Designing digital circuits for FPGAs using parallel genetic algorithms (WIP)

Rizwan A. Ashraf, Francis Luna, Damian Dechev and Ronald F. DeMara
Department of Electrical Engineering and Computer Science University of Central Florida
Orlando, FL, USA - 32816 - 2362
rizwan.ashraf@knights.ucf.edu, francis.luna@knights.ucf.edu, dechev@eecs.ucf.edu, demara@ucf.edu

Keywords: FPGA, Circuit Synthesis, Parallel Genetic Algorithms, Evolvable Hardware, Multicore Computing

Abstract

Multicore processors are becoming common whereas current genetic algorithm-based implementation techniques for synthesizing Field Programmable Gate Array (FPGA) circuits do not fully exploit this hardware trend. Genetic Algorithm (GA) based techniques are known to optimize multiple objectives, and automate the process of digital circuit design. In this paper, parallel GA algorithms are proposed for the synthesis of digital circuits for LUT-based FPGA architectures. Parallel modes of the GA such as Master-Slave and the Island model are compared to see which scheme results in better speedup and quicker convergence for effective utilization of current multicore hardware. Speedup of about five over the sequential single-threaded implementation is achieved with both the schemes on a six-core machine. Convergence is also found in fewer number of generations. The methods described here-in can be employed in Evolvable Hardware Systems as well as FPGA CAD tools.

1. INTRODUCTION

The realm of developing electronic circuits via techniques based on evolutionary principles such as Genetic Algorithms is referred to as *Evolutionary Electronics* [22]. Electronic circuits such as amplifiers, analog and digital filters, digital circuits (e.g. combinatorial arithmetic circuits, parity circuits, sequential circuits, etc.) have been synthesized using such algorithms [1],[11],[22]. An evolutionary algorithm based design approach is an excellent tool for optimizing human-generated designs or synthesizing designs which meet multiple objectives of power constraints, size constraints in terms of number of gates required, and timing constraints in terms of circuit delay [6],[10]. Essentially it automates the process of developing electronic circuits and is characterized by its ability to search complex solution space. This field, characterized by the use of reconfiguration techniques for hardware based on Genetic Algorithms, is known as *Evolvable Hardware* [21]. Adaptive systems can be realized using such techniques, which can adjust based on dynamics of the operating environment e.g. tolerate a failure by self-configuring a fault-tolerant design.

Reconfigurable hardware such as FPGAs is an excellent platform for the application of the above mentioned techniques. An FPGA can be used to implement any given digital circuit and it can be reconfigured in runtime by using its dynamic reconfiguration feature [20], thus serving as an ideal platform for Evolvable Hardware systems. Its architecture is composed of reconfigurable logic and interconnect elements. The reconfigurable logic elements are based on SRAM-based Lookup Tables (LUTs). Adaptive systems based on this platform can utilize on-chip PowerPCs or use dedicated implementation in hardware to efficiently implement the genetic algorithm [5],[4]. In such systems, the performance of the genetic algorithm matters in terms of the time required to converge to a solution. The current trend in computing is pushing towards the use of multicore technology, which we intend to exploit in our implementation. Also with the introduction of FPGAs which tightly integrate on-chip multicore processor with the reconfigurable logic [2] - there is a need to effectively utilize these parallel processing units for the above mentioned systems. Further, with the introduction of commodity multicore processors, the proposed technique can be effectively utilized for implementation in VLSI CAD tools [3].

A Genetic Algorithm (GA) mimics evolutionary principles by maintaining multiple candidate solutions in the form of a *population*. Each candidate solution, referred to as an *individual*, describes a potential FPGA configuration and is initially generated purely randomly. The representation of an individual is shown in figure 1 and is adopted from [14]. A fixed number of LUTs are selected for a design configuration depending on the complexity of the desired digital function. Each individual is ranked and assigned a *fitness* value, which is used to quantify its correctness. It can be calculated by some aggregate property of output, or by exhaustively applying all the possible input combinations and comparing the output of the individual with the desired output as described via the truth table of the digital function to be implemented on the FPGA. The genetic algorithm performs the operations of *crossover* and *mutation* on individuals according to user-specified probabilities, with the intent to increase their fitness values. After application of these operators, the population for the next generation (iteration) is selected based on a *selection* scheme designed to guarantee the ”survival of the fittest” i.e. the most fit individuals make it to the next generation.

Tournament-based selection is chosen for this implementation, as described later. A constant-size population is maintained in this work *(finite-population GA)*. This process continues over multiple generations until an individual is found with the required *threshold fitness* or maximum number of generations t_{max} are achieved according to the described *exit criteria*. The sequential genetic algorithm is summarized in Algorithm 1.

Algorithm 1 Genetic Algorithm

```
1: $t := 0$;
2: *Initialize* Population $P(0)$;
3: **repeat**
4:     Apply GA Operators *(Mutation, Crossover)* $P^t(t)$;
5:     *Fitness Evaluation* $P^t(t)$;
6:     Create new Population via *Tournament-based Selection* $P(t+1)$;
7:     $t := t+1$;
8: **until** Exit criteria *(max Fitness achieved OR max Number of Generations* $t_{max}$ *reached)*
```

Genetic algorithm based applications can be effectively parallelized to achieve significant speedups and to better utilize parallel processing units. Various attempts have been made to classify different models of parallel genetic algorithms [15],[4]. According to [15], genetic algorithms can be parallelized based on how fitness is evaluated, whether single or multiple populations are used, and whether GA operations like crossover and selection are performed locally in a sub-population or globally in the entire population. The authors also suggest that parallel genetic algorithms may perform better in terms of finding the solution as compared to their sequential counterparts which may get trapped in the sub-optimal portion of the search space. Thus, they have a better probability of getting out of this local optima as multiple sub-populations will tend to explore multiple portions of the search space. In this paper, the synchronous master-slave and island model of the parallel genetic algorithm as introduced in [15] are compared and contrasted for the proposed problem. We adopt these two models in our implementation as they are employed in most related works.

The main contribution of this work is to design digital circuits for FPGA-based architectures using parallel genetic algorithms (GAs). The GA employed involves the use of a linear representation which can be readily employed for intrinsic evolution systems such as through direct manipulation of FPGA configuration bitstream as proposed in [16].

The paper is organized as follows. Section 2 presents the related works. The sequential and parallel implementations are illustrated and contrasted in section 3. Experimental setup and the results are presented in section 4. Whereas, the conclusions of this work are presented in section 5.

2. RELATED WORK

Parallel Genetic Algorithms have been used for the design of digital [8] and analog circuits [1],[11]. They can be used to

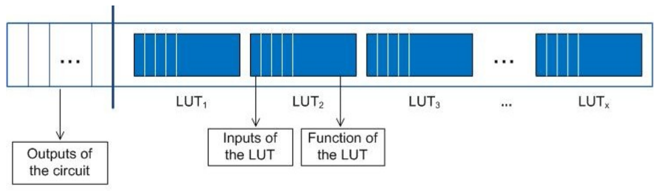

Figure 1. An individual: The representation of FPGA configuration in GA

automate the process of circuit synthesis which has useful applications in the field of evolvable hardware, where a design may need to be adapted at runtime to meet new requirements [7] or produce diverse designs which can be employed in a fault-tolerant system as proposed in [19]. In addition, GAs are also employed in VLSI layout tools for optimal placement and routing [13].

Design of analog circuits such as amplifiers and filters has been the focus of most works which employ parallel genetic algorithm [1],[11], as the fitness evaluation in this case is considered to be the most computationally expensive part. Synchronous master-slave implementation is employed in [12] and the workload of fitness evaluation is distributed off to slave nodes; fine-grain partitioning is done and a small number of configurations are evaluated by slave nodes at any given time, which potentially results in increased stall times. The experimentation for the above mentioned work is performed on a Beowulf cluster. Similarly, GA is parallelized in [1] to evolve simple VLSI circuits; coarse-grain parallelism is done using a shared memory programming model. Different implementations with a centralized single population and multiple distributed populations are done for the parallel GA in this case. The distributed scheme proposed in [1] involves infrequent communication among populations, whereas the island model proposed in our work involves communication at every generation and a possible migration is done if the sub-population has less fit individuals as compared to other competing sub-populations. Nearly linear speedups are reported for the different implementations in [1], though the results are limited and thus it is difficult to establish the generality of this work, e.g. speedups are only reported with 16 parallel computing units. Whereas, in our work, extensive experimentation with different number of threads is done, using both the synchronous master-slave and island model.

In [8], digital circuits are designed using multi-expression programming. The representation employed in [8] is a variation of linear genetic programming, whereas our work employs a representation which is targeted for FPGA-based architectures. Asynchronous island model is employed in [8], where sub-populations are maintained on parallel machines which exchange individuals after a certain defined period. The idea is to evolve multiple genetic programs in parallel on multiple processors. The computing nodes are connected in a ring topology and the Message Passing (MPI) programming model is employed for communication between different nodes. Results show a considerable decrease in computational effort as compared to the non-parallel GA. In our work, as mentioned earlier, FPGA-based architectures are targeted, and comprehensive comparison is done between the synchronous master-slave and the island model. The island model employed in our work is different as compared to [8], as the best individual in our work may be communicated to sub-populations on every generation. Further, most of the implementations have been done on clusters of computers whilst using the MPI programming model [8],[12] and look to exploit the characteristics of a group of individuals by creating sub-populations which are evolved independently for many generations (this number is generally fixed) with little or no communication. Whereas, we have proposed the parallel models of the GA on a shared memory machine for targeting today's multicore processors.

3. IMPLEMENTATION

3.1. Sequential Implementation

The following sections present a bottom-up overview of the classes and functions that form the sequential program. These parts are reviewed in this section so as to see which portions could be made parallel and to see which parts might be problematic when they are made to work in parallel. We start from the smallest unit, the Lookup Table (LUT) class, and work our way up to the largest, a Generation class. Finally, we review the main function to see how the algorithm works in this implementation.

3.1.1. LUT class

Each LUT object contains three vectors and its function type $\in \{NOT, AND, NAND, OR, NOR, XOR\}$. The vectors contain information about which LUT outputs are connected to the LUT inputs, the input binary values of these connections into the LUT, and the output value from the LUT.

3.1.2. Individual class

Each individual which represents a circuit contains four vectors, a fitness value, and various functions. The first vector is of LUTs that the circuit uses. The other three are the inputs, outputs, and connections of the circuit. The main function from this class is the CalculateFitness function which goes through each LUT and calculates each output value. It then compares these outputs with the expected value as defined by the truth table and assigns a fitness value to the individual by incrementing it's fitness, starting at zero, for every output that is correct. As an example, an individual representing a circuit with n inputs and m outputs will have a maximum fitness value of $2^n * m$ which indicates that every output line of the circuit matches the desired output for every possible input combination

3.1.3. Generation class

The generation object consists of a vector of Individuals, and an index of the Individual having the maximum fitness. It also contains the functions for the GA operations. The PerformCrossover function, according to the crossover rate (crossover probability), takes two individuals and randomly picks a crossover point, such that the boundaries of LUT objects are not violated. The LUT configurations before this point on Parent A and after this point on Parent B are copied to the offspring as shown in fig 2. If no crossover takes place, the individual is copied as is to the offspring.

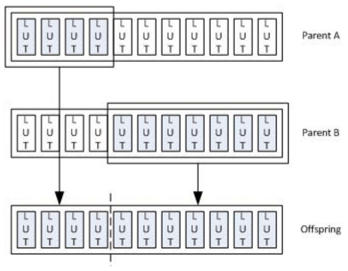

Figure 2. The application of crossover operator between two individuals.

The PerformMutation function performs three types of mutation on one individual. The first type is functionality mutation where it takes the individual and for each selected LUT based on a user-defined mutation rate, assigns a randomly chosen function. The second type is interconnection mutation where it takes each selected LUT of the individual and randomly changes its input connections. Finally, it performs output line mutation where the selected output line of the individual is randomly assigned to either any of the inputs of the circuit or any of the outputs of LUTs in the configuration.

The Selection function randomly selects k (Tournament Size) individuals from both the parent population (current generation) and the offspring population (evolved individuals). The best fit individual in that pool is picked to move on to the next generation. This is repeated until a new generation of the same size is formed.

The PerformElitism function maintains forward progress by copying the individual with maximum fitness into the next generation.

3.1.4. Main-The GA loop

The main function performs all the steps necessary for the implementation of the genetic algorithm. Firstly, individuals from the current generation are chosen to have the genetic operators performed on them based on user-defined probabilities. The crossover operation is performed on two randomly chosen individuals from the parent population. After crossover and mutation take place, the fitness for the new offspring is calculated. This is repeated until all the individuals of the population have been evolved. Then selection is performed where individuals from the parent and offspring populations are chosen to form the new population. Next, Elitism takes place. If there exists an individual with higher fitness than the elite individual, then that individual becomes the new elite and it replaces the old elite. If the highest fitness is less than the fitness of the elite individual, the elite individual is copied to the population, replacing a randomly chosen individual. This procedure is then repeated until the desired fitness level is achieved. The GA flow is outlined in fig 3.

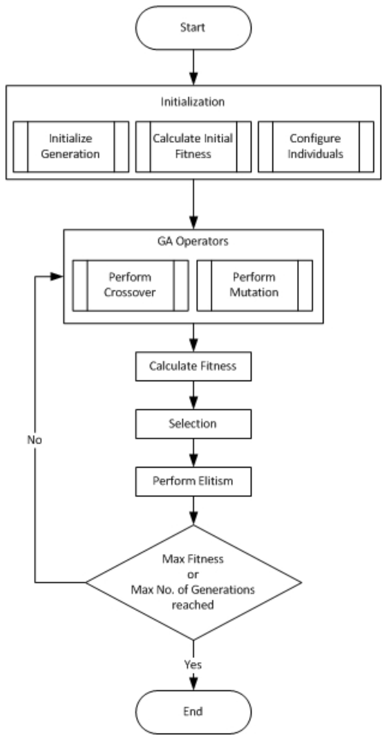

Figure 3. Genetic Algorithm flow.

3.2. Parallel Implementation

Understanding the flow of sequential program allows us to parallelize the portions of it that would be most beneficial to the overall execution time. Two different models were used in the parallel versions of the program. The first, the master-slave model, has one master thread that calls the parallel functions, and *N* worker threads that run these functions. These functions operate on one individual, but can perform genetic operators with another individual from the entire population. The second, the island model, partitions the population in smaller sub-populations at the beginning of the GA loop. The functions used operate on a smaller portion of the population. The genetic operators are limited to only use individuals from that sub-population. Each thread has a sub-population and iterates through one instance of the GA loop independently of one another. These two models were used to determine if one achieved better speedup and performance than the other.

3.2.1. Master-Slave model

The portions that are made parallel in the master-slave model are within the GA loop, where the program spends most of its execution time. More specifically the parts we parallelize are: the portion where genetic operators are performed, the Selection routine, and the portion that updates the maximum fitness. Each worker thread works on one individual before the master sends it the next individual to work on. The entire population is available to perform GA operators with.

3.2.2. Island model

The GA loop is partitioned and made to run concurrently in the island model of the program. Each island thread calls the GA operators, selection, and determines the maximum fitness on their own sub-population. If any of the islands have a higher fit individual than the elite, that individual becomes the new elite and it gets distributed to the other islands at the end of a generation.

3.2.3. Library

A library that provided the following had to be chosen for the parallel versions of the program:

- concurrent vectors
- parallel loops
- locks

Since the sequential implementation used vectors throughout its implementation we needed to find a concurrent vector implementation that multiple threads can safely access at one time. We also needed an easy construct to parallelize the portions highlighted as well as an implementation of a *lock* which can be used to find the global maximum fitness. After some research we settled on Intel Threaded Building Blocks (TBB) [18] because it contained everything needed in an easy to use library. We have used constructs of parallel_for, concurrent_vector, and spin_mutex for our parallel implementations.

The TBB concurrent_vector will replace any STL vector that will be accessed by more than one thread at a time. It guarantees that elements in the vector will never move until it is cleared which is needed in this implementation. Spin_mutex will be used to place a lock on the individual with maximum fitness. This spin_mutex version of a lock was chosen because the amount of time that the lock will be held is relatively short. The overhead for other lock implementations will not be beneficial in our design. The lock is only acquired when the new fitness of an individual is better than the current individual with maximum fitness. After the lock is acquired, it checks this condition again to ensure no other thread has changed this value since before the lock was acquired and updates the values accordingly. This method of checking whether to grab the lock first, and then checking the condition again, will greatly decrease the contention on that lock since it will only be acquired when needed as opposed to always grabbing the lock and then doing the comparison. This technique is mentioned in [9].

4. EXPERIMENTS AND RESULTS

4.1. Experimental Setup

Experiments are done to evaluate the speedup and performance of the parallel implementations as compared to the sequential single-thread implementation as used in [17]. The genetic algorithm is supposed to start-off with completely random configurations as described earlier. Further, the GA operators are applied to configurations based on user-defined values. Similarly, the selection operation is performed on a pool according to the tournament size. We keep track of timestamps throughout the program in order to compute the speedups of the parallelized portions and the entire program.

Experiments were conducted with population sizes of 120, 240, 480, and 960. The effect of population sizes on the speedups is to be observed. Each test was run 20 times for averaging due to the stochastic behavior of the algorithm. The objective of the experiments was to realize a 3-to-8 decoder configuration for a LUT-based FPGA architecture. Mutation and crossover rates were set to 0.007 and 0.60 for all the experiments. A fixed tournament size of six was used for all population size experiments. The experiments were run for a fixed number of generations (1000) to calculate the speedups. To measure GA performance, each test was run until a maximum fit individual was found and the number of generations it took to find this individual was recorded.

All tests were run on an Intel Xeon X5670 (6 cores) with 6GB RAM running Ubuntu 11.04 with Intel TBB version 3 update 5 [18]. The speedup and performance of the implementations were calculated with programs using 2, 4, 6, 8, 12 and 16 threads.

4.2. Results

4.2.1. Speedup

Speedup over the sequential version of the algorithm was achieved, however the problem is observed to not be perfectly parallelizable. Figure 4 details the speedup achieved with multiple population sizes for the master-slave model with respect to varying thread count. The highest speedup achieved by the master-slave model is 5.01 with a population size of 960 running on 6 threads. The speedup degrades after six threads because the scheduler needs to run more threads than the physical cores available (six). Figure 5 details the speedup achieved for the island model with respect to varying thread count. The highest speedup achieved by the island model is 5.04 with a population size of 960 running on 16 threads. This may be because when one island finishes an iteration of the GA loop before other islands, the other islands can be run on that physical core, resulting in better utilization. Also, larger population sizes achieve higher speedups

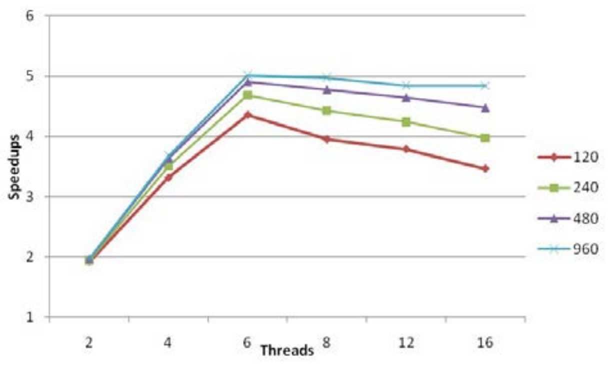

Figure 4. Speedups achieved with sync Master-slave model

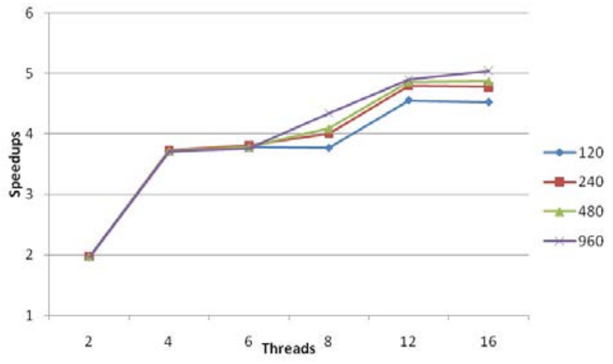

Figure 5. Speedups achieved with Island model

because there is more parallel computation than smaller population sizes, i.e. the parallel loops run for more iterations.

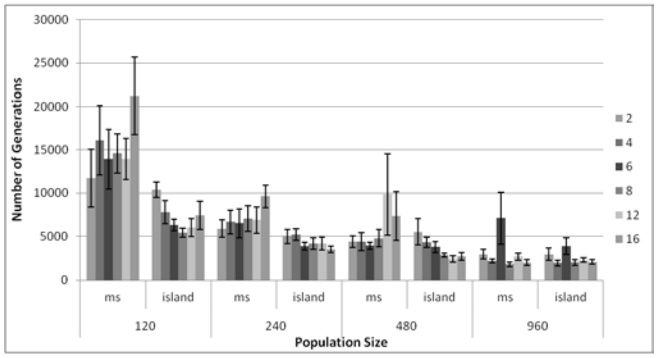

Figure 6. Comparison of Convergence Properties of Master-Slave and Island Models

4.2.2. GA Performance

Both models converged to find a completely fit individual in all the runs within the maximum number of generations set as threshold for the experiments. The mean number of generations required to find a solution over 20 experiments for different population sizes and number of threads for both master-slave and island models are shown in figure 6. For small population sizes, the island model converged in fewer generations than the master-slave model. In general, the selection pressure in case of island model is different as compared to the master-slave model. The individuals in master-slave model have to compete against the entire population as opposed to the island model where individuals compete within their own sub-populations.

5. CONCLUSION AND FUTURE WORK

Two models for parallelizing the genetic algorithm used for realizing configurations for LUT-based FPGA architectures were successfully realized. Results indicate speedups of approximately five are achieved for both the parallel modes of the genetic algorithm on a machine with six physical cores. In addition to achieving speedup over the sequential implementation, it was observed that using the island model for parallelizing this problem actually allowed the genetic algorithm to converge and find a maximum fit individual in fewer generations than the master-slave model. This results in the genetic algorithm running for less time.

Other areas of the algorithm could be explored in order to optimize them and make the parallel performance better. One such area is the fitness calculation which is inherently sequential, and could be improved by evaluating independent test vectors in parallel. Various parameter settings might also be modified to improve GA convergence properties.

REFERENCES

[1] M. Davis, L. Liu, and J. Elias. VLSI circuit synthesis using a parallel genetic algorithm. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 104–109. IEEE, 1994.

[2] K. DeHaven. Extensible Processing Platform Ideal Solution for a Wide Range of Embedded Systems. Technical report, Xilinx, apr. 2010.

[3] R. Drechsler. *Evolutionary algorithms for VLSI CAD*. Kluwer Academic Publishers, 1998.

[4] S. E. Eklund. A massively parallel architecture for distributed genetic algorithms. *Parallel Comput.*, 30:647–676, May 2004.

[5] P. Fernando, S. Katkoori, D. Keymeulen, R. Zebulum, and A. Stoica. Customizable FPGA ip core implementation of a general-purpose genetic algorithm engine. *Evolutionary Computation, IEEE Transactions on*, 14(1):133 –149, feb. 2010.

[6] F. Ferrandi, P. Lanzi, G. Palermo, C. Pilato, D. Sciuto, and A. Tumeo. An evolutionary approach to area-time optimization of FPGA designs. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pages 145 –152, july 2007.

[7] P. Haddow and G. Tufte. An evolvable hardware FPGA for adaptive hardware. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, volume 1, pages 553–560, 2000.

[8] F. Hadjam, C. Moraga, and M. Benmohamed. Cluster-based evolutionary design of digital circuits using all improved multi-expression programming. In *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2475–2482. ACM, 2007.

[9] M. Heinrich and M. Chaudhuri. Ocean warning: avoid drowning. *SIGARCH Comput. Archit. News*, 31:30–32, June 2003.

[10] T. Kalganova and J. Miller. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In *Evolvable Hardware, 1999. Proceedings of the First NASA/DoD Workshop on*, pages 54 –63, 1999.

[11] J. Lohn, G. Haith, S. Colombano, and D. Stassinopoulos. Towards evolving electronic circuits for autonomous space applications. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 5, pages 473–486. IEEE, 2000.

[12] J. D. Lohn, S. P. Colombano, G. L. Haith, and D. Stassinopoulos. A Parallel Genetic Algorithm for Automated Electronic Circuit Design. Technical report, NASA, 2000.

[13] P. Mazumder and E. M. Rudnick. *Genetic algorithms for VLSI design, layout & test automation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[14] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits. *Genetic Programming and Evolvable Machines*, 1:7–35.

[15] M. Nowostawski and R. Poli. Parallel genetic algorithm taxonomy. In L. C. Jain, editor, *KES*, pages 88–92. IEEE, 1999.

[16] R. Oreifej, R. Al-Haddad, H. Tan, and R. DeMara. Layered approach to intrinsic evolvable hardware using direct bitstream manipulation of Virtex II Pro devices. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 299 –304, aug. 2007.

[17] R. S. Oreifej, C. A. Sharma, and R. F. DeMara. Expediting ga-based evolution using group testing techniques for reconfigurable hardware. In *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, pages 1 –8, sept. 2006.

[18] J. Reinders. Intel threading building blocks, 2007.

[19] A. Stoica, R. Zebulum, D. Keymeulen, R. Tawel, T. Daud, and A. Thakoor. Reconfigurable VLSI architectures for evolvable hardware: from experimental field programmable transistor arrays to evolution-oriented chips. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(1):227–232, 2001.

[20] A. Upegui and E. Sanchez. Evolving hardware by dynamically reconfiguring xilinx FPGAs. In *ICES'05*, pages 56–65, 2005.

[21] X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 29(1):87 –97, feb 1999.

[22] R. S. Zebulum, M. A. C. Pacheco, and M. M. B. R. Vellasco. *Evolutionary Electronics: Automatic Design of Electronic Circuits and Systems by Genetic Algorithms*, volume 1. CRC Press, 2002.