Evolving a Better Branch Predictor

Dan Zhang and Ben Lin, *University of Texas at Austin*

***Abstract*—Branch prediction has a strong effect on a system's single-threaded performance. However, branch prediction research has died down over recent years, partly due to diminishing returns at high cost. We propose a new method to improve branch prediction: using directed search heuristics such as genetic algorithms to automatically generate and refine branch predictors. To accomplish this goal, we further propose a new language to describe any branch predictor and can be easily extendable to describe any hardware device. We discuss our framework, which can automatically generate branch predictors when provided with a module library, parse the language, and refine the predictors based on genetic algorithm techniques. In addition, we propose modifications to the base genetic algorithm to improve population diversity, increase quality of results, and reduce stagnation.**

I. Introduction

Modern microprocessors aggressively exploit instruction level parallelism (ILP) through deep pipelines, superscalar issue, and out-of-order (OoO) execution. Such processors require a constant stream of instructions to execute in order to achieve maximum performance. However, branch hazards can disrupt the instruction stream because the correct order of instructions cannot be determined until the given branch is resolved.

Consequently, modern microprocessors rely heavily on *speculative execution*. They aim to predict the direction and target of a branch and speculatively fetch the next instructions before the true order of instructions is known. Once the branch is resolved, if the wrong instructions were executed, then the processor must roll back to the previous state before speculative execution, and re-execute with the correct instructions.

The process of predicting the direction of a branch is called branch prediction. Many branch prediction strategies exist, with the most basic strategies being static, meaning that the branch predictions are made at compile-time and remain static during execution. For example, one can predict that all backward branches will be taken, while all forward branches will not be taken [1], under the rationale that loop branches jump backwards and are usually taken.

More advanced branch prediction strategies are *dynamic* and use run-time history to predict future branches. The pioneering dynamic branch predictor was Yeh and Patt's Two-Level Adaptive Predictor [2], which uses both first-level branch history information and second-level pattern history to make a prediction. The two-level adaptive predictor was very successful and was able to achieve an average of 97% accuracy on benchmarks at the time [2]. Many other variations of the two-level adaptive predictor exist, and are differentiated by whether the branch history is global, per-address, or per-set, and whether the pattern history is global, per-address, or per-set [3].

One branch predictor may work best for a particular execution pattern, while another predictor may be better for another. Hybrid branch predictors exploit this property by combining several predictors into a single predictor through the use of a meta-predictor, which remembers what predictor worked best for a given branch, so that the most suitable predictor is always used [4].

Researchers are still coming up with newer and better designs for branch predictors. Some of the more recent designs, such as Jimenez and Lin's Perceptron predictor [5], have been based on machine learning techniques. The Championship Branch Prediction competition, which has been held in 2004, 2006, and 2011 [6], is a competition in which contestants aim to design the most accurate branch predictor. The competition has been dominated by Andre Seznec's designs, such as [7].

Branch prediction has come a long way and modern branch predictors are highly accurate. For example, [7] was able to achieve a mispredict rate of under 3 mispredicts per 1000 instruction. However, it remains an open question whether further improvements in branch prediction accuracy are possible and, if so, what these new branch predictors might look like. In addressing this problem, difficulties arise from the fact that:

1) Branch Prediction Design is Ad-hoc: Branch prediction design is currently highly ad-hoc and non-systemized. The prevalent approach is to simulate an existing design with a set of benchmarks, find the mispredictions, then modify the design in an ad-hoc fashion to handle the mispredictions. This is problematic, because without a systematic approach, we might only be picking at localized regions of the design space while other promising designs remain unexplored.

2) Current Evaluation Methodologies Ignores Hardware Realities: There are presently no effective ways to evaluate the implementability of a predictor design. The existing evaluation methodology simulates a predictor design in software by feeding it instruction traces from real benchmarks, then measuring the number of times the predictor would have mis-predicted, along with the number of cycles that the fetch unit would have spent on the wrong paths [6].

However, this evaluation strategy does not take into account the implementability of the design. The only implementation metric which can be extracted from a software model of the predictor is the amount of storage the design requires. Other metrics, such as power consumption, design complexity,

delay, etc., are summarily ignored. Indeed, a common criticism of the Championship Branch Prediction competition is that some of the submitted designs aren't even implementable in real hardware. For example, neural network-based designs like [5] require a large adder-tree that would have unacceptably large delays in real hardware.

We wish to address these problems by developing a systematic methodology for designing branch predictors and providing the means to evaluate the cost of implementation of such a design. To accomplish these goals, we design a high-level language capable of expressing any branch predictor. This language can be easily modified to express other hardware devices, such as prefetchers. Then, we present a constrained random generator to automatically generate any number of possible branch predictors of any given complexity. Finally, we apply genetic algorithms, a directed search heuristic that mimics the process of natural evolution to refine these generated models, to create new branch predictors. This entire process, from generation to evaluation and refinement, is completely automated.

II. Prior Work

The concept of designing hardware modules using genetic algorithms is not novel: Emer and Gloy proposed a similar approach in 1997 [8]. In their work, they developed a general language in which most table-based branch predictors can be expressed. Figure 1 shows Emer's model of a branch predictor. Their proposed language modeled all predictors as a table with an *Input* signal which indexes into the table, a *Prediction* output signal representing the value stored at the table index, and a *Feedback* input signal which can be fed back into the table.

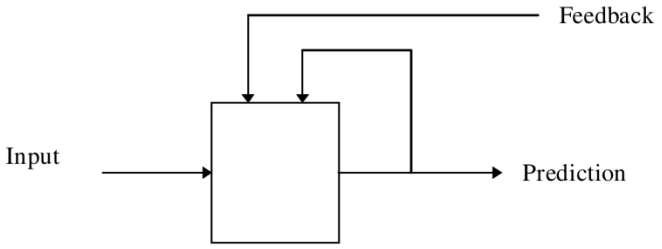

Fig. 1. Emer and Gloy's model of a branch predictor

Any design expressed in Emer and Gloy's language can then be mapped to a tree structure. These trees can be automatically generated and refined using genetic algorithms [9]. A high level overview of the algorithm is as follows:

1. **Generation**: an initial population is randomly generated, forming the initial generation.
2. **Selection**: Individuals are evaluated and a fitness metric is generated for each individual. Those with high fitness are selected for reproduction. Those with low fitness may be discarded.
3. **Reproduction**: Attributes from each parent are combined to form children (crossover), as well as some change for random mutations.

Additional constraints were added to ensure that we produce "legal and usable individuals" [8]. After the genetic operations step, individuals that map to illegal language expressions are corrected, and individuals that require too much memory have their storage structures randomly truncated until their memory requirement falls under a given limit.

Using the methodology outlined above, Gloy and Emer were able to achieve respectable results. They were able to automatically synthesis predictors that were comparable to the state-of-the-art predictors at the time (e.g. GShare) [8].

It's also worth noting that Seznec used simulated annealing (another directed search heuristic) for tuning parameters in his predictor design in [7], but otherwise maintained an ad-hoc design approach.

We improve on Emer's work in several ways. Firstly, our language is more general and can be used to describe any hardware structure, besides just branch predictors. Secondly, our language supports the notion of "introns" or "junk DNA", nodes that are disconnected and do not affect system output but may later do so due to mutations. Thirdly, we develop constrained random techniques to generate better quality predictors for the initial population. Fourthly, we use genetic algorithm improvements, such as elitism, seeding the initial population with known good results, and anti-clustering techniques to improve results.

III. Methodology

In this section we explain our methodology and infrastructure.

A. Language

Our first step was to define a new description language for branch predictors. A main goal was to not limit ourselves to describing only predictors: it should be easy to extend the language to support other hardware devices. We aimed to achieve this by including both high-level, branch predictor-specific language constructs, and low-level generic constructs. For example, aside from having NAND, MUX, and XOR gates, we have tables, branch history registers, path history registers, and 2-bit counter tables in our language library. New library constructs can be easily added without touching the interpreter and generator source code.

Our language consists of a set of generic modules, each of which may contain any number of parameters and inputs. For simplicity, we assume that each module can only produce a single output. All inputs and outputs are assumed to be wires of arbitrary length. Any necessary zero-extension or truncation is done internal to the module.

Below we show a TABLE module. It has two parameters: the number of entries in the table, and the width of each entry. For all parameters, it is possible to specify a possible range of values, using the notation [*upper limit* : *lower limit*]. The predictor generation algorithm will then pick a value from within the range.

Next, we specify the inputs for each module. Each input is either the output of another module, or one of the primitive inputs. In our example table module, the table inputs are the

read index, *write value*, *write index*, and *write enable* signals, respectively.

For each input, you can specify the affinity that the particular input has for a given type of signal. One can then assign a probability that the particular input will be attached to the specified preferred input. We use the keyword *null* to denote that the given input has no particular affinity for any specific value. In this case, all signals have an equal probability of being attached to that input.

```
TABLE#(
[max # of entries : min # of entries]%100,
[max size of entry : min size of entry]%100
)
{
preferred read index % probability,
preferred write value % probability,
preferred write index % probability,
preferred write enable % probability
}
```

Finally, we define the primitive inputs and outputs which our language supports. The primitive inputs are the inputs that the branch predictor will receive from the outside environment.

We support the following primitive inputs:

Input	Description
readValid	true if fetching an instruction
readPC	PC of the instruction being fetched
writeValid	true if retiring an instruction
writePC	PC of instruction being retired
writeTaken	if a branch instruction, whether or not the branch was taken
writeMispredicted	if a branch instruction, whether or not the prediction given at fetch time was false

We only have one primitive output, which is the prediction given. The least significant bit of the prediction signal will decide whether or not the branch is taken:

Output	Description
Prediction	bit 0 is set if branch should be taken

Shown in the following table are the modules we decided to include in the language, and what each parameter and input affinity is set to. Note that our language is extremely extensible, and new modules can very easily be added (or existing modules modified):

Basic logic:

Module	Parameters	Inputs
XOR		{GHR_FETCH%20, GHR_RETIRE%20}
AND		{null%0, null%0}
ADD		{null%0, null%0}
OR		{null%0, null%0}
CONCAT		{null%0, null%0}
MUX		{EQUAL%20, TABLE_2BITCNTR%20, TABLE_2BITCNTR%20}
MSB		{null%0}
NOT		{null%0}
EQUAL		{null%0, null%0}
SHIFT	#([3:0]%100)	{null%0, null%0}
HASH	#([128:8]%100)	{null%0}

Branch prediction specific logic:

Module	Parameters	Inputs
GHR_FETCH	#([128:8]%100)	
GHR_RETIRE	#([128:8]%100)	
PHR_FETCH	#([128:8]%100)	
PHR_RETIRE	#([128:8]%100)	
TABLE_2BITCNTR	#([16384:2048]%100)	{readPC%80, writeTaken%50, writePC%80, writeValid%70}
TABLE	#([16384:2048]%100, [64:0]%100)	{readPC%80, writeTaken%50, writePC%80, writeValid%70}
TABLE_CNTR	#([16384:2048]%100, [8:0]%100)	{readPC%80, writeTaken%50, writePC%80, writeValid%70}

For the branch predictor specific modules, a GHR is a global history register. It records the history of all branches encountered in the program. It is basically a shift register such that a 1 is shifted in if the most recent branch encountered was taken, and a 0 otherwise. Separate Fetch and Retire versions exist for the global branch history at fetch and retire time, respectively. Similarly, the PHR is a path history register; it is a shift register that records the least significant bit of all the branch addresses encountered so far.

The TABLE module is a generic table; TABLE CNTR is a table of saturating n-bit counters; TABLE 2BITCNTR is a table of saturating 2-bit counters, which we included as a special instantiation of TABLE_CNTR

We now present an example instantiation of a GShare predictor, written in our language:

```
GShare in our language
ghr_fetch_0 = GHR_FETCH#(32)
ghr_retire_0 = GHR_RETIRE#(32)
xor_0 = XOR{ghr_fetch_0, readPC}
xor_1 = XOR{ghr_retire_0, writePC}
prediction = TABLE_2BITCNTR#(16384) {xor_0,
writeTaken, xor_1, writeValid};
```

The corresponding dependency graph of GShare is shown below:

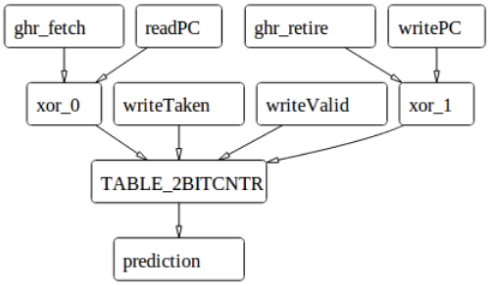

Fig. 2. Dependency graph of GShare

Because of the nature our language, it is totally possible for us to generate a predictor whose dependency graph is disconnected. In Figure 3, we see that the component highlighted is not connected to the predictor output in anyway:

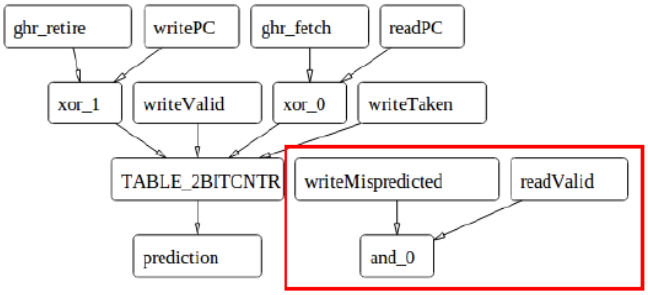

Fig. 3. A predictor which is disconnected (highlighted component is not connected to the predictor out)

While this may appear at first to be a drawback, there are actually advantages to having "useless" components in the predictor. It turns out that for genetic algorithms, it is often usful to have useless genomes called "introns" (or "junk DNA") that, while not directly useful, serve to protect developing individuals from mutations and other genetic operations [8].

This is because in general, most mutations initially decrease the fitness of an individual. Improvements to an individual's fitness usually only occur after a series of mutations. Thus, if there were no intron, then any mutation would immediate impact the fitness of the individual, most likely lowering it, making it far less likely for the individual to survive to the next generation. On the other hand, if the mutations initially occurred in the introns, then the fitness of the individual would not be immediately affected. Consequently, it's more likely for several mutations to accumulate in the introns while the fitness of the individual remains unchanged. A later mutation may finally connect the intron component to the predictor output. If the accumulated mutations led to a better structure, the individual's fitness level will now increase (if it didn't, then of course the individual's fitness will decrease).

B. Fitness Test

To actually evaluate the fitness of generated individuals, we rely on the trace simulation framework provided by the Championship Branch Prediction competition [6]. The Championship Branch Prediction framework provides traces of various workloads, as well as a test harness which will feed the traces into the predictor under test and generates performance statistics. Thus, we needed to implement a compiler which can take a description of a predictor in our language and generate the appropriate C++ code for the simulation framework. To do this, we implemented a C++ class for each module in our language. The module parameters become arguments into the class constructor. In addition, each module class implements a `Invocate()` function, which takes as input the input arguments to the module, and produces a single output.

To model the fact that all our signals and variables can be bitstrings of arbitrary length, we chose to use the `dynamic_bitset` class from the Boost library. This is a class that represents a bitstring of arbitrary width, and the width of the bitstring can be resized at run-time. This gave us the flexibility to combine modules in the most general and arbitrary ways (since we not have to worry about input/output width mismatches, which was a problem that plagued Emer and Gloy [8]). However, we also suffer a clear performance penalty as a result of our approach. Many of the operations (i.e. logic operators like AND, OR, etc.) are far more computationally expensive when performed on these generic bitstrings of arbitrary length, instead of on primitive types like `int`. We also often need to truncate or pad inputs to the appropriate width because we allow for arbitrary widths.

As an example of the slowdown we suffer, the simulation of GShare takes roughly 1 second if the C++ code was written using int, but takes 4 seconds when using dynamic_bitset. Obviously, for more complex predictors, the slowdown becomes worse. A big predictor can take up to 30 seconds to simulate. We profiled the simulation run and fund that most of the times were spent on `dynamic_bitset` method calls. Yet despite the significant slowdown created by using arbitrary length bitstrings, we were able to mitigate this somewhat by pruning dead code generated by disconnected components (discussed below), and by doing several (12) simulations in parallel.

In terms of generating code, we first generate a dependency graph for the predictor like in Figure 2 and 3, and prune out the disconnected components, since they do not affect the prediction. Then, we perform a topological sort on the dependency graph to get the order in which we need to

invocate all the component modules. We then generate the C++ code by calling the `Invocate()` methods on the different module class objects in the order specified by the topological sort. The (simplified) generated C++ code for Gshare is shown below in Figure 4:

```
dynamic_bitset<> readValid;
dynamic_bitset<> writeValid;
dynamic_bitset<> readPC;
dynamic_bitset<> writePC;
dynamic_bitset<> writeTaken;
dynamic_bitset<> writeMispredicted;
...;
GHR_FETCH module_ghr_fetch_0 =
    GHR_FETCH(32);
GHR_RETIRE module_ghr_retire_0 =
    GHR_RETIRE(32);
XOR module_xor_0 = XOR();
XOR module_xor_1 = XOR();
TABLE_2BITCNTR module_prediction =
    TABLE_2BITCNTR(16384);

void PredictorRunACycle() {
    ...;
    dynamic_bitset<> ghr_fetch_0 =
        module_ghr_fetch_0.Invocate();
    dynamic_bitset<> ghr_retire_0 =
        module_ghr_retire_0.Invocate();
    dynamic_bitset<> xor_0
        = module_xor_0.Invocate(
            ghr_fetch_0, readPC);
    dynamic_bitset<> xor_1 =
        module_xor_1.Invocate(
            ghr_retire_0, writePC);
    dynamic_bitset<> prediction =
        module_prediction.Invocate(
            xor_0, writeTaken,
            xor_1, writeValid);

    // Report prediction
    if(readValid[0]) {
      assert(report_pred(
                fe_ptr, false, prediction[0]))
    }
    ...
  }
}
```

Fig. 4. Generated C++ code for GShare

One clear drawback with our approach is that we cannot have loops in the dependency graph. Obviously, if we had a loop in the dependency graph with purely combinational (i.e. logic) modules, then this would be a problem, as this would translate to a combinational loop in real hardware. However, our method also prevents us from having a table module output feed back into its own input chain. This is a false constraint because in real hardware, such loops would not be a problem, because there would be a register in the loop, so it is not a real combinational loop. However, our method currently prevents such constructs, and it actually can prevent us from being able to generate more advanced predictors like [5] and [7]. This is something we definitely need to address in the future.

Also, it's worth noting that although the Championship Branch Prediction framework provides many traces, we chose to use the smallest trace in order to reduce simulation times. Our rationale is that we can use the smallest trace to first identify promising predictor candidates, then run the entire trace on them to get the actual performance statistic.

C. *Predictor Generation*

One key aspect to successfully using genetic algorithms is to generate a diverse and reasonable initial population. Generated predictors need to be sufficiently complex, yet with high enough performance such that useful components can be identified and propagated amongst the population. Thus, the naïve approach of randomly selecting a module, parameters, and inputs does not suffice. Instead, we propose using a constrained random generator to ensure a sufficiently high quality initial population. We focus on improving the quality of parameters and module input selection for this work.

Generating reasonable parameters is simply a matter of selecting between a range of values for each parameter in a configuration file. A better approach would be to define an upper bound, lower bound and mean, but we were not able to implement this due to a lack of time.

Selecting proper inputs is a much more difficult task. A module may have any number of inputs, and the order of inputs is important. For example, a table has a read index, write enable, write data, and write index as its inputs. These inputs are not interchangeable: for example, one would prefer to connect the branch PC value to a table index instead of its write-enable line. To account for this issue, we propose the notion of **input affinity**: each module input has some degree of preference for a specific input wire name. The generator will attempt to choose the specific wire or the output value of a module that used the wire as its input.

D. *Predictor Mating Selection*

After evaluating the code, we rank the predictors based on their performance provided by the framework (cycle penalty per thousand instructions). We use a tournament approach for selecting pairs of branch predictors for mating: first, we select a random pair of predictors. We then choose the best predictor out of those two. Then, we perform this operation again. These two best predictors (the tournament winners) are chosen to be mates. We continue with this process for all members of the population.

E. *Predictor Mating*

To mate two branch predictors, we read the predictor language files and generate dependency trees. These two trees are then merged by randomly selecting a sub-tree from each and swapping the two sub-trees. Note that since the tree is a subset of the predictor description (due to dead code), special care needs to be performed to make sure illegal code is not produced. Conflicting output names are renamed and now-invalid inputs are randomly attached to valid inputs.

F. *Predictor Mutation*

Once the diversity of the initial population is exhausted, we depend on mutation to generate new branch predictors. Thus, we need to support many mutation functions to enable us to perform any branch modification possible. We chose the following set of operations to support:

1. **Input mutation**: a node is randomly chosen and one of its inputs is randomly swapped for a different input.
2. **Parameter mutation**: a node is randomly chosen and

one of its parameters is regenerated using the constrained random technique described in Section 3b.

3. **Node addition**: a new module is randomly generated and its inputs and parameters are randomly selected using the constrained random technique described in Section 3b.
4. **Node deletion**: a random module is deleted. Any resulting invalid inputs are randomly swapped for valid inputs.

G. *Population Selection*

After the children are generated, we re-rank all predictors (parents + children) and choose the predictors with the highest fitness for the next generation. This is an extreme form of **elitism** and provides the fastest method of convergence but may lead to a sub-optimal solution. We leave testing various methods of selection as future work.

IV. Methodology Refinement

After running initial experiments, we identified numerous problems with the basic genetic algorithm approach and introduced heuristics to address these issues.

A. *Local Maximum Stagnation*

After 5-10 generations, the genetic algorithm converges on a relatively poor local maximum and does not find a better alternative. To combat this, we implemented **stagnation detection**: after a certain number of generations in which the best predictor does not change, we start to increase the mutation rate. For our tests, the mutation rate starts out at 1 mutation per generated child, and then is doubled after every generation up to a maximum of 32. We were not able to improve performance much even after using this approach.

B. *Poor Initial Population*

Due to a lack of time, we were not able to effectively tune our constrained random predictor generator. Thus, we decided to introduce **population seeding**: we augmented our initial population pool with known good predictors, such as gshare and some hybrid predictors. This pool is separate from the rest of the population and is only available for the purposes of mating. This did improve convergence significantly and produced a better local maximum.

C. *Poor Population Diversity*

After a few generations, the population diversity started to suffer greatly as it converged on a solution. Poor diversity prevents the genetic algorithm from operating efficiently and can cause it to converge on a poor local maxima. To combat this issue, we introduced **cluster detection**: when "clusters" of predictors with the exact same performance (and thus the same internal structure) are detected, their fitness score is multiplied by a function of the number of members in the cluster. However, we would still like to keep a few of the members. Therefore, for predictors $\{P_1, \ldots P_n\}$, we compute the modified fitness of the predictors as $\{ P_1*1, \ldots, P_n*n\}$. In practice, this improved the population diversity significantly.

V. Evaluation

Due to the sheer magnitude of the project, we were unable to spend enough time tuning the overall process. Performance results were poor, although the overall technique seems relatively promising. Overall, we were not able to evolve anything better than a 2-bit counter table for unknown reasons. Further work is necessary to identify potential issues related to our complicated generation and mating algorithm. We did all of our testing on our lab machine, an overclocked 6-core machine running at 4.0GHz with 24GB DDR3, as well as the LRC machines. Due to our performance optimizations (launching parallel compilations and simulation), runtimes of 30 seconds for each predictor simulation were common. We only had 6 cores and 12 threads available, so each generation would complete in about POPULATION_SIZE*0.5/12 minutes. It is possible to farm the job out to multiple computers to further improve runtimes.

We evaluated the genetic algorithm with and without cluster detection. The runs are with a population size of 24, 100 generations, branch predictor complexity of up to 20 nodes, and mutation rate of 1 mutation per child. As Figures 5 and 6 show, after tens of generations, the run without cluster detection converges completely on a single solution. However, with cluster detection, the population stays diverse (and with a similar local min), improving its changes to find a better solution.

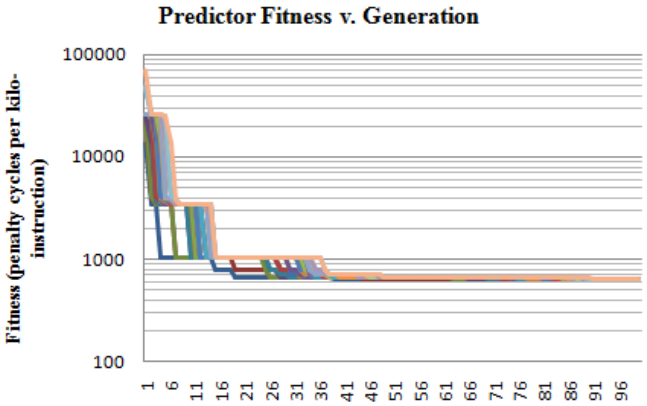

Fig. 5. Predictor fitness vs. generation without cluster detection

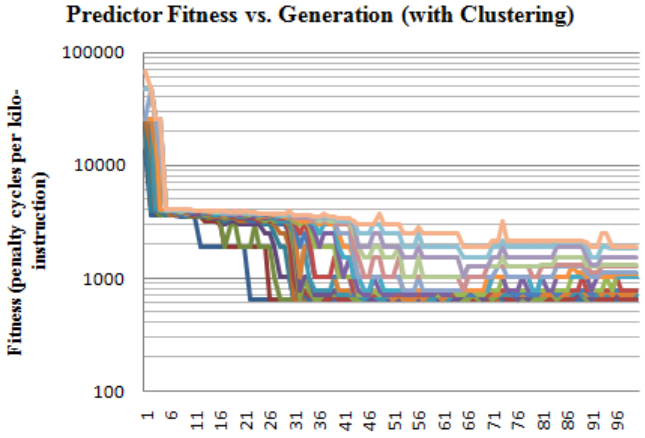

Fig. 6. Predictor fitness vs. generation with cluster detection

VI. Conclusion and Future Work

In this work, we propose a new language to describe branch predictors that can be easily extended to any type of hardware devices. We propose a method of automatically generating, evaluating, and refining predictors and provide a proof of concept design. Although we were not able to improve on existing predictor designs, we have demonstrated that our automated approach can improve the performance of the randomly generated predictors, albeit currently not well. Genetic algorithms need a lot of tuning to produce good results, and we simply were not able to do this due to a lack of time. However, due to the modularized and parameterized way in which our code was written, doing so is not difficult.

Future work includes allowing for table outputs to feed back into table inputs, generating HDL code from the branch predictor description language, improving the constrained random generator, tweaking the library modules and their input affinity weights, and improving the predictor mating and mutation functions.

VII. References

[1] J. E. Smith, "A study of branch prediction strategies," in 25 years of the international symposia on Computer architecture (selected papers), ser. ISCA '98. New York, NY, USA: ACM, 1998, pp. 202–215. [Online]. Available: http://doi.acm.org/10.1145/285930.285980

[2] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in Proceedings of the 24th annual international symposium on Microarchitecture, ser. MICRO 24. New York, NY, USA: ACM, 1991, pp. 51–61. [Online]. Available: http://doi.acm.org/10.1145/123465.123475

[3] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in Proceedings of the 20th annual international symposium on computer architecture, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 257–266. [Online]. Available: http://doi.acm.org/10.1145/165123.165161

[4] B. Predictors and S. McFarling, "Combining branch predictors," 1993.

[5] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," High-Performance Computer Architecture, International Symposium on Computer Architecture, vol. 0, p. 0197, 2001.

[6] "Championship branch prediction," 2011. [Online]. Available: http://www.jilp.org/jwac-2/

[7] A. Seznec, "Analysis of the o-geometric history length branch predictor," International Symposium on Computer Architecture, vol. 0, pp. 394–405, 2005.

[8] J. Emer and N. Gloy, "A language for describing predictors and its application to automatic synthesis," in Proceedings of the 24th annual international symposium on Computer architecture, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 304–314. [Online]. Available: http://doi.acm.org/10.1145/264107.264212

[9] J. R. Koza, "Genetic programming," 1992.