THE ANNALS OF "DUNAREA DE JOS" UNIVERSITY OF GALATI
FASCICLE III, 1998 YEAR, ISSN 1221-454X
ELECTROTECHNICS, ELECTRONICS, AUTOMATIC CONTROL, INFORMATICS

SELF-ADAPTATION AND SELF-REPAIR IN EVOLVABLE HARDWARE

Rustem POPA, Mircea ILIEV

"Dunărea de Jos" University of Galati, Automatic Control and Electronics Department, Domnească Street - 111, Galați - 6200, Romania, Tel. / fax: 036.460182, e - mail: rpopa@ac.ugal.ro

Abstract: Evolvable Hardware is a hardware which modifies its structure in order to adapt to the environment in which it is embedded. It is implemented on a programmable logic device, whose architecture can be altered by downloading a binary bit string called *architecture bits*. The architecture bits are adaptively acquired by genetic algorithms. In this paper we considered a simple Finite State Machine and we proved through simulation that the evolved circuit is a flexible and fault-tolerant structure which responds in real-time to a changing environment.

Keywords: Genetic Algorithms, Evolvable Hardware, Machine Learning, Programmable Logic Devices, Fault-Tolerant Systems.

1. INTRODUCTION

Evolvable Hardware is a hardware built on a software reconfigurable logic device, such as Programmable Logic Device (PLD) and Field Programmable Gate Array (FPGA). Evolvable Hardware architecture can be reconfigured through the evolutionary method so as to adapt to the new environment. If hardware errors occur or a new hardware functionality is required, Evolvable Hardware can alter its own hardware structure in order to accommodate such changes.

The target task of the gate-level Evolvable Hardware is a Boolean concept formation. An *n*-variable Boolean function is defined as a function whose ranges and domain are constrained to have 0 (false) or 1 (true) values, i.e.

$$y = f\left(x_1, x_2, \ldots, x_n\right) = \begin{cases} 0, & \text{false value} \\ 1, & \text{true value} \end{cases} \qquad (1)$$

where the *n* variables are

$$x_1 \in \{0,1\} \wedge x_2 \in \{0,1\} \wedge \ \ldots \ \wedge x_n \in \{0,1\}\,. \qquad (2)$$

The goal of Boolean concept formation is to identify an unknown Boolean function, from a given set of observable input and output pairs, i.e.

$$\left\{\left(x_{i1}, x_{i2}, \ldots, x_{in}, y_i\right) \in \{0,1\}^{n+1} \mid \ i = 1, \ldots, N\right\}, \qquad (3)$$

where *N* is the number of observations. In general, *N* is less than the maximum possible number of 2^{2^n} distinct *n*-variable Boolean functions. Since the ratio of the size of the observable data to the size of the total search space, $\frac{N}{2^{2^n}}$, drastically decreases with *n*, effective generalizing ability is required for Boolean concept learning. Some evolutionary methods have been proposed for this purpose (Iba, *et al.*, 1996; Michalewicz, 1994).

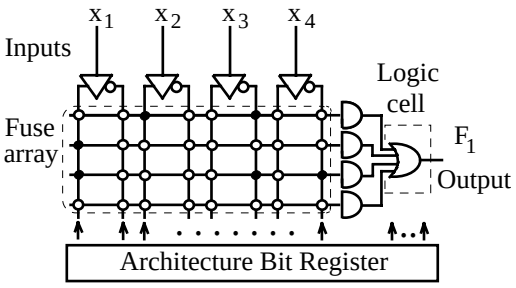

Fig. 1. A simplified PLD (Programmable Logic Device) Structure

In Evolvable Hardware adaptation takes the form of direct modification of the hardware structures according to rewards received from the environment. This results in a number of advantages. Adaptation in real-time is feasible due to a speed-up by many orders of magnitude. The system will be flexible and fault-tolerant since this new hardware can change its own structure in the case of environmental change or hardware error.

The feasibility of hardware evolution in combinational circuits was reported in some papers. Higuchi, *et al*. (1994) has implemented four Boolean functions for a robot control system, and Zebulum, *et al.* (1996) has designed some basic circuits as multiplexer 4-1 or decoder 3-8. Our goal is to demonstrate that an evolved sequential circuit is an adaptive and fault-tolerant structure.

The rest of this paper is structured as follows. Section 2 describes the fundamental principle of Evolvable Hardware. Section 3 discuss in more detail the genetic learning component of the hardware. Section 4 presents a case study of evolved structure of a Finite State Machine and the experiments of adaptation and reparation. Section 5 points the conclusions of our experiments.

2. EVOLVABLE HARDWARE

There is a clear distinction between a conventional hardware and an Evolvable Hardware. A designer can begin to design a conventional hardware only after its detailed specification is given. In this sense, conventional hardware is a top-down approach. However, Evolvable Hardware is applicable even when no hardware specification is known before. Its implementation is determined through a genetic learning in a bottom-up way. This kind of hardware is a combination of reconfigurable hardware devices and genetic learning. Multiple hardware structures are maintained in parallel and they are continuously evaluated by genetic algorithms in order to create better hardware structures.

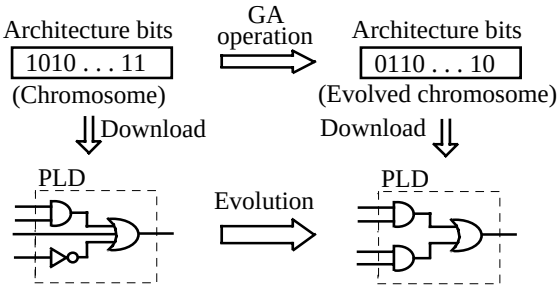

Fig. 2. The basic idea of Evolvable Hardware.

The basic idea is as follows. In reconfigurable hardware devices like PLDs and FPGAs, the logic design is compiled into a binary bit string. By changing the bits, arbitrary hardware structures can be implemented instantly. The key idea is to regard such a bit string as a chromosome of a genetical algorithm. Through genetic learning, Evolvable Hardware finds the best bit string and reconfigures itself accordingly.

A PLD consists of logic cells and a fuse array (see figure 1). The architecture of the PLD is determined by architecture bits stored in an architecture bit register. Each link of the fuse array corresponds to a bit in the register.

The fuse array determines the interconnection between the device inputs and the logic cell. It also specifies the logic cell's AND-term inputs. If a link on a particular row of the fuse array is switched on, which is indicated by a black dot in figure 1, then the corresponding input signal is connected to the row. In the architecture bits, these black and white dots are represented by 1 and 0, respectively.

Consider the example shown in figure 1. The first row indicates that x_2 and $\overline{x_3}$ are connected by an AND-term, which generates the minterm $x_2 \cdot \overline{x_3}$. Similarly, the second row generates x_1, and the third row generates $x_1 \cdot \overline{x_3} \cdot \overline{x_4}$. On the last row, no inputs are connected by an AND-term, thus, the resultant output is $F_1 = x_2 \cdot \overline{x_3} + x_1 + x_1 \cdot \overline{x_3} \cdot \overline{x_4}$.

As mentioned above, both of the fuse array and the functionality of the logic cell are represented in a binary string. This binary string is regarded as a chromosome of a genetic algorithm, and is down-loaded onto a PLD, on and after the genetic learning. In this way, the hardware structure is adaptively searched by genetic algorithm (see figure 2).

Higuchi, *et al*. (1994) proposed an architecture based on software reconfigurable devices (PLDs) and a parallel genetic algorithm hardware.

3. GENETIC LEARNING

The genotype of an evolved structure on PLD basis is given by the bits for fuse array and bits for logic cells. However, this genotype representation has inherent limitations, since the fuse array bits are fully included in the genotype, even in the case that only a few bits are effective. This causes the increase of the chromosome length, increasing execution time of the algorithm.

Iba, *et al*. (1996) proposed a new method for chromosome representation. Each chromosome is represented by the architecture bits, which effectively determine the hardware structure. For example, we need only six bits for fuse array in figure 1, while 32 bits are required for initial genotype representation. Because of this short chromosome, this method known as Variable Length Chromosome Genetic Algorithm (VGA) can increase the maximum evolvable circuit size and establish an efficient adaptive search.

The fitness function is determined depending on the application , but it basically evaluates the corectness of the output for the training data set.

In the case of fault-tolerant applications, Evolvable Hardware works in parallel with the target circuit. The I/O patterns of the target circuit are observed by the Evolvable Hardware. While the target circuit is working, the Evolvable Hardware evolves the circuit by genetic algorithm. The fitness value is the number of the correct outputs of the evolved circuit.

In the case of any unpredictable changes of the environment, Evolvable Hardware can adapt himself to these changes, through on-line genetic learning. The adaptation result is configured into a new hardware structure on the spot.

We have used the fundamental structure of a genetic algorithm. The initial population of chromosomes is constructed randomly. All these potential solutions are evaluated using a fitness function. In our case, fitness is the ratio between the number of the correct values of the binary function and the number of all possible values of the function.

The next step is selection and reproduction. For each individual, a number of copies are made, proportional to its fitness, while keeping the population size constant. The least fit individuals are deleted. This is the survival of the fittest part of the genetic algorithm.

The next step is crossover, where individuals are chosen two at a time, at random, as parents. They are converted into two new individuals, called offsprings, by exchanging parts of their structure. Thus, each offspring inherits a combination of features from both parents. We have used in our experiments only one point crossover.

The next step is mutation. An incremental change is made to each member of the population, with a small probability. After mutation is performed on an individual, it no longer has just the combination of features inherited from its two parents, but also incorporates the additional change caused by mutation. This ensures that the algorithm can explore new features that may not yet be in the population. It makes the entire search space reachable despite the finite population size.

This completes the production of a new generation. This process is repeated for several generations, and the fittest gate permutation seen in the entire run is output at the end.

We have taken a simple example with 4 inputs functions and maximum 4 minterms for each function. The number of columns in the fuse array is 8 (i.e. 4 inputs, where each input signal is divided into 2 columns) and the number of rows is equal with 4, i.e. the maximum number of minterms. Therefore, the number of fuse array links is 32, and we have considered this number as the total length of the chromosome. We have supposed the logic cell's function to OR gate function, and so, the bits to set the logic cell's function have been ignored.

Our genetic algorithm uses the population size of 100, each chromosome has 32 bits. One point crossover is executed with a probability of 50% and the mutation rate is 1%. A number of 10 worse chromosomes are replaced each generation. The stop criterion is the number of generations.

Evolvable Hardware is feasible in PLD structures like GAL16V8 chips, for example. This chip consists of an AND array and 8 logic cells configurable as OR gate and register device through some special configuration bits. The execution time is 10 nanoseconds, so the reconfiguration can be very fast.

4. EXPERIMENTS

We organized three experiments for the learning task described before. The evolved circuit is a simple synchronous sequential machine with 3 states and 2 inputs. The state diagram corresponding to this circuit is represented in figure 3. The evolved circuit based on a PLD structure like GAL16V8 is represented in figure 4.

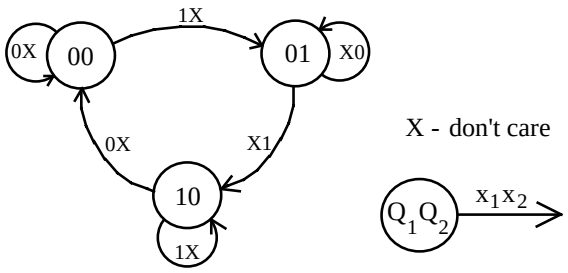

Fig. 3. The state diagram of the evolved circuit.

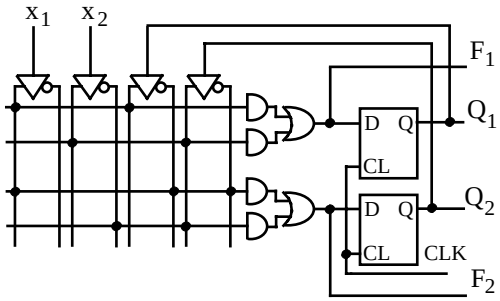

Fig. 4. The evolved circuit.

Actually a circuit generating 2 binary functions is evolved. However, this circuit is built from 2 independent circuits, each generating one output bit, F_1 and F_2. Therefore, the evolution of a circuit with one output bit is repeated 2 times. Figure 5 shows the result. The Y axis is the correct answer rate. If it reaches 100%, then the hardware evolution succeeds. The first circuit is successfully obtained at 12-th generation and the second circuit at 32-th generation.

The same evolution, but with a significant less number of chromosomes in population, is represented in figure 6. We have used here only 20 chromosomes in population, instead 100 chromosomes, like in the previous case. We can see that the first circuit is successfully obtained at 21-st generation, while the second circuit is obtained at 68-th generation. These experiments show that the evolution is faster in populations with a larger number of chromosomes, but the parallel hardware necessary for a large population might be unacceptable. Thus, a compromise between speed and complexity must be found in this area.

Evolution may provide some non-minimal expressions for boolean functions. For example, a minimal expression of the first function is $F_1 = x_1 \cdot Q_1 + x_2 \cdot Q_2$, but another expression like $F_1 = x_1 \cdot Q_1 + x_2 \cdot Q_2 + x_1 \cdot x_2 \cdot \overline{Q_1} \cdot Q_2$ is possible as a result of evolution. Actually, it's not a real problem, because minimization is not necessary for PLD implementations .

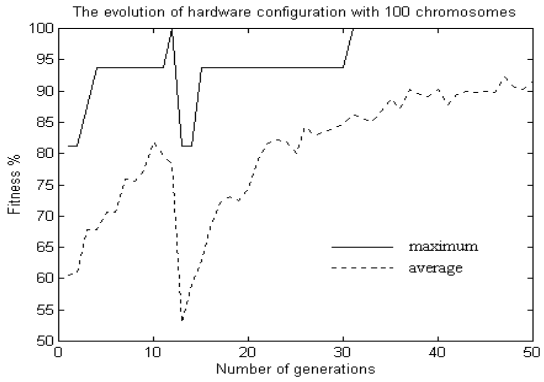

Fig. 5. Evolution with a population of 100 chromosomes

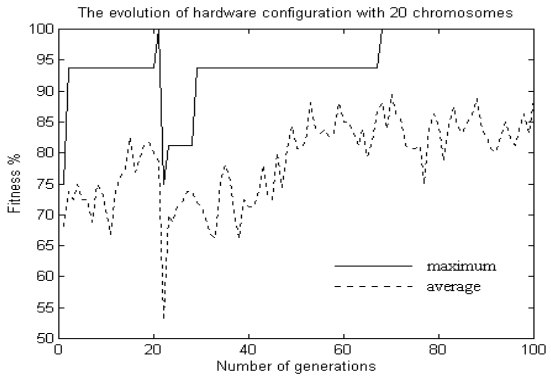

Fig. 6. Evolution with a population of 20 chromosomes

The second experiment was organized to confirm the self-repair property of the Evolvable Hardware. Let's consider the evolved circuit in figure 4 as the target circuit, and another Evolvable Hardware in parallel with the target circuit. The inputs of these two circuits are connected together and the outputs are observed. While the target circuit is working, the Evolvable Hardware evolves the circuit by genetic algorithm. When the evolution is over, the outputs of the two circuits are equal, i.e. $F_1 = F_1^*$, $F_2 = F_2^*$, $Q_1 = Q_1^*$, and $Q_2 = Q_2^*$.

When a fault occurs in one of the circuits, one of the above relations is not true and a new evolution repairs the fault through reconfiguration. We have supposed a "stuck-at 0" fault in the AND array of the PLD. Figure 7 shows the new circuit obtained through reconfiguration. The evolution is based on a genetic algorithm with the same parameters as shown before (see figure 8).

The third experiment was organized to confirm whether Evolvable Hardware can adapt to environmental change or not. Let's suppose that our circuit commands a motor with two control bits,

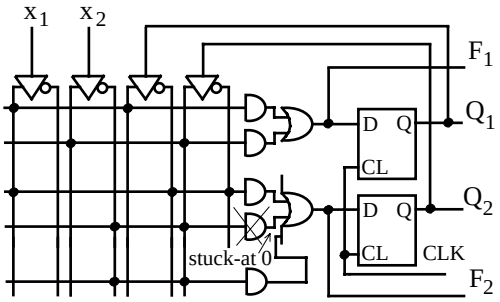

Fig. 7. Self-repair in Evolvable Hardware.

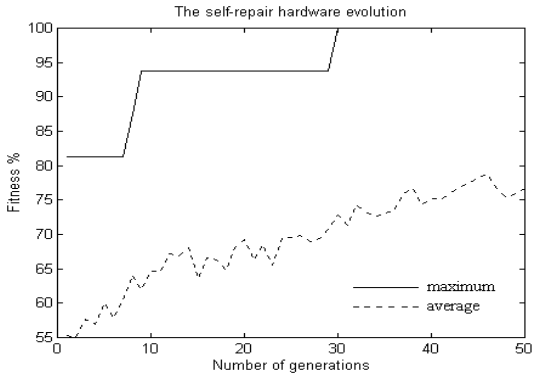

Fig. 8. Self-repair of the circuit through evolution.

according with the next rule: $Q_1Q_2 = 00$ for stop, $Q_1Q_2 = 01$ for half-speed, and $Q_1Q_2 = 10$ for full-speed motion. We examined the hardware adaptability in a situation where the motor become unable to operate at full speed any more and half of speed is the only choice. This environmental change corresponds to learning two new binary functions according with this new behaviour. Figure 9 shows the new hardware configuration and figure 10 shows how the hardware evolution goes after the re-adaptation starts.

We have organized this experiment starting first with the population of chromosomes which already learned the original circuit, and then with an initial population generated randomly, but we could not find that there exist significant differences between these two cases.

5. CONCLUSIONS

In this paper, we have shown that the advantages of using Evolvable Hardware are huge, due to the attributes of self-repair and self-adaptation. Fault tolerant and flexible design is realized because Evolvable Hardware can change its own structure in the case of hardware error or environmental change, utilizing its on-line adaptation capability.

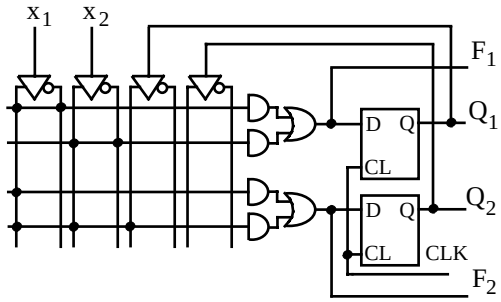

Fig. 9. Self-adaptation in Evolvable Hardware.

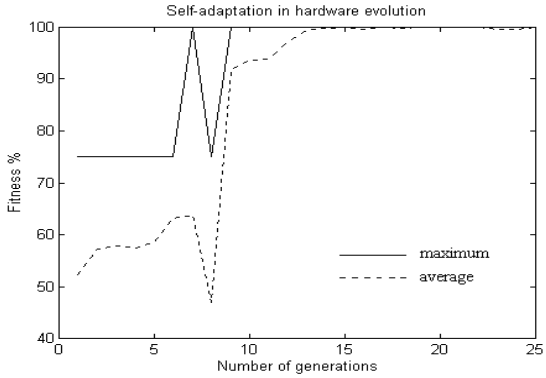

Fig. 10. Self-adaptation of the circuit

The execution speed of the evolved system will be extremely fast, because the result of adaptation is the hardware structure itself (Higuchi, *et al*., 1994).

We have shown that Evolvable Hardware can implement Finite States Machines. Thus, exists the possibility that all microprocessor's control programs could be replaced with Evolvable Hardware.

REFERENCES

Higuchi, T., H. Iba and B. Manderick (1994). Applying Evolvable Hardware to Autonomous Agents. In: *Proc. of the Third Conf. on Parallel Problem Solving from Nature, Jerusalem, Israel, October 1994,* 524-533

Iba, H., M. Iwata and T. Higuchi (1996). Machine Learning Approach to Gate-Level Evolvable Hardware. In: *Proc. of the First International Conference on Evolvable Systems, ICES'96, Tsukuba, Japan, October 1996,* 327-343

Michalewicz, Z. (1996). *Genetic Algorithms+Data Structures = Evolution Programs,* Springer-Verlag, Berlin-Heidelberg-New York

Zebulum, R. S., M. A. Pacheco and M. Vellasco (1996). Evolvable Systems in Hardware Design: Taxonomy, Survey and Applications. In: *Proc. of the First International Conference on Evolvable Systems, ICES'96, Tsukuba, Japan, October 1996,* 344-357