be available after the 'decode' stage in the execution pipeline, typically the Program Counter (PC) and any prior knowledge or state in the lookup tables (LUTs or BLUTs). From this, the data is run through the individual and a prediction might be made. After this is done, the second phase of evaluation, the update phase, begins. In this phase, the actual values are calculated and updated throughout the individual (we separated the LUT and BLUT because of their different update times) and the cycle continues. Finally, the actual values are compared to the predicted values to get a fitness measure for the individual. Fitter individuals have a higher chance of surviving and propagating into the next generation.

To determine the fitness of an individual, instruction traces were applied to individual predictors and the following counts linearly combined:

- Prediction made; value correct (MC)
- Prediction made; value incorrect (MI)
- Prediction not made; value correct (NC)
- Prediction not made; value incorrect (NI)

The objective of the evaluation function was simply to reward correct predictions and to punish incorrect ones so that the overall fitness of the population would increase. In order to ensure that the predictor would not degrade into one that did not ever make predictions even when it could do so correctly, we weighted the NC case so that it was not as desirable as the MC case. NI is undesirable but does not hurt the system (and is thus ignored), and MI, which makes an incorrect prediction, causes bad results to be produced further in the pipeline and should be avoided if possible. It is important that individuals get credit for producing correct values but failing to decide to predict. This allows the GA to get credit for improving prediction accuracy even though each individual is better of making no predictions than using the predictions it is currently making. Without this, the GA would quickly converge to a solution that makes no predictions and then remains stuck for a long time. The evaluation function we initially used was:

```
Fitness = 2(MC) – (MI) + (NC)
```

The scores are then scaled linearly such that the probability of the selecting the best individual for propagation is a constant factor, which we will call the quality probability multiplier (QPM), times the probability of the selecting the worst individual.

4. METHODOLOGY AND RESULTS

Because this is a prototype system, we chose a simple data set on which to operate and optimize. We picked an arbitrary component of the Spec95 benchmark suite, '099.go'. The 099.go component is an example of the use of Artificial Intelligence in game playing. It is based largely upon the internationally ranked 'Go' playing program called 'The Many Faces of Go' by David Fotland. This is a cpu-bound integer benchmark and a simple base test of simple integer performance. Theoretically, methods used to attain results using this set of data can be applied to a more general set of data in order to produce more general data value predictors.

We instrumented the 099.go benchmark using the ATOM analysis tools [17] in order to produce traces (including program counter, branch behavior, and generated values) of the program. Since the evaluation function is only used as an input to the selection process, we made the tradeoff of small inaccuracies incurred by using only a subset of the benchmark (10,000 instructions) for reduced running time.

We ran the genetic algorithm several times, randomly seeding the initial populations (of 100 – 400 individuals). We manually inspected the individuals and the average weighted scores that were produced at each generation. During the initial tests, we noticed that the generations were not converging as quickly as we had expected. We found simple tweaks that help the convergence immensely.

In the initial runs, individuals would quickly grow very large and complex. This makes the population consume extremely large amounts of memory, causes crossovers to become somewhat ineffective, and makes the individuals difficult to interpret. In order to remedy this, we imposed a penalty for trees that grew too large. We subtracted a thousandth of the node size from the fitness of all individuals to encourage the GA to decrease size by compacting the representation where possible. This solved these size-related problems.

Another interesting observation in the initial runs of the GA was that the best individuals were not influencing the populations as much as we would have liked. We had to

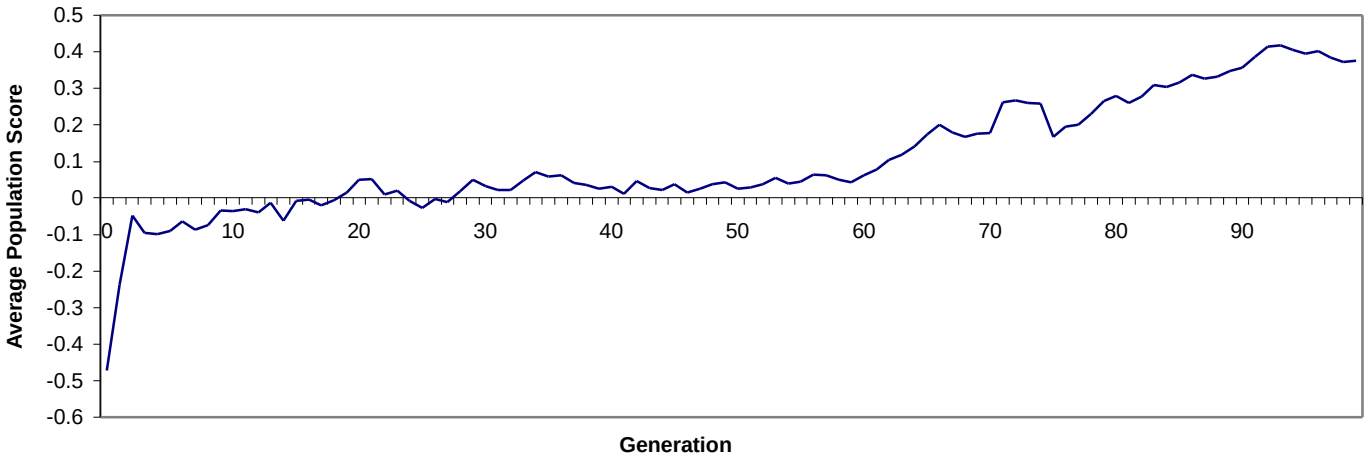

Figure 2: Progress of Genetic Algorithm

increase the QPM from a value of 1.5 to 5. This basically makes it 5 times as likely for the best individual in a population to propagate than the worst. Because of this, better solutions tend to spread more rapidly and have a better chance of influencing the rest of the population in the crossovers.

With these two adjustments, the GA started to converge a lot more quickly. The GA showed steady progress (see Figure 2 for a typical run). In fact, the GA, given no prior knowledge of any predictors, was able to come up with a last value predictor and a variation of a context predictor. It also found a novel confidence predictor, which predicts based on whether or not the last branch was taken.

In Figure 3, we examine the interesting predictors produced by the GA. We simplified each tree to remove redundancy and irrelevant features. In the output produced by the GA, indentation indicates that a node is a child of another node. For a LUT, the first child is the index into the LUT and the second child is the value that is used to update the LUT. The number in "[]"s is the number of bits in the output of a node. This output will become the input to its parent node. The number in "{}"s is the extra argument for the node (table reference number for the LUT, and value for CONST).

One interesting note is that, for both value predictors, the GA decided to use the value of the previous instruction to index into the LUT. We believe that this is more a figment of the implementation of crossover than a useful feature. For the confidence predictor, the GA was able to find looping patterns where backward branches usually indicated good prediction accuracy.

Improvements

Although the GA was able to find interesting results, we recognize certain problems with the current

Produced by GA	Hand-simplified version	Description

LUT[64]{61360}(
  LUT[8]{58877}(
  CONST[1]{0}(),
  PC[8]()),
  CV[64]())
	LUT[64]{61360}(
	  PC[8](),
	  CV[64]())
		Basic last value prediction scheme, where a LUT is indexed by bits from the PC and updated with the correct value (CV)

LUT[64]{100163}(
  LUT[13]{100164}(
  LUT[8]{100165}(
  LUT[8]{100166}(
  CONST[1]{0}(),
  PC[8]()),
  LUT[8]{100167}(
  PC[1](),
  PC[8]())),
  CV[13]()),
  CV[64]())
	LUT[64]{100163}(
	  LUT[13]{100164}(
	  PC[8](),
	  CV[13]()),
	  CV[64]())
		Context predictor, where the context is provided with the previous correct value (CV) of the instruction

BLUT[1]{22632}(
  CONST[1]{0}(),
  BT[1]())
	BLUT[1]{22632}(
	  CONST[1]{0}(),
	  BT[1]())
		Decides to make a prediction if the last branch was taken

Figure 3: Selected predictors produced by Genetic Algorithm

implementation. In the generation of the initial population, we currently randomly generate all individuals. Seeding it with both known good predictors and randomly generated individuals instead could potentially provide crossover with more powerful building blocks to work with. This would give the GA the opportunity to make improvements to existing good solutions. This could be achieved through either small variations or combinations of predictors.

Since individuals are punished for incorrect predictions, the population quickly converges to one that never predicts. Once the value prediction has improved, it is advantageous for individuals to start predicting. Unfortunately, in our design, by the time this occurs, the genetic material required may be lost. We lose genetic material in two ways: material disappearing from the population, and material becoming locked in an inappropriate evaluation phase. In the first case, material is lost when unfit individuals are eliminated from the population.

In the latter case, material can move only from the predict phase to the update phase and not back. After enough generations, material may get trapped in the update phase and cannot be used by the predict phase. This results in the removal of possible solutions from the search space. A solution to this is to check to see if material in the update phase could potentially move back to the predict phase and to allow such crossovers. In order to do this we would have to check all possible sub-trees for material specific to the update phase. Sub-trees without such material can safely be moved back to the predict phase if desired.

We did not implement mutation due to time constraints. We do not believe that this is a large problem for the value predictors because of the relatively small number of generations that we were able to run (<100). It did, however, limit the production of useful confidence predictors because material was lost to the predict phase. The confidence predictors tended to converge to BLUTs indexed by a constant zero, which can easily be made to never predict. In these individuals, the only material available in the predict and update phases is the constant zero. A simple mutation scheme would include replacement of sub-trees with random trees. This would allow lost material to be reintroduced into the population when it is potentially more useful.

5. FUTURE WORK

While we have been encouraged by our results, there is still work to be done in simulating a more complete set of data in order to generalize the predictors that are produced. Also, genetic programming searches tend to produce somewhat verbose predictors that could perhaps be reduced in complexity. Methods such as eliminating unused memory and extracting useful sub-components of the predictors are another line of interesting research. Our evaluation function does raw counts and linearly combines them to form the fitness value. Perhaps this could be extended to account for other intangible factors in determining a good predictor, such as cost-effectiveness or ease of implementation.

6. CONCLUSION

We have explored the use of genetic programming to search the design space in the synthesis of novel and more complete data value predictors. We have implemented a tree-based genetic algorithm with a very specific set of building blocks, which we believe to be minimal and mostly complete. We have also implemented a means to generate a random population and to propagate the generations (with single point crossover and mutation) in order that they may evolve and improve. Simple experiments conducted show that even with the small data sets that were used (due to time constraints), large improvements in the performance of predictors was seen. We hope that further work will be conducted in this area to explore the synthesis of more general predictors.

REFERENCES

[1] Steven J.Beaty. "Genetic Algorithms and Instruction Scheduling," in Proceedings of the 24th Annual International Symposium on Microarchitecture, 1991, pp. 206.

[2] Brad Calder, Glenn Reinman, and Dean M.Tullsen. "Selective Value Prediction," in Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999, pp. 64-74.

[3] Karel Driesen, and Urs Hölzle. "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction," in Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998, pp. 249-258.

[4] Joel Emer, and Nikolas Gloy. "A Language for Describing Predictors and its Application to Automatic Synthesis," in Proceedings of the 24th International Symposium on Computer Architecture, 1997, pp. 304.

[5] Stephanie Forrest. "Genetic algorithms," in ACM Computing Surveys 28, 1 (Mar. 1996), pp. 77-80.

[6] Jason R., and C.Patterson. "Accurate Static Branch Prediction by Value Range Propagation," in Proceedings of the Conference on Programming Language Design and Implementation , 1995, pp. 67-78.

[7] David E.Goldberg. "Genetic and Evolutionary Algorithms Come of Age," in Communications ACM 37, 3 (Mar. 1994), pp. 113-119.

[8] J. Gonzalez and A. Gonzalez. "The Potential of Data Value Speculation to Boost ILP," in 12th Annual International Conference on Supercomputing, 1998.

[9] Marshall Graves, and William Hooper. "A Few New Features for Genetic Algorithms," in Proceedings of the 36th Annual Conference on Southeast Regional Conference, 1998, pp. 228-235.

[10] Samuel P. Harbison. "An Architectural Alternative to Optimizing Compilers," in Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), 1982, pp. 57-65.

[11] Mikko H.Lipasti, and John Paul Shen. "Exceeding the Dataflow Limit via Value Prediction," in Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, 1996, pp. 226.

[12] Bhuslav Rychlik, John Faistl, Bryon Krug, and John Paul Shen. "Efficacy and Performance Impact of Value Prediction," Technical Report CMuART-1998-04.

[13] Yiannakis Sazeides, and James E.Smith. "Modeling Program Predictability," in Proceedings of the 25th Annual International Symposium on Computer Architecture, 1998, pp. 73-84.

[14] James E.Smith. "A Study of Branch Prediction Strategies," in 25 years of the International Symposia on Computer Architecture (selected papers), 1998, pp. 202-215.

[15] James E.Smith. "Retrospective: A Study of Branch Prediction Strategies," in 25 years of the International Symposia on Computer Architecture (selected papers), 1998, pp. 22-23.

[16] Avinash Sodani and Gurindar S. Sohi. "Understanding the Differences Between Value Prediction and Instruction Reuse," in Proceedings of 32st Annual International Symposium on MicroArchitecture, 1998, pp. 205-215.

[17] Amitabh Srivastava and Alan Eustance. "ATOM: A System for Building Customized Program Analysis Tools," Western Research Laboratory Research Report 94/2.

[18] Dean M.Tullsen, and John S.Seng. "Storageless Value Prediction using Prior Register Values," in Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999, pp. 270-279.

[19] Kai Wang, and Manoj Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors," in Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997, pp. 281-290.