GENETIC LEARNING BASED FAULT TOLERANT COVER FOR DIGITAL SYSTEMS

A.P. Shanthi, Balaji Vijayan, Manivel Rajendran, Senthilkumar Veluswami, Ranjani Parthasarathi

a.p.shanthi@cs.annauniv.edu

School of Computer Science and Engineering,
Anna University,
Chennai-600 025,
India.
GENETIC LEARNING BASED FAULT TOLERANT COVER FOR
DIGITAL SYSTEMS

ABSTRACT

This work proposes a genetic algorithm based technique to learn the structural description of the circuit under study, in order to give it an on line fault tolerant cover. As the circuit operates, its input / output sequences are used to learn the structural configuration of the hardware. Once the structure is evolved, the algorithm redistributes the available redundancy and generates multiple versions of the structure to provide 100% single component fault coverage. The input /output pairs are constantly monitored for possible faults. When a fault is detected, the multiple versions already evolved are made use of to provide fault correction. For a two-dimensional array configuration, it has been found that the number of versions to be evolved ranges from just two to a maximum of the dimensionality of the array. With the number of versions getting reduced because of the redistribution that is done at the learning stage, and the fact that the versions are evolved off-line, the overhead in terms of the system downtime and the number of reconfigurations to be done are minimised. The efficacy of the proposed techniques has been studied using simulations. Hardware implementation has been carried out as a proof of concept. It is found that the proposed fault tolerant cover detects the fault and provides 100% fault correction for single component faults.

1. INTRODUCTION

As hardware systems become more and more complex, it becomes very difficult to provide comprehensive fault testing to determine the validity of the system. Hence, faults can remain in a system and manifest themselves as errors later on. Furthermore, external sources of interference or component failures may also cause a system to fail. Hence, the ability of the system to function in the presence of faults is a continually challenging area of research.

Fault tolerance is typically achieved by continuously monitoring the system in operation, looking for faults and handling them appropriately when they arise. One popular approach for this purpose is the N-version approach, where N versions are maintained and a voting mechanism is used to ascertain the consensus opinion [1]. But the extra equipment and design time required are significant in terms of cost, space, weight and power. Also it is not capable of adapting to faults, particularly in the case of ASIC based designs. However, Field Programmable Gate Arrays (FPGAs) with their abundance of programmable logic and routing resources, provide a viable alternative to ASICs [2]. It is possible to optimise an FPGA based system from various perspectives, including performance, cost and dependability [3,4,5]. For FPGA based reconfigurable systems used in dependable applications, they can be reconfigured to operate without using the defective element and utilising the available redundancy. Thus the basic replaceable unit is a fine- grained programmable logic block or a routing resource, instead of a chip or a board as in conventional fault tolerant systems, providing a more
cost effective solution to tolerance of faults in the system. FPGAs have also given rise to an amazingly different and unconventional way of designing hardware-the evolution-based design, viz. Evolvable Hardware (EHW) [6]. EHW gives an opportunity to provide optimal performance by tailoring the architecture to the characteristics of the given problem. Several researchers have emphasised the inherent fault tolerance property of EHW [7] [8]. Thomson [9] has investigated the effects of fault injection within the fitness measure of the evolutionary process and proved that it is possible to tolerate single faults. Tyrrell [10] has brought in an analogy between the fault tolerance provided in hardware systems and the human immune system and has demonstrated the error detecting capabilities of a hardware immune system [11].

The work proposed in this paper exploits the redundancy provided by evolutionary techniques and provides a complete system capable of detecting faults as well as correcting them. The proposed fault tolerant cover learns the structure of the circuit without any prior knowledge, constantly monitors it for faults and corrects the faults.

The GA based learning process, which is a crucial part of the cover, deduces an internal representation of the system from samples of its functioning given by several input / output sequences. Once the circuit is learnt, 100% single component fault coverage is provided by redistributing the resources and evolving multiple versions. Fault detection is done by constantly monitoring the input /output combinations. When an error occurs, the already evolved versions are considered and an alternate version that bypasses the possible faulty component is downloaded, thus correcting the fault. The significance of this approach is that it provides a simple, general-purpose cover for any type of circuit. Also, with the number of versions ranging from just two to the dimensionality of the structure, the system downtime and the number of reconfigurations are minimised.

The paper is organised as follows. Section 2 discusses the fundamentals of genetic algorithms and their application to genetic learning of hardware systems and fault tolerance. The proposed fault tolerant cover with its various phases is presented in section 3. Section 4 gives an introduction to JBits, which has been used for the hardware implementation of the circuit under consideration. The experimental results are presented and discussed in section 5. Section 6 summarises the salient features of this work and proposes future directions of research.

2. GENETIC ALGORITHM BASED LEARNING AND FAULT TOLERANCE

The metaphor underlying genetic algorithms is that of natural evolution. Genetic algorithms have been applied to a wide range of problems, from optimising the design of mechanical objects to evolving computer programs [12] and EHW. The application of genetic algorithms to design digital circuits has been attempted at the gate level and functional level [13,14]. Recently they have been applied to evolving fully synthesized, placed and routed reconfigurable circuits [15]. Attempts have also been made to use GA to learn the Finite State Machine (FSM) of a circuit [16]. The general procedure is as follows. The model of the circuit to be learnt / designed is represented as a chromosome.
The standard genetic operators such as initialisation, recombination and selection are carried out on the chromosomes, till the representation that satisfies all the input / output sequences evolves. The suitability of the evolved configuration is decided by means of a fitness function. The evolved solutions may be evaluated using software simulation models, as in the case of extrinsic evolution [17], or alternatively evaluated entirely in hardware, as in intrinsic evolution [18]. In both these cases, the evolution process itself is carried out in software. It is also possible to have the evolution process itself done on hardware to speed up the process, which is termed as complete evolution [19].

Genetic algorithm based learning of structural representation has found application in different fields such as logic learning, system design, test and verification [20,21,22]. The application of this approach to provide fault tolerance to hardware systems looks promising because of the inherent fault tolerance supported by evolutionary strategies [23,24].

3. GENETIC LEARNING BASED FAULT TOLERANT COVER

The learning approach used here makes use of a genetic algorithm to learn the internal structure of the circuit under study, without any prior knowledge of it. The learning algorithm continuously takes in the input / output combinations as they occur, and evolves a suitable hardware configuration. The learning is done on-line, autonomously, with the learnt circuit adapting to the changing environment. The learnt structure is then modified and multiple working configurations are evolved to provide fault correction. Also a constant monitoring of the circuit is done to detect faults. Thus the implemented GA based fault tolerant cover learns the hardware structure, checks it for errors and corrects them. It consists of the following phases.

- A learning phase that learns the hardware structure on-line and autonomously.
- A monitoring phase that constantly checks the hardware structure for possible faults providing fault detection and
- A correction phase that rectifies the error with minimal system downtime and reconfigurations.

The fault tolerant cover alternates between the learning phase, monitoring phase and the correction phase depending upon the input / output sequences encountered.

**Learning Phase:** This is the initial phase and also crucial to the success of this approach. As new inputs are presented to the hardware and outputs are generated, the GA takes in these input / output combinations and evolves a structural layout satisfying these combinations. A two-dimensional array structure is used for evolution. This process continues for a period of time in order to enable the system to learn the hardware. The learning process is not a one-time process. It may be turned on even while the system is monitoring for faults, if a valid new input arrives. In such a case, the system incorporates this new input with the already learnt sequences, and evolves a structure satisfying all the input / output sequences. The circuit evolved is documented with the used resources and the redundant resources information. This is called the Primary Replacement Version
(PRV). The learning process does not stop with this. Merely providing redundancy is not enough. The learning algorithm must be modified to utilize the available redundancy in a better and effective manner to achieve maximum fault tolerance. Hence, the redundancy available in the evolved circuit is distributed in such a way that maximum fault coverage is provided. This redistribution of useful redundancy ensures that there is at least one redundant component in each column of the evolved circuit. When a column has more than one used component, it is not possible to provide fault correction for all the used components with this single version alone. Hence, depending on the used resources, multiple alternate versions of the PRV called Secondary Replacement Versions (SRVs) that handle faults in each of these used components are evolved, thus providing 100% single component fault correction. By appropriately distributing the used components in the two dimensional array considered for evolution, and thus providing useful redundancy in all the columns of the array, the number of versions needed for 100% single component fault coverage gets reduced and ranges from a bare minimum of two to a maximum of the array dimensionality. Hence the number of reconfigurations needed to correct the errors gets minimised. This process of generating multiple versions depending on the used resources is repeated for a number of PRVs of the hardware under study. This is done to obtain the optimal of the evolved PRVs that provides the maximal fault coverage, to start with.

Monitoring Phase: Once the system has learnt the hardware, the algorithm enters the monitoring phase checking for discrepancies between the learnt input / output combinations and the ones that occur now. A discrepancy in the input / output combination is regarded a faulty state. Hence, fault detection is a simple process of just identifying any change in the previously encountered input /output pair. When a new input that has not been encountered previously arrives, this is taken as a new input to be learnt and the learning process is invoked. The learning process evolves the PRV incorporating the new input and also generates the multiple secondary versions for correction. This entire operation is done off-line without affecting the system operation. After the learning process is over, control is again transferred to the monitoring phase.

Correction Phase: Once a fault has been detected in the monitoring phase, the algorithm enters the correction phase. Out of the evolved PRVs, the one with the minimum amount of used resources and hence the maximum amount of useful redundancy is downloaded. As this version has the maximum fault coverage, this version itself may provide the fault correction. The versions evolved may not be structurally equivalent to the actual circuit implemented in hardware, though they are functionally equivalent. But the optimal of the versions evolved increases the chances of hitting upon a fault free version. If the version downloaded does not rectify the defect, the already evolved secondary replacement versions corresponding to this PRV are tried out. Since the discrepancy in the input/output combination just indicates that there is an error, and there is no knowledge about the actual faulty component, the multiple SRVs which handle each one of the used components are to be tried out one after the other. Thus correction is carried out by downloading each of these possible solutions and checking the validity of the circuit. For single component faults, 100% fault coverage is provided with two to a maximum of n versions, where n is the dimensionality.
The experimental setup consisting of a two dimensional array of gates used to simulate this algorithm and the results obtained are discussed in section 5. The implementation of the fault tolerant cover for a circuit under test has been done on a Virtex XCV300 FPGA [2] using JBits, a Xilinx supplied tool. An overview of JBits and its usage for the hardware implementation is provided in the following section.

4. AN OVERVIEW OF THE JBITS TOOL SUITE

JBits is a set of Java classes that provide an Application Program Interface (API) into the Xilinx Virtex FPGA family bit stream [25]. This interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams read back from actual hardware. This permits all configurable resources like look-up tables, routing and the flip-flops in the FPGA to be individually configured under software control. It provides the capability of designing and dynamically modifying circuits in Xilinx Virtex series FPGA devices. The programming model used by JBits is a two dimensional array of Configurable Logic Blocks (CLBs). Each CLB is referenced by a row and column index, and all configurable resources in the selected CLB may be set or probed.

The circuits developed can be downloaded on to the Xilinx hardware and probed using BoardScope [25]. BoardScope is a graphical and interactive hardware debug tool for Xilinx FPGAs. It enables a user to probe inside the chip and examine the internal states and circuit configurations while the hardware is in operation. The data is sampled using the readback capabilities of the FPGA and displayed. The interface to the hardware is provided by XHWIF, the Xilinx standard HardWare InterFace for FPGA based hardware [25]. Thus the JBits API along with BoardScope and XHWIF enables run-time reconfigurable application development, hardware debugging, and remote configuration capabilities. In order to support run time reconfiguration at the simulation level, the Virtex Device Simulator (VirtexDS) has been incorporated as part of the JBits tool suite [25]. VirtexDS operates at the device level and provides a software model of the entire Virtex family of FPGAs. The circuit being simulated is the actual FPGA implementation.

This tool suite has been used for the hardware implementation. The experimental results and discussion follow in the next section.

5. EXPERIMENTAL SETUP, RESULTS AND DISCUSSION

Simulations to illustrate the ideas in Section 3 have been carried out on a two dimensional array of gates. The chromosome structure used for learning the internal structure of the circuit is as shown in Figure 1. There are totally N components, with each component having the structure indicated in Figure 1.

![Figure 1: Chromosome structure](image-url)
Each block has the following fields:

G: Gene: Specifies whether it is a flip-flop or an AND, OR, XOR, NAND, NOR or XNOR gate
T: Type: Specifies whether the inputs are in the true form or complemented form.
I: Set of Inputs
O: Set of Outputs
F: Specifies the fitness value for the set of outputs
TF: Maximum fitness of the circuit
N: Number of gates: DimensionOfRow * DimensionOfColumn

It is assumed that each column of the array takes its inputs from the previous columns or from external inputs. The results obtained are for a population size of 75, with a probability of mutation of 0.02 and two-point crossover, using elitist selection [26].

A decade counter and a 1010 pattern detector, which consist of sequential as well as combinational components, are considered for simulation. As a proof of concept the decade counter has been implemented in hardware. The learning phase evolves the structure from the input / output combinations. One of the PRVs evolved for the decade counter is shown in Figure 2.

![Figure 2: Decade Counter: Genetically Learnt Structure](image-url)
Tables 1(a) to 1(c) show the usage of gates obtained by the learning algorithm for the decade counter. A 5 x 5 array of gates has been used. The highlighted entries indicate the unused resources. The PRV1 shown in Table 1(a) is the optimal of the three PRVs shown.

<table>
<thead>
<tr>
<th>Primary Replacement Version</th>
<th>Secondary Replacement Version 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 21 22 23 24</td>
<td>20 21 22 23 24</td>
</tr>
<tr>
<td>15 16 17 18 19</td>
<td>15 16 17 18 19</td>
</tr>
<tr>
<td>10 11 12 13 14</td>
<td>10 11 12 13 14</td>
</tr>
<tr>
<td>5 6 7 8 9</td>
<td>5 6 7 8 9</td>
</tr>
<tr>
<td>0 1 2 3 4</td>
<td>0 1 2 3 4</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Primary Replacement Version</th>
<th>Secondary Replacement Version 1</th>
<th>Secondary Replacement Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 21 22 23 24</td>
<td>20 21 22 23 24</td>
<td>20 21 22 23 24</td>
</tr>
<tr>
<td>15 16 17 18 19</td>
<td>15 16 17 18 19</td>
<td>15 16 17 18 19</td>
</tr>
<tr>
<td>10 11 12 13 14</td>
<td>10 11 12 13 14</td>
<td>10 11 12 13 14</td>
</tr>
<tr>
<td>5 6 7 8 9</td>
<td>5 6 7 8 9</td>
<td>5 6 7 8 9</td>
</tr>
<tr>
<td>0 1 2 3 4</td>
<td>0 1 2 3 4</td>
<td>0 1 2 3 4</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20 21 22 23 24</td>
<td>20 21 22 23 24</td>
<td>20 21 22 23 24</td>
<td>20 21 22 23 24</td>
<td>20 21 22 23 24</td>
</tr>
<tr>
<td>15 16 17 18 19</td>
<td>15 16 17 18 19</td>
<td>15 16 17 18 19</td>
<td>15 16 17 18 19</td>
<td>15 16 17 18 19</td>
</tr>
<tr>
<td>10 11 12 13 14</td>
<td>10 11 12 13 14</td>
<td>10 11 12 13 14</td>
<td>10 11 12 13 14</td>
<td>10 11 12 13 14</td>
</tr>
<tr>
<td>5 6 7 8 9</td>
<td>5 6 7 8 9</td>
<td>5 6 7 8 9</td>
<td>5 6 7 8 9</td>
<td>5 6 7 8 9</td>
</tr>
<tr>
<td>0 1 2 3 4</td>
<td>0 1 2 3 4</td>
<td>0 1 2 3 4</td>
<td>0 1 2 3 4</td>
<td>0 1 2 3 4</td>
</tr>
</tbody>
</table>

(c)

**Table 1: Decade Counter: Usage of Gates.** (a) For PRV1. (b) For PRV2. (c) For PRV3.

The left most matrices of the tables give the PRVs that are evolved by the learning process. The subsequent entries indicate the redistribution of the useful redundancy across the structure, and the multiple SRVs generated to provide 100% single component fault coverage. With reference to Table 1(c), the PRV1 has 0,1,5,7,13,17,21,22 and 24 as the redundant components. The second entry shows how these components are distributed across the structure so that there is at least one redundant component in each layer. Now this version cannot handle all the faults. Hence the subsequent versions in the other entries give an alternate configuration for each of the used resources. That is, each component is a redundant component in at least one of the
versions thus providing 100% single component fault coverage. The number of versions to be evolved ranges from two to a maximum of the dimensionality as indicated in Tables 1(a)-1(c). The PRVs in Tables 1(a)-1(c) have different number of SRVs to provide complete fault coverage viz., one, two and four respectively.

Tables 2(a) and 2(b) show the usage of gates obtained by the learning algorithm for a 1010 sequence detector. A 4 x 4 array of gates has been used. The highlighted entries indicate the unused resources. The PRV1 shown in Table 2(a) is the optimal of the two PRVs shown.

<table>
<thead>
<tr>
<th>Primary Replacement Version</th>
<th>Secondary Replacement Version 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 13 14 15</td>
<td>12 13 14 15</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>8 9 10 11</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4 5 6 7</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12 13 14 15</td>
<td>12 13 14 15</td>
<td>12 13 14 15</td>
<td>12 13 14 15</td>
</tr>
<tr>
<td>8 9 10 11</td>
<td>8 9 10 11</td>
<td>8 9 10 11</td>
<td>8 9 10 11</td>
</tr>
<tr>
<td>4 5 6 7</td>
<td>4 5 6 7</td>
<td>4 5 6 7</td>
<td>4 5 6 7</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

(b)

Table 2: Sequence Detector: Usage of Gates. (a) For PRV1. (b) For PRV2.

As a gate level implementation has been considered for simulation, the CLBs on the Virtex FPGA were configured as either logic gates or flip-flops. The PRVs 1 and 2 for the decade counter evolved by the learning process were implemented on the hardware. Faults were injected into the circuits. The injected faults caused discrepancies in the learnt input/output sequences, which were recognized by the monitoring phase. In order to correct the faults, the corresponding multiple secondary versions were tried out till the faults were corrected.

Figures 3 and 4 show the CLB configurations evolved for PRV1 and 2 as obtained from BoardScope. Only the flip-flop states are indicated here. The white color in the bottom left quarter of a CLB indicates that the flip-flop is in the OFF state and the white color in the bottom right quarter of a CLB indicates that flip-flop is in the ON state. The initial CLB configuration for PRV1 without any faults is shown in Figure 3(a). With the injection of a fault at CLB11, there is a discrepancy in the input/output sequence during Clock 2, which is detected by the monitoring phase. Because of this, there is a change in the state of this flip-flop as indicated in Figure 3(b). In order to overcome this fault,
SRV1 is downloaded to correct the fault. Figure 3(c) shows the corrected circuit. Referring to Figure 4 corresponding to PRV2, it can be seen that a fault injected at CLB 22 does not get rectified with its corresponding SRV1 as shown in Figure 4(c) and 4(d). Hence the next version SRV2 needs to be downloaded. Figure 4(e) shows that the fault gets corrected with SRV2.

![Figure 3: CLB configurations for PRV1. (a) Without any faults. (b) With fault injected at CLB 11. (c) SRV corresponding to PRV1.](image)

![Figure 4: CLB Configurations for PRV2. (a) Without any faults. (b) With fault injected at CLB 22. (c) SRV1 corresponding to PRV2 without any faults. (d) SRV1 corresponding to PRV2 with fault. (e) Fault correction with SRV2.](image)
It is to be noted that the learning phase evolves the PRV and the corresponding SRVs off-line. Hence the system operation is not affected. The only factor contributing to the system downtime is the download time of the PRV and the SRVs, if need be. But this is also minimised because of the reduction in the number of versions for complete fault coverage as demonstrated in this work. This idea can be easily extended to guarantee 100% correction of n simultaneous faults by providing n layers of fault tolerant coverage on the evolved structure.

6. CONCLUSION

This work has demonstrated a genetic algorithm based fault tolerant cover, which learns multiple versions of the structural configuration of the circuit under study from its behavioural description, even when there is no access to the internals of the circuit. It continuously checks for errors and corrects them using the learnt configurations. The maximum number of reconfigurations required for correction depends on the dimensionality of evolution. This framework provides a systematic approach for fault detection and correction of any digital system. The efficacy of this approach when only part of the behaviour is known has to be explored. Also the faults that may occur in the interconnections has not been taken into account. This needs to be probed further.

REFERENCES


