Fault-Tolerant Hardware Through N-Version Genetic Programming

KOSUKE IMAMURA    JAMES A. FOSTER

Initiative for Bioinformatics and Evolutionary Studies (IBEST)
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
{kosuke, foster}@cs.uidaho.edu
(208) 885-7062

Abstract

Classical techniques for fault tolerant hardware use redundant components, often with fault detection through voting schemes and sometimes with fault correction strategies. One can also build fault tolerant software systems by implementing N-versions of syntactically different but semantically identical modules, together with a voting scheme. Both approaches assume that one can design multiple versions of a computational object that have independent, uncorrelated faults. This is often difficult, since engineers share the same education, design tools, and assumptions about design methodology. Our approach is to use an island model Genetic Programming (GP) to build redundant computational objects. We have applied this approach to software in earlier work, and in this paper apply it to evolvable hardware. We produce fault tolerant square root circuits, with 5-way redundancy in a voting-oriented system. We compare our fault tolerant circuit to a single evolved circuit without explicit fault tolerance, and show that both types of circuits have the same probability of producing erroneous results, but that the variance in error is significantly smaller with the N-version system. This means that system failures are significantly less frequent with the N-version system than with a single circuit system, so the N-version approach to evolvable hardware design can produce significantly more reliable hardware.

1. INTRODUCTION

Hecht cites Dijkstra, “program testing can be used to show the presence of bugs but never to show their absence” [Hecht and Hecht]. Exhaustive testing may prove the correctness of programs built with sample training based methods such as genetic programming and artificial neural networks, but this is a nearly impossible task in real applications because of the vast input space. Moreover, without exhaustive testing, inferred programs are likely to be incorrect [Imamura et al.]. We may evolve a Mars landing control system with all the data we have. However, the system behavior may prove unpredictable when it starts encountering unseen data. A risk minimizing measure is of critical importance under such circumstances.

The goal of N-version programming is to adapt techniques used in hardware redundancy with majority voting to build programs that tolerate software design faults [Hilford et al.]. These techniques are problematic with software, since programming teams often share assumptions, tools, and design methodologies—even when efforts are made not to do so [Hecht and Hecht]. We have addressed this problem by using island model genetic programming (GP), where individuals programs evolve on isolated islands. This produces sufficient speciation pressure to evolve multiple software versions with relatively
uncorrelated faults, from which a fault tolerant system can be built [Imamura and Foster, 2001]. In this paper, we apply the same GP approach in order to evolve multiple speciated square root circuits. Each circuit participates in a voting scheme, so that its parts determine the final output of the overall system. We refer to this approach as N-Version Genetic Programming (NVGP).

In order to sustain continuing operation of most systems, especially autonomous or safety critical ones, it is necessary to provide means to detect and recover from faults. With evolutionary techniques, there are no design specifications and therefore no path testing. Consequently, conventional software testing or design techniques are difficult to apply to evolved systems. The natural evolutionary strategy in a voting-oriented system is to subject modules whose votes are in the minority, and which are therefore presumed to be faulty, to further evolution. We applied this approach to software systems [Imamura and Foster, 2001] and showed that it is an effective way to reduce system failures. We have not yet applied any fault recovery scheme to our evolved fault tolerant hardware systems.

Our test case is to evolve real-valued arithmetic circuits for computing square roots, where the targeted implementation is FPGAs. This problem was taken from Miller and Thompson [Miller and Thompson], and we used their representations and fitness functions.

Statistics from our experiment suggests that NVGP may be a viable option for increased reliability of evolvable digital circuits. It produces systems with redundant circuits that have error rates comparable to single-module systems, but with significantly less error variability, and therefore with significantly higher reliability.

2. BACKGROUND AND PREVIOUS WORKS

A fault is an undesirable behavior in a system, such as incorrect output from a program module or a dropped bit in a communication line. A failure is when a system behaves incorrectly, for example by returning an incorrect value to the user or by forwarding an erroneous packet. Fault tolerant systems detect and correct faults before they become failures. Fault tolerant software by N-version programming is one of the techniques to cope with software faults [Imamura and Foster, 2001]. N-version programming (NVP) is defined as the independent generation of N≥2 functionally equivalent programs (versions) from the same initial specifications [Avizienis and Kelly].

A fundamental assumption of the NVP is that independent programming effort will reduce the probability that similar errors will occur in two or more versions [Avizienis and Kelly]. Independent programming is believed to promote diversity in algorithms, data structures, programming languages, platforms, and error recoveries. Different approaches to attaining program design diversity are discussed in [Hillford et al.], including NVP. Independent faults can be masked by votes. For simplicity, if we assume the probability of fault of each version is uniformly p and that the faults are independent, then the probability of the system failure, f, is

\[ f = \sum_{k=m}^{n} \binom{n}{k} (1-p)^{n-k} p^k, \]

where n is the total number of versions, and m is the minimum number of faulty votes to cause system wide failure.

Typical N-version software is constructed by isolating N individuals or N groups of programmers. Difficulties of attaining diversity among the versions, which is prerequisite for an independent fault, by human effort may be attributed to the fact that we have similar education and training backgrounds, and therefore we tend to use the similar programming techniques and algorithms to solve common problems.

Knight and Leveson applied a probabilistic metric to measure the assumed independence of NVP [Knight and Leveson]. Interestingly, the theoretical system fault rate (f in the above case) was not observed in their experiment and they rejected the hypothesis of the assumed independence of fault by independently developed programs. However, the rejection of the independent fault hypothesis does not mean we should not try N-version. Hatton followed up the Knight and Leveson’s study, and his finding was that N-version is more reliable and cost effective than one good version [Hatton], even with non-independent faults. He also observed non-independent faults, and his 3-version system (with fault probability is 0.004 each) increased the reliability 45 times, while the complete independent faults would have increased it 833.6 times. This result suggests that N-version can substantially increase the reliability even with non-independent faults.

Another important aspect of NVP is ability to detect faults. Even with mildly non-independent faults, if a large majority of modules agree in their output, the minority outputs are likely to be incorrect. This
probability increases with the number of voters. Naturally, if the voting is highly dependent, then there is a chance that a majority of voters will agree to produce erroneous output.

NVGP is identical to a conventional NVP in concept, but without design/implementation issues of human-effort. NVGP automates the creation of redundant modules by using speciation with genetic programming.

A similar study to our NVGP is found in Hashem’s Linearly Optimal Combination of ANN. An artificial neural network is another example of a black box approach. Hashem reports superior performance of a linearly combined neural network (LOCANN) [Hashem 1995 and Hashem 1997]. In his approach, redundant modules (trained ANNs) are assigned optimal weights for computing a weighted average of the module outputs. NVGP is dissimilar to LOCANN in that NVGP’s weights are all equal because we cannot assume that some circuits are better than others, while LOCANN estimates the best weight vector from the given sample set.

Since isolating groups of programmers is very much like an isolated island model, and since it is the only method known to us, we developed an NVGP using the isolated island model. In the next section, we describe our experiment.

3. EXPERIMENT

We duplicated the experiment by [Miller and Thomson], where FPGA is configured by genetic algorithm to approximate a real-valued function. We used a GA library package by [libGA] and simulated our NVGP on the Initiative for Bioinformatics and Evolutionary Studies (IBEST) Beowulf cluster supercomputer at the University of Idaho. The GA was configured as follows: Population 2000, generational, 200 iterations or convergence (whichever occurs first), crossover rate 0.9, and mutation rate 0.6. Smaller population and lower mutation rate resulted in premature convergence and undesirable performance. This extremely high mutation rate suggests that the problem is difficult for GA or chromosome representation may not be appropriate. However, this analysis is beyond the scope of this paper. The function to approximate is a square root function in the range [0.00, 1.01]. The input and output are 8 bit data and mapped to real by (binary number * 0.004). We used 21 equally spaced input and corresponding output pairs out of the total of 128 pairs for the training set. For example, for input 00000010 (decimal 2, real-value 2*0.004=0.008, the correct output is 00010110 (decimal 22, real-value 22*0.004=0.088).

Our target system was the same FPGA (from the Xilinx 6200 family) as in Miller and Thompson’s work [Miller and Thompson], and as in their work we simulated the FPGA behavior, rather than implementing it in physically. Our evolved configurations are correct for this family of device, and our routing assumptions do not violate any device specific assumptions. Consequently, our evolved circuits should run without modification on a Xilinx 6216, though we have not actually run them on this device. Our circuits take identical input data from East and South pins, and send outputs to both North and West pins. The better output is considered to be the overall circuit output. See Figure 1.

Figure 1. FPGA/data interface.

Our NVGP implementation consists of 5 modules (so N=5). Each module is trained with the same training set. After the training, we evaluate the system on all 128 inputs. For each input, the 5 outputs from each module are averaged to produce the final output. For performance comparison, we used the best single module as a standalone system. Since the single module system was designed with the same GP as
the N-version one, the resulting system behaviors are solely a result of having an N-version voting system (and not of different design methodologies). We ran both systems 20 times and recorded average error margins and their standard deviations. A large standard deviation on error margin leads to unreliable system behavior, while errors on average may be easily fixed by calibration if the standard deviation is small. Table 1 is a summary of two-sample t test by Satterthwaite Approximation.

Table 1. Average Error and Standard Deviation for 5-module square route system and 1-module system over 20 iterations.

<table>
<thead>
<tr>
<th>System</th>
<th>Average Error</th>
<th>Average Std Dev of Error</th>
<th>Std Dev of Error</th>
<th>Std Dev of Std Dev of Error of Each Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVGP System (5 module version)</td>
<td>-0.009139</td>
<td>0.060348</td>
<td>0.008083</td>
<td>0.011023</td>
</tr>
<tr>
<td>Single Module System</td>
<td>-0.009301</td>
<td>0.083665</td>
<td>0.016717</td>
<td>0.019645</td>
</tr>
</tbody>
</table>

λ(average error) = 0.098705 < 3.291 (p= 0.9995) df ≈ 183
NVGP does not have significantly a smaller average error margin.

λ(average std_dev) = -11.710852 < -3.291 (p= 0.9995) df ≈ 199
NVGP has significantly a smaller standard deviation on error margin.

where,

\[ \lambda = \frac{(r_1 - r_2)}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \]

\[ df \approx \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{(s_1^2/n_1)^2(n_1-1) + (s_2^2/n_2)^2(n_2-1)} \]

average error = \(\sum x_i/20\),
average std dev of error = \(\sum \text{std dev}(x_i) /20\),
std dev of error = \(\text{std dev}(x_i)\),
\(\text{std dev}(\text{std dev}(x_i))\) of each run.

Table 1 indicates that NVGP and a single version system have the same average error margin. However, NVGP has significantly smaller average standard deviation of error margin at confidence level = 0.0005 (one sided). For visual confirmation, the Figure 2 plots all the errors. Notice the NVGP plot band is narrower than the single version plot band. This is an important issue for the performance guarantee error-threshold.

**Figure 2.** Error Plot. Vertical axis is deviation from the correct output (the error margin). Horizontal axis is the input value.

A. Single module system

B. NVGP (5 module version)

Suppose that a system failure occurs when the output of the system deviates from the target value by a pre-specified error threshold. Then the smaller standard deviation of errors with NVGP implies a smaller
failure rate for the overall system, since the system output is the average output from each of the modules. Of course, the system is not aware of its own failure rate, since the correct output value is an unknown.

Table 2 shows the reliability of NVGP and a single version system in terms of number of errors beyond the error threshold. If the error threshold is 0.18, a single version system would produce 4.67 times more errors than NVGP from the table 2 (A) and the standard deviation on number of error is 2.83 times larger than NVGP from the table 2 (B). This is statistical evidence that a single best version is far more unpredictable than NVGP. Relatively large errors on input range near 0 and smaller errors toward 1.0 is explained in [Imamura et al.]

Table 2. Reliability Comparison

<table>
<thead>
<tr>
<th>A. Error Threshold (allowable deviation from correct output) and Number of system failures.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Threshold</td>
</tr>
<tr>
<td>Avg. failures (NVGP)</td>
</tr>
<tr>
<td>Avg. failures (Single module version)</td>
</tr>
<tr>
<td>Single/NVGP ratio</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B. Error threshold and Standard Deviations on Number of errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Threshold</td>
</tr>
<tr>
<td>std deviation of # of err (NVGP)</td>
</tr>
<tr>
<td>std deviation of # of err (Single module version)</td>
</tr>
<tr>
<td>Single/NVGP ratio</td>
</tr>
</tbody>
</table>

4. CONCLUSION AND FUTURE RESEARCH

With statistical significance, the NVGP (5-version) by an isolated island model is far more predictable than a single best version. Whether or not this improvement is important depends on how critical an application is. For safety critical applications, such as aircraft control modules or nuclear power plant cooling system regulators, any improvement is significant. Reliability increases when faults are independent.

Currently, we are exploring alternative speciation strategies for evolvable systems. With speciation metrics, we could discourage similar hardware modules from participating in voting, resulting in increased ability of fault masking. Textual similarity in genomes, semantic similarity in behaviors, and runtime profile are being considered as speciation metric components.

Another area of our research is to maintain an optimal voter pool in order to respond in real-time, essentially adapting our software failure recovery strategies to hardware systems. This will result in self-repairing evolvable hardware, which will remove minority voters (or whose output is significantly different from the system output) from the voter pool for retraining, and a voter that completed retraining will be brought back into the pool.

Acknowledgement:

This research was funded in part by NSF EPS0080935 and NIH F33 GM20122-01. The authors are grateful to Dr. Axel Krings, Dr. Robert Heckendorn, and Dr. Terence Soule of the Initiative for Bioinformatics and Evolutionary STudies (IBEST) at the University of Idaho for helpful discussions.

References:


LibGA Package
http://euler.mcs.utulsa.edu/~corcoran/LIBGA101.zip
ftp.aicnrl.navy.mil:/pub/galist/src/ga/libga100.tar.Z