Self-evolving Hardware

Gregory J. Barlow and Marc A. Edwards

North Carolina State University
Department of Electrical and Computer Engineering
Box 7911, Raleigh, NC 27695
gjbarlow@eos.ncsu.edu, maedward@eos.ncsu.edu

Abstract

A complete approach to self-contained evolvable hardware, called self-evolving hardware, is presented. Self-evolving hardware implements the entire evolutionary process and reconfigurable area inside a single FPGA. Unlike previous hardware implementations, self-evolving hardware is capable of evolving both logic and connectivity at run-time. Self-evolving hardware increases the speed, flexibility, and scalability of the evolutionary process, facilitating the implementation of complex circuits.

1 Introduction

Evolvable hardware has shown itself to be an effective method for developing useful circuits. However, several major barriers still stand in the way of developing large and complex designs. Evolvable hardware systems must be capable of scaling to large reconfigurable areas if they are to be a method for practical design. To be capable of realizing the most possible circuits the evolution should be flexible, with as few constraints as possible. Systems must significantly improve the speed of the evolutionary process to compensate for the increases in time needed to produce more complex circuits with increasingly extensive fitness tests.

This paper presents self-evolving hardware, the most exhaustive approach to date which implements a completely self-contained evolvable hardware system. Self-evolving hardware instantiates the entire evolutionary process and reconfigurable area inside a single Field Programmable Gate Array (FPGA). Self-evolving hardware is designed using parallelism and partial run-time reconfiguration to provide fast and flexible evolution of a large reconfigurable area. With the addition of an external memory, a completely independent module is obtained, which could potentially be duplicated and placed in parallel to form scalable evolutionary platforms.

In Section 2, brief background information on evolutionary algorithms and evolvable hardware is provided. Section 3 describes the operation of self-evolving hardware, along with its challenges and benefits. Section 4 describes the *HereBoy* evolutionary algorithm used by self-evolving hardware. System design is presented in Section 5, with the evolution of logic and connectivity described in Section 6. Section 7 outlines how the connectivity configuration is used to reroute the reconfigurable area, and potential applications for self-evolving hardware are described in Section 8.

2 Background

Evolutionary algorithms are a biologically inspired computational method for problem solving [7]. Genetic operations are performed on a population of chromosomes (configuration data strings), and the fitness of each individual is used to form the new generation. Like natural selection, evolutionary algorithms tend to converge on an optimal solution. Evolvable hardware applies evolutionary algorithms to circuit design in reconfigurable hardware. Evolved systems consider a larger solution space than a human designer, with the potential to create systems that are unachievable through traditional design methodologies. Many types of devices can be used for evolvable hardware [11]; one of the most common types used is the Field Programmable Gate Array (FPGA).

Much of the early work with evolvable hardware implemented using FPGAs relied on the transient analog characteristics of the device [12]. While effective, the resulting circuits proved highly device and temperature specific, making robust designs difficult to achieve. The inconsistencies result from the fact that the FPGA is designed to be a purely digital device. Evolving hardware which relies only on the digital characteristics of the FPGA produces circuits which are both synchronous and reproducible across temperature and hardware variations.

Evolvable hardware is divided into three main approaches. Extrinsic evolution implements the entire evo-

lutionary process in software using a simulator or abstraction of the hardware. Intrinsic evolution performs the evolutionary process in software, but uses the physical hardware to test each individual. The third approach implements the entire evolutionary process within reconfigurable hardware, producing systems capable of self-evolution. Of the three approaches, the last has received the least attention.

Several self-contained evolvable hardware systems have been proposed. While some have been successfully implemented [13, 9], all place serious restrictions on evolution. Complete Hardware Evolution, the most robust implementation of this approach to evolvable hardware, initially used a chromosome size of only 64 bits, which is only enough data to configure four look-up tables (LUT) in a current FPGA. Other proposed methods for self-contained evolvable hardware severely limit routing and/or logic configuration [4, 5, 10].

3 Self-evolving Hardware

3.1 Overview

Self-evolving hardware implements the entire evolutionary process within an FPGA. Self-evolving hardware presents a number of advantages over previous evolvable hardware techniques, including parallelism, speed, evolution for both logic and connectivity, and adds the potential for sequential logic. Self-evolving hardware is comprised of a series of modules which implement the evolution process, similar to Tufte and Haddow's Genetic Pipeline [13]. Self-evolving hardware must overcome the challenges present in current FPGA technology.

3.2 Challenges

A great deal of evolvable hardware research has used the Xilinx 6200, which has many evolution friendly features [6]. Some of the features of current FPGAs are less conducive to evolvable hardware. As a result of these architectural limitations, it is necessary to place constraints on the configuration to ensure that no damaging or asynchronous circuits are produced.

The possibility for an FPGA to be damaged during hardware evolution stems from the lack of physical limitations to multiple line drivers. Multiple drivers occur when two internal logic outputs drive the same line. The two drivers can conflict if they reach opposite logic states. The conflicting states produce a current overload which can eventually destroy the device. Therefore, the configuration must be constrained to not allow such conflicts.

Current FPGAs include no restrictions on the instantiation of asynchronous logic. Asynchronous race conditions must be avoided because the resulting system output will be inconsistent over time. The presence of race conditions will also increase dependence on temperature and device specific factors. Race conditions must be averted without impeding the flexibility of the evolvable system.

To avoid similar conditions to asynchronous logic, driver fanout must be considered when evolving circuits. Problems arise when a single logic source is connected to too many sinks. When this is the case, the logic can no longer drive the loads in a timely or reliable fashion. Thus, restrictions based on the characteristics of the particular device in use must be accounted for.

Finally, to achieve truly self-evolving hardware with the ability to alter logic connectivity, partial run-time reconfiguration is necessary. Evolving the routing configuration requires both the means to randomly change connectivity while accounting for the aforementioned issues, and the ability to translate the desired changes into physical configuration data. To maintain complete encapsulation the device must then be reconfigured to implement the evolution without interrupting the operation of the unaffected modules.

3.3 Benefits

Self-evolving hardware has a number of benefits compared to other evolvable hardware techniques. Both logic configuration and connectivity are included in the evolutionary process, providing maximum flexibility. While many systems allow the evolution of combinational logic, self-evolving hardware also includes structures to facilitate the generation of sequential logic. Self-evolving hardware is designed to provide a scalable and robust hardware evolution platform using current FPGA technology. The use of standard hardware makes custom evolutionary architectures, which can be prohibitively expensive and complex, unnecessary for useful evolutionary development.

A structural description analogous to a netlist is used to represent connectivity of elements in the reconfigurable area. This scheme prevents multiple drivers without further constraining connectivity. Also inherent in the netlist is a dynamic method to control fanout. In addition, the fundamental units of self-evolving hardware prevent asynchronous conditions from being instantiated. Finally, by directly connecting the system memory to the parallel configuration port and preserving critical system resources, internally controlled partial run-time reconfiguration is possible.

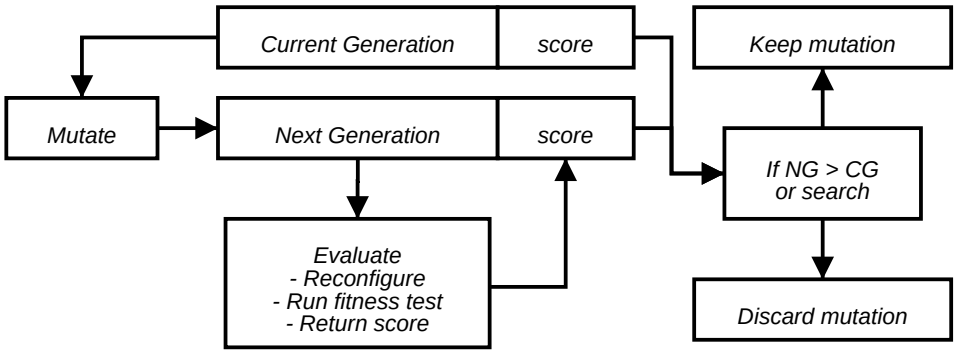

Figure 1: Evolutionary system

3.4 Evolutionary System

Self-evolving hardware is comprised of a series of modules which implement the evolutionary process as shown in Figure 1. A seed is loaded and scored to form the initial generation. During the evolutionary process, the netlist and logic configuration data for the current generation are mutated using the *HereBoy* evolutionary algorithm to form the new generation. The mutated netlist is then converted to actual routing information by the routing module. Using the new routing and configuration information, the device is reprogrammed, updating the reconfigurable area. The fitness test is run on the new circuit and a score for the new generation is returned. The selection characteristics of the *HereBoy* algorithm determine which configuration carries over to the next iteration. This process is repeated until a satisfactory solution is evolved.

4 Evolutionary Algorithm

4.1 Requirements

While many evolutionary algorithms are suitable for evolvable hardware, self-evolving hardware presents an additional set of constraints on algorithm selection. First, evolvable hardware requires a very large chromosome. For example, Thompson's tone discrimination circuit, which used an extremely small evolvable area, had a chromosome of 1800 bits. [12] For current FPGA technology, like the Xilinx Virtex series, each LUT requires sixteen bits of configuration data. A block of 1600 LUTs would require a chromosome of 25600 bits for the logic configuration alone. Routing configuration adds an additional chromosome, even longer than the logic chromosome. As the chromosome size increases, population storage becomes an issue. The length of these chromosomes makes storage of numerous individuals expensive. Many of the real world problems where evolvable hardware is applicable have many acceptable solutions or no known optimal solution. Rapid convergence on a good solution is therefore more desirable than absolute optimization. Last, convenient seeding of the evolutionary algorithm is desirable. For many physical systems, it is more efficient to initially simulate the environment and then use the result as a seed for the actual evolution.

4.2 HereBoy Evolutionary Algorithm

The *HereBoy* evolutionary algorithm [8] combines features from several classes of evolutionary algorithms. *HereBoy* evolves a single individual using adaptive mutation and adaptive search. The number of bits mutated for a given iteration is defined in Formula 1. The number of bits mutated is dependent on *M*, the maximum mutation percentage, and *L*, the length of the chromosome. As the individual converges on the optimal solution, the number of bits which undergo mutation decreases.

$$N_{mutate} = M * L * \frac{MaxScore - MaxCurrentScore}{MaxScore} \quad (1)$$

The search rate is also adaptive to the distance from the optimal solution. The probability of keeping a worse solution is given by Formula 2. The search probability depends on *S*, the maximum search probability. As the individual converges on the optimal solution, the search probability decreases.

$$P_{search} = S * \frac{MaxScore - MaxCurrentScore}{MaxScore} \quad (2)$$

HereBoy offers a number of benefits over other evolutionary algorithms for use in self-evolving hardware. While single bit mutation does not scale well as chromosome sizes become very large, the number of bits mutated by *HereBoy* depends on the chromosome length. Less storage is required than for a typical genetic algorithm, because *HereBoy* operates on a single individual, rather

than on a large population. One of the main advantages of *HereBoy* is its ability to rapidly converge on a solution. While its performance very near the optimal solution is only as good as simulated annealing, *HereBoy* is more effective for the first 99% of evolution. Because *HereBoy* operates on a single individual, it is more receptive to seeding than algorithms which use a population. Seeding can aid in quicker convergence on a solution. A seed configuration is simply used in place of a randomly generated initial chromosome.

5 System Design

Self-evolving hardware is composed of a series of modules as shown in Figure 2. These modules and the supporting logic provide a series of building blocks; when combined a complete process for evolving complex hardware emerges.

5.1 Reconfigurable Logic Cells

The Reconfigurable Logic Cell (RLC) is the primary structure of logic evolution, duplicated throughout the reconfigurable area. The reconfigurable area consists of any number of RLCs, limited only by available space. Each RLC consists of four basic building blocks, which can be any combination of Synchronous Gate Elements (SGE) and edge-triggered D-type flip-flops, as shown in Figure 3. A SGE is a 16 x 1 bit 4-input LUT whose output is connected directly to the input of a clocked flip-flop [15]. An individual RLC can have any of five possible combinations of SGEs and flip-flops. Any ratio of these five combinations can exist in the reconfigurable area. It is plausible that the configuration of the RLCs could be included in the chromosome and evolved in conjunction with the SGE configurations; however, there is no significant evidence at present to suggest that this would improve the system capabilities and thus these parameters are set prior to evolution.

5.2 Fitness Test

Evaluation of a circuit's fitness occurs every generation and typically takes a significant, if not the major portion of the overall evolution time. If the time for evaluation becomes too great, it prohibits the ability of the system to traverse enough generations to reach a solution in a reasonable amount of time. This reality often makes the fitness function one of the most significant barriers to practical circuit evolution. By implementing the fitness inside the FPGA it not only keeps the entire evolutionary process internal to the device, but can also take advantage of hardware parallelism to significantly reduce evaluation time. These time savings give self-evolving hardware the potential to evolve larger and more complex designs.

5.3 Random Numbers

Random numbers are used throughout the evolutionary process, thus a source of uniformly distributed random numbers is essential. A module consisting of n parallel linear feedback shift registers (LFSR), where n is the length in bits of the random number desired, is used to implement the generator. LFSRs output a non repeating bit sequence of $2^m - 1$ bits in length, where m is the length of the LFSR in bits [1]. The parallel LFSR implementation will produce hundreds of thousands of spectrally clean random numbers. Each LFSR is loaded with a randomly generated seed during the initial device configuration, to ensure unique bit streams are produced. The LFSR is especially well suited for implementation in FPGAs because they consume relatively few resources and can operate at high speeds without any special considerations.

6 Hardware Evolution

6.1 Logic Configuration

The logic configuration chromosome is composed of SGE configuration data and clock decimation factors, which affecting the triggering of the stand-alone flip-flops. The SGE configuration data can be directly mapped into the configuration string for the reconfigurable area. The decimation factors control the clock rates which are used to trigger the stand-alone flip-flops.

The SGE is a suitable building block for hardware evolution because it provides more than adequate flexibility for functional evolution while alleviating the problems of race conditions and multiple driver contention. The 16-bit LUT in the SGE allows for the modeling of 2^{16} different combinational logic structures. The binary nature of the string loaded into each LUT allows for a simple 1 to 1 mapping in the chromosome giving the evolutionary algorithm maximum granularity for evolving digital gates. By pushing the output of the LUT through a flip-flop triggered by a clock, global to the evolutionary area, race conditions are avoided. The structured nature of the SGE's input and output ports are mapped by the netlist which prevents multiple drivers.

The stand-alone flip-flops are used to enable the evolution of sequential logic structures. The combinational logic is only able to propagate at the rate of the global clock. To enable multiple levels of logic to be traversed before being clocked into the storage flip-flops, a limited

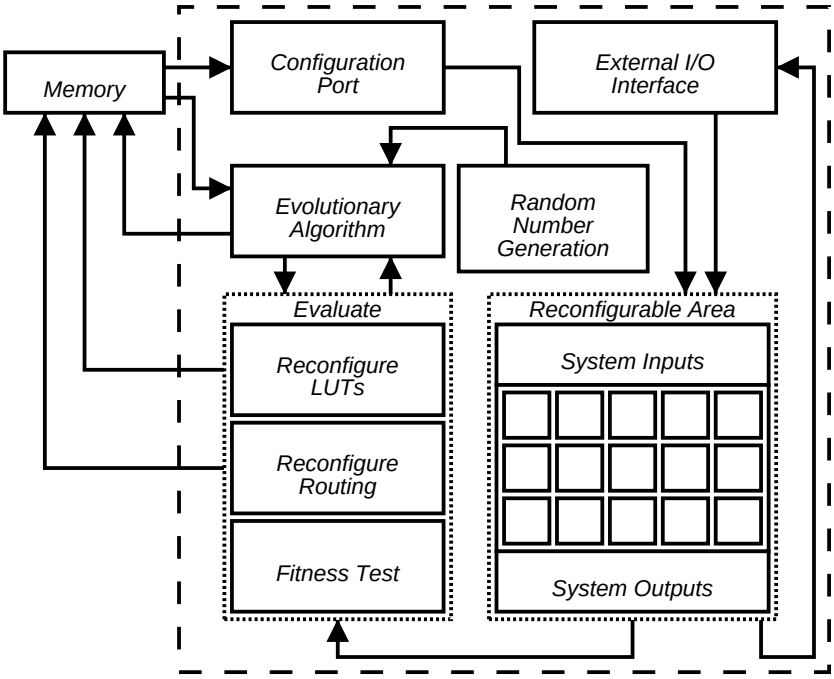

Figure 2: System diagram

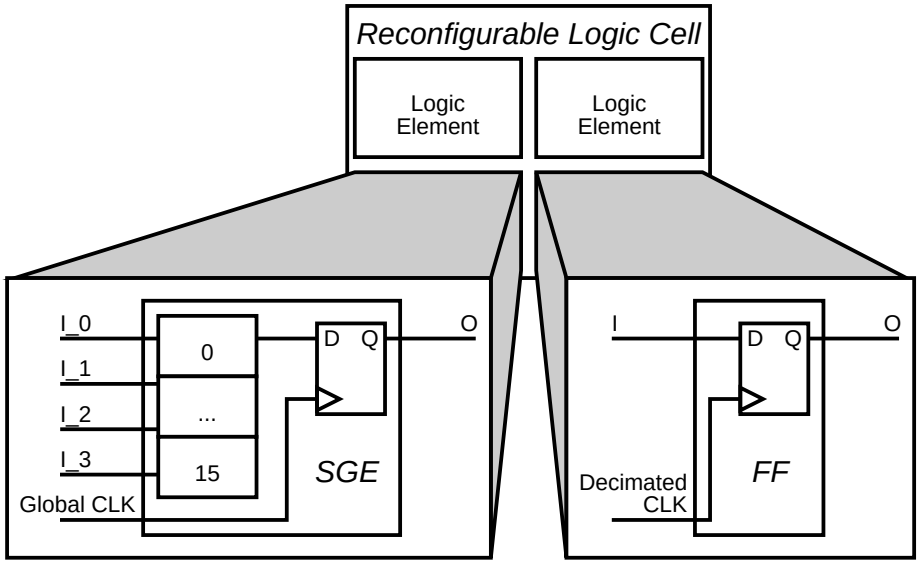

Figure 3: Reconfigurable Logic Cell

number of decimations to the global clock are generated. Each flip-flop is clocked by one of the decimations to the global clock, allowing the levels of logic traversed to be equal to the decimation factor. The amount of decimation for each of the sub-clocks is represented in the chromosome and mutated by the evolutionary algorithm.

6.2 Netlist Generation

In addition to controlling the type of instantiated gates, the evolutionary algorithm controls the port connections between the SGEs, flip-flops, and system I/O. The ability to adjust these connections greatly increases the possible solution space, increasing the probability that a suitable design will be evolved. Due to the challenges previously stated, it is not feasible to allow the evolutionary algorithm to directly evolve the routing configuration. In addition to damaging configurations there are likely to be more resources available than are actually used by an implementation. Thus, if given direct mapping to the routing matrix, many mutations would place unconnected routing and possibly break the connections between active blocks and the rest of the evolutionary area. To ensure that only legal configurations are produced, an evolvable netlist can be generated from two lists of ports, shown in Figure 4.

The first list contains all inputs to the SGEs, flip-flops and system outputs. The input list contains one instance for each input port; each port can only be paired with a single output. By forcing each input to connect to a single output, it is not possible for multiple drivers to be instantiated. To store the list, one need only have a small register or memory block for each input. The register or memory block for each input contains pointers to the specific configuration information for the input and its associated output port. When the evolutionary algorithm mutates the connectivity, random input ports are selected and reconnected. The old connection is released and a new connection is randomly selected from the list of available outputs, a process that is described in the next section. This spartan representation allows the evolutionary algorithm to manipulate the connections with speed and few memory resources.

The second list contains all outputs from the SGEs, flip-flops, and system inputs. While only one connection is allowed for each input, outputs may be connected to as many input ports as desired. The number of times each output may be connected is given by n, set prior to evolution and limited by FPGA resources and electrical issues related to fanout and propagation delay. At minimum, n must be four to ensure that no input is left floating. Floating inputs could create inconsistent output over time. The register or memory block for each output contains n flag bits and a pointer to the specific configuration information for the output. Each time a given output is paired or unpaired the corresponding flag bit is toggled.

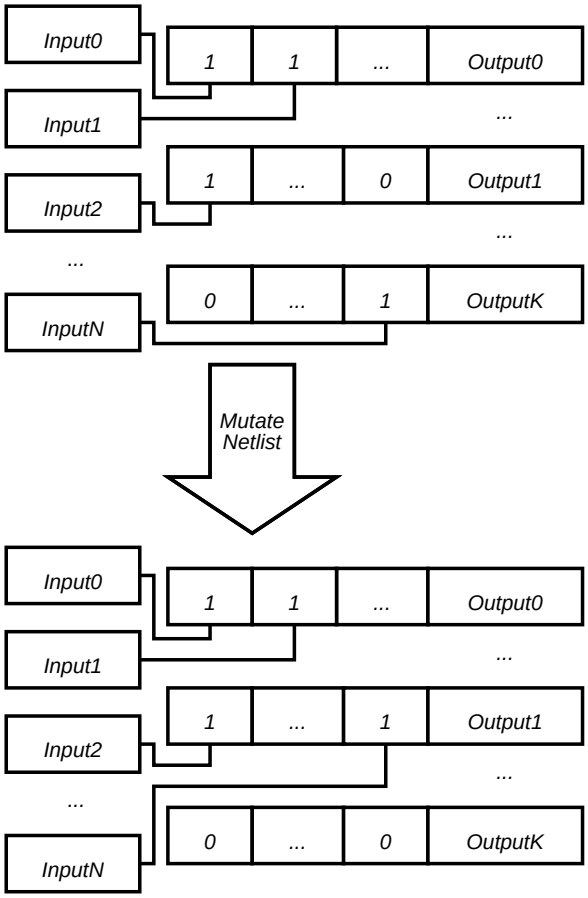

Figure 4: Netlist

7 Routing

Once a netlist is created there are still several challenges before the routing in a device can be changed. The first is reconfiguring the routing of the device while not disrupting the operation of the fixed system modules. The second difficulty is performing the changes in a timely manner. The time needed to determine the routing configuration and to physically update device hardware are both factors. Intrinsic and Extrinsic hardware evolution use traditional routing tools and re-route the entire device; this is time consuming and requires an entire external platform. As described in the previous section, the only changes in routing are those to the connections in the reconfigurable area, which are mutated during the netlist evolution. The number of modifications is very small relative to the overall number of connections in the system. Thus, it is feasible to reroute only these connections without disturbing

any other blocks.

Current FPGA technology is capable of changing configuration data for sectors not performing critical tasks without creating any transient signals or breaking buses that traverse the affected region [14]. This feature makes it possible to internally control the reprogramming of the FPGA.

To physically implement the connectivity changes, it is necessary to develop a structural representation of the routing matrix. This representation must include information about which resources are used by critical modules and are not available for rerouting. To facilitate this, a snapshot of used resources is taken during the initial configuration of the device. The snapshot is then separated into critical resources, composed of those used by static modules, and resources used by the reconfigurable area, which may be deallocated. Once an evolutionary run has completed, the resources used by mutated connections are deallocated and become available for use. The available resources can now be determined by subtracting the allocated resources from the total resources of the device. These unallocated resources are available to the routing algorithm.

A suitable algorithm implemented in the FPGA routes the new connections, such as [2]. The algorithm makes the new routes using only the unallocated resources. Normally a constraint like this would make it excessively difficult for the algorithm to find acceptable routes in a timely manner or at all. However, because self-evolving hardware changes only a very small subset of the device, the routes should be possible even with the restricted set of resources available. The small number of changes necessary should also serve to greatly decrease the route times, as opposed to traditional methods, which reconfigure the entire devices even if only one change is made.

Using the partial reconfiguration capabilities in conjunction with the limited rerouting scheme makes runtime evolution possible. Information for loading the LUTs using partial reconfiguration is publicly available. However, information for the physical configuration of the device routing is proprietary, though in some cases it can be obtained from the vendor under a NDA agreement. Also advances in copy protection in the new generation of FPGA devices makes it much more likely that the configuration schemes will soon be made publicly available.

8 Applications

Self-evolving hardware, because of its self-contained nature, is uniquely suited to several types of applications. Systems incorporating self-evolving hardware, once deployed into the field, could use evolution for fault tolerance or run-time optimization. Traditionally designed digital modules could be combined with self-evolving hardware to produce a unique class of device capable of adapting to its users needs and behaviors. Self-evolving hardware modules could be combined in large parallel arrays to produce extremely large evolution clusters similar to [3]. When evolvable hardware proves capable of evolving complex and practical circuits, the applications are numerous.

9 Conclusion

A methodology for completely self-contained hardware evolution was presented. Schemes for evolving both the logic and connectivity from within an FPGA were illustrated. The most difficult portion of the implementation is creating the module for generating physical routing information and reconfiguring the device at run-time. Overall, self-evolving hardware presents viable solutions to the most prominent impediments to reaching the next level in complex hardware evolution.

References

[1] P.P. Chu and R.E. Jones. Design techniques of fpga based random number generator. In *Proc. of Military and Aerospace Applications of Programmable Devices and Technologies Conference*, 1999.

[2] J.M. Emmert and D. Bhatia. In *Proc. of the Eleventh Annual IEEE International ASIC Conference 1998*, pages 217–221, 1998.

[3] M. Korkin, G. Fehr, and G. Jeffery. In *Proc. of the The Second NASA/DoD Workshop on Evolvable Hardware*, 2000.

[4] P.C. Haddow and G. Tufte. An evolvable hardware FPGA for adaptive hardware. In *Proc. of the 2000 Congress on Evolutionary Computation*, pages 553–560, 2000.

[5] P.C. Haddow and G. Tufte. Bridging the genotype-phenotype mapping for digital fpgas. In *Proc. of the Third NASA/DoD Workshop on Evolvable Hardware*, pages 109–115, 2001.

[6] P.C. Haddow and G. Tufte. From here to there : future robust ehw technologies for large digital designs designs. In *Proc. of the Third NASA/DoD Workshop on Evolvable Hardware*, pages 109–115, 2001.

[7] J.H. Holland. *Adaption in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[8] D. Levi. Hereboy: a fast evolutionary algorithm. In *Proc. of the Second NASA/DoD Workshop on Evolvable Hardware*, pages 17–24, 2000.

[9] N. Macias. The pig paradigm: The design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture. In *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 175–180, 1999.

[10] N. Macias. Ring around the PIG: A parallel GA with only local interactions coupled with a self-reconfigurable hardware platform to implement an O(1) evolutionary cycle for evolvable hardware. In *1999 Congress on Evolutionary Computation*, pages 1067–1075, 1999.

[11] A. Stoica. Evolvable hardware: From on-chip circuit synthesis to evolvable space systems. In *Proc. of the 30th IEEE International Symposium on Multiple-Valued Logic*, pages 161–169, 2000.

[12] A. Thompson. *Hardware Evolution: Automatic design of electronic circuits in reconfigurable hardware by artificial evolution.* Distinguished dissertation series. Springer-Verlag, 1998.

[13] G. Tufte and P.C. Haddow. Prototyping a ga pipeline for complete hardware evolution. In *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 18–25, 1999.

[14] Xilinx, Inc. *Virtex Series Configuration Architecture User Guide*, v1.5 edition, September 2000.

[15] Xilinx, Inc. *Virtex 2.5V Field Programmable Gate Arrays*, July 2001.