GeneticFPGA: Evolving Stable Circuits on Mainstream FPGA Devices

Delon Levi
Xilinx Inc.
2100 Logic Drive
San Jose, Ca 95124
Delon.Levi@xilinx.com

Steven A. Guccione
Xilinx Inc.
2100 Logic Drive
San Jose, CA 95124
Steven.Guccione@xilinx.com

Abstract

GeneticFPGA *is a Java-based tool for evolving digital circuits on Xilinx XC4000EX™ and XC4000XL™ devices. Unlike other FPGA architectures popular with Evolutionary Hardware researchers, the XC4000 series architectures cannot accept arbitrary configuration data. Only a small subset of configuration bit patterns will produce operational circuits; other configuration bit patterns produce circuits which are unreliable and may even permanently damage the FPGA device.* GeneticFPGA *uses novel software techniques to produce legal circuit configurations for these devices, permitting experimentation with evolvable hardware on the larger, faster, more mainstream devices. In addition, these techniques have led to methods for evolving circuits which are neither temperature, voltage, nor silicon dependent. An 8-bit counter and several digital frequency dividers have been successfully evolved using this approach.* GeneticFPGA *uses Xilinx's* JBits™ *interface to control the generation of bitstream configuration data and the* XHWIF *portable hardware interface to communicate with a variety of commercially available FPGA-based hardware.* GeneticFPGA, JBits, *and* XHWIF *are currently being ported to the Xilinx* Virtex™ *family of devices, which will provide greatly increased reconfiguration speed and circuit density.*

1 Introduction

Evolutionary Hardware is a relatively new field that uses genetic algorithms to produce digital and analog circuits. Thompson's [1] groundbreaking work produced simple functional circuits, but these circuits operated primarily in the analog domain and were temperature, voltage, and silicon dependent. Korkin and de Garis's [4] ongoing work resolves these problems by evolving register values in pre-defined digital macros. Each of these macros, however, consumes large amounts of resources. In addition, both methodologies use proprietary hardware and non-mainstream FPGA devices which are of limited availability.

GeneticFPGA is a Java-based toolkit for evolving circuits on mainstream Xilinx XC4000EX/XL FPGAs. This software is highly portable and operates on a variety of commercial-off-the-shelf FPGA hardware platforms. In addition to using standard hardware platforms, *GeneticFPGA* uses novel techniques to evolve circuits which are temperature, voltage, and silicon independent.

2 The Algorithm Flow

As is standard in genetic algorithms, a population of circuits are produced, tested, scored, and selected based on survival of the fittest principals for reproduction and creation of the next generation. In *GeneticFPGA*, a single chromosome represents each circuit, which specifically is a 1-dimensional array of 1s and 0s in RAM. The codes on the chromosome are delineated into Configurable Logic Block (CLB) structures, the basic repeating logical unit in FPGAs, with the CLB structures further broken down into genes that define the state of Look-Up-Tables (LUTs), flip flop configurations, and input/output routing (see Figure 1). In this way, each CLB in the area undergoing evolution has a specific sub-string on the chromosome.

At the beginning of each chromosome read, a bitstream with "null" circuit is called. A null circuit has all of the interconnect disabled and all of the CLBs in a zeroed-out state. As the codes on the chromosome are read, the *JBits* configuration bitstream interface is used to instantiate the corresponding circuit structures in the configuration bitstream. After the device has been configured, all CLBs in the evolved area are fully specified. At this point, a complete digital circuit in the specified area has been created.

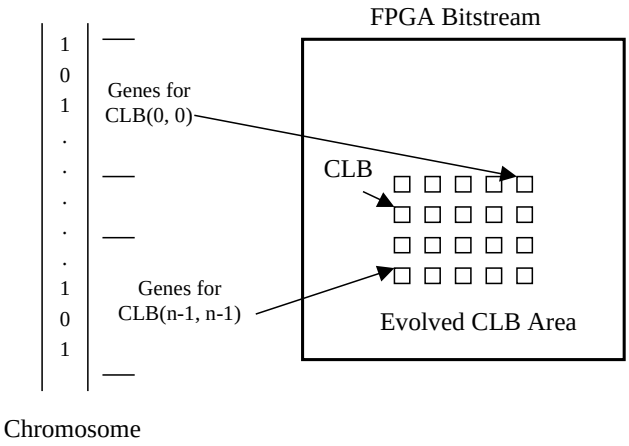

Figure 1 - CLB Gene Mapping

In addition to evolving a circuit in a region of the FPGA, it is also possible to set up other more permanent structures to facilitate testing and scoring of the evolving circuits. These test structures are typically used to supply input vectors and read results from the evolving circuit. The alternative to this test structure approach is to use dedicated device IO pins to perform this testing. Using IO pins in this manner will typically be platform-dependent. Test structures built on the FPGA device permit test vectors to be supplied and results returned via the standard configuration and readback port of the device. This allows a very large degree of platform independence.

Once all of the structures have been instantiated in the configuration bitstream, it is downloaded to an FPGA using the *XHWIF* portable hardware interface. The configured circuit is run for a specified number of clock cycles or until a specified amount of time has elapsed, and then the configuration bitstream is read back from the FPGA. The final state of the circuit is extracted from the test structures in the bitstream and is used to determine how well the circuit performed its expected function, which is then translated into a score. The score is associated with the circuit and the corresponding chromosome.

The translation of the chromosome into a configuration bitstream, the bitstream download, the bitstream readback and the scoring processes are performed on each of the chromosomes in the population as shown in Figure 2. Using fitness proportionate selection and optionally elitism, chromosomes are selected for production of next generation chromosomes. Reproduction includes one-point crossover, straight copy, copy with mutation, and elitism. Elitism is either enabled or disabled. The other schemes are selected probabilistically, according to weights the user sets at the outset.

3 The Software Architecture

Figure 3 shows a diagram of the *GeneticFPGA* software structure. The GeneticAlgorithm module provides all of the chromosome data structures and the reproductive methods

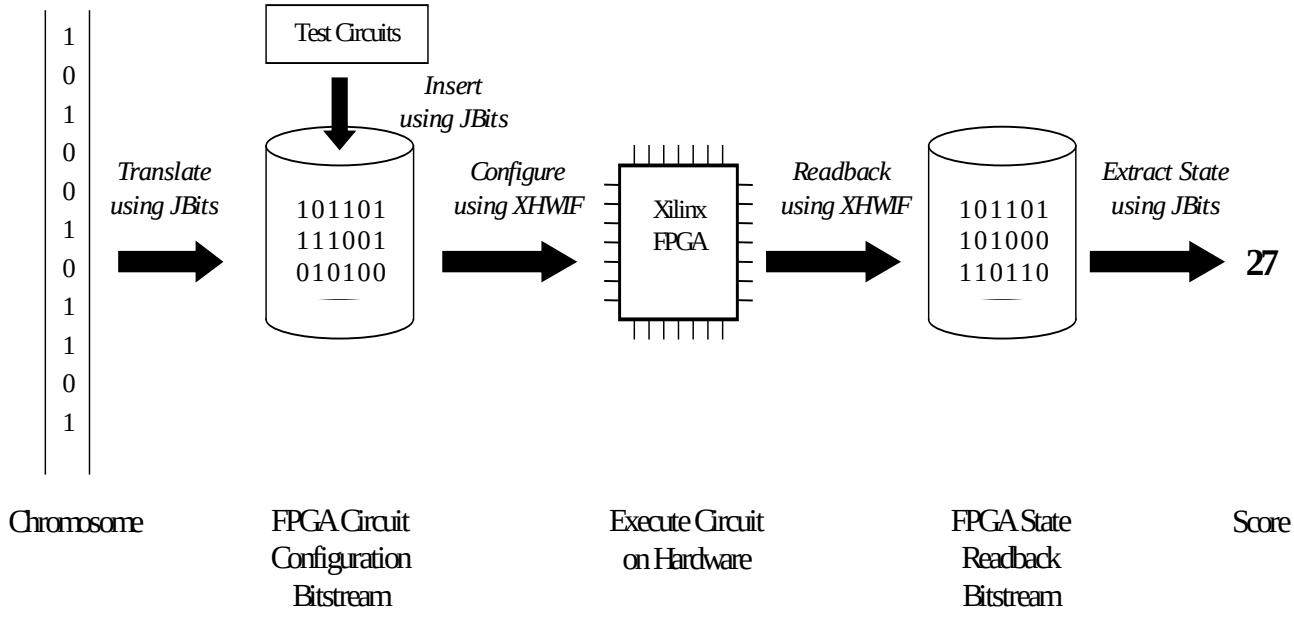

Figure 2 - The Algorithm Flow

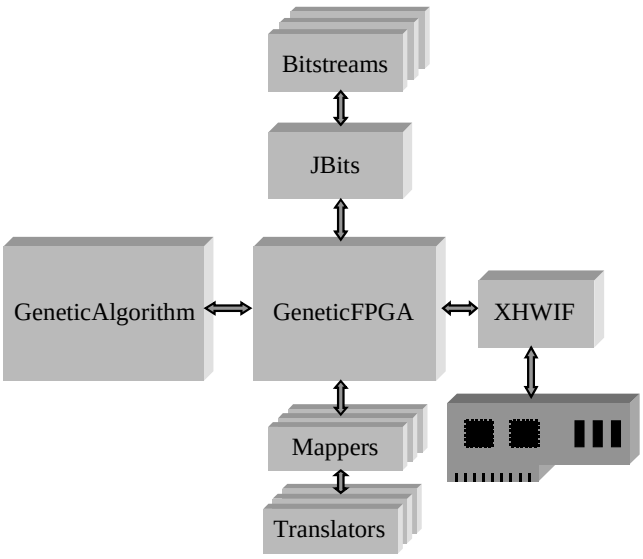

Figure 3 - Software Architecture

for creating next generation chromosomes. This component is defined generically so that it can be modified by users to solve other problems that can be mapped to a genetic algorithm.

The Mapper module determines which circuit resources are defined on the chromosome and the location of the corresponding genes on the chromosome. As the chromosome is read, the Mapper decides which circuit-gene is specified, and then calls a translator to transform that information into a circuit instantiation in the bitstream. *GeneticFPGA* comes with two mappers, one for creating digital synchronous circuits and one for creating asynchronous analog circuits. Additional mappers can be added to the system.

The Translator module is a set of functions that convert the 1s and 0s in the chromosome genes into 1s and 0s in the circuit bitstream. Each function translates a particular gene into a sub-circuit in the bitstream. *JBits* is a software interface to the Xilinx XC4000EX/XL bitstream which allows the translators in *GeneticFPGA* to specify the logic, placement, and routing of the circuit defined on the chromosome.

Finally, *XHWIF* is used as a portable software interface to various FPGA-based hardware platforms. This interface enables *GeneticFPGA* to download configuration bitstreams to live FPGAs, to advance the system clock, and to readback the final state of the circuit to score its performance. *XHWIF* also has a remote interface that allows communication with boards plugged into networked machines. The *XHWIF* interface currently supports boards from Annapolis Microsystems, MiroTech, TSI-Telesys/LavaLogic, DEC/Compaq Research, GigaOps and others.

Both the *GeneticFPGA* and the GeneticAlgorithm modules are defined as "abstract" Java classes. An abstract class indicates that some methods (functions) may be defined, but that others are left for the user to customize in a subclass. The user defines the methods in these subclasses, and *GeneticFPGA* uses the subclass to perform the specified functions.

Within the GeneticAlgorithm class the evaluate() method, which determines the scores for all the chromosomes, is left abstract for definition in a subclass to define. In defining the evaluate() function, a user specifies how the chromosomes are translated into problem specific data structures and how these data structures are evaluated for fitness. For example, the *GeneticFPGA* class translates the chromosomes into bitstream-based circuits and uses an FPGA to determine fitness. However, the GeneticAlgorithm class can also be extended by the user and tailored to solve other problems, including problems not involving evolvable hardware or FPGAs.

4 The User Interface

Within *GeneticFPGA* the stimulus(), score(byte bitstream[]), insertTest(byte bitstream[]), and output() functions are left abstract for a subclass to define. Any software instruction involving the circuit operation is included in the stimulus() method: incrementing the on-board clock, driving input vectors, etc. The score(byte bitstream[]) method returns a score for the given configuration bitstream. If the state of some external device is used to measure the performance of the circuit, then it can also be passed to the method as a byte array in place of the bitstream. The insertTest(byte bitstream[]) method is used to insert input and/or output test circuitry into the bitstream. The output() method is used to print and store metrics on the state of the population at the end of every generation.

To execute a class extended from either GeneticAlgorithm or *GeneticFPGA*, the user passes parameters like the population size, chromosome length, and number of generations to the class constructor, and then calls the start(), suspend(), resume(), or stop() methods to control the execution flow. Since these classes are threaded, multiple instantiations of the inherited classes can be made, which allows multiple populations to independently evolve in parallel. Since many FPGA-based hardware platforms use multiple FPGAs, each population can evolve on a separate device. This can improve the solution search, since multiple initial conditions and evolutionary paths may be examined at the same time.

GeneticFPGA also has the capability to dynamically relocate the CLB area undergoing evolution. At the end of each generation, the population can be moved to a new area on the same FPGA device, moved to a different device on the same board, or moved to devices on a different board. Because all parts in the XC4000EX/XL family of FPGAs use the same CLB, the relocation does not even have to be to an identical FPGA device. For example, in one generation the population can be evaluated on a XC4028EX, and on the next it can be evaluated on a XC4085XL. Of course, the test circuitry has to be relocated along with the evolving circuit, but if the test circuitry is resident on the device, the insertTest(byte bitstream[]) method easily allows this capability. One use of this feature is to permit asynchronous analog circuits to average out the effects of local silicon, voltage, and temperature irregularities.

5 Creating Stable Digital Circuits

One fundamental and seemingly unbreachable limitation has restricted the general applicability of evolutionary hardware: evolved circuits operate in the analog domain, and are voltage and temperature sensitive, in spite of being implemented on digital FPGAs. This means that the FPGA needs to operate in an environment where the temperature and voltage are finely controlled. Even then, circuits evolved on one piece of silicon are not guaranteed to work on another. In this paradigm, every circuit has to be optimized not only to solve the problem at hand, but also to various irregularities of the particular silicon device on which it is running. This behavior has made evolved circuits impractical for widespread commercial usage.

The source of this behavior is that the signals generated from the evolved circuits are highly asynchronous. Typically, the implementations employ only asynchronous logic gates. In a large complex network, the gates give rise to pulses with large time variances. Some of the pulses are shorter than the gate delays and some are shorter than the transistor switching times. If the pulse edges stream too quickly for the transistors to saturate in either high or low states, then for significant portions of time, the transistors remain in intermediate analog states.

GeneticFPGA eliminates these problems by optionally forcing the evolved circuits to use only synchronous signals. Only the CLB flip-flop outputs are used to drive other CLB inputs. All of the flip-flops have a single non-gated clock source. The inverters on the flip-flop clock source are disabled, as are the flip-flop asynchronous set and resets. Finally the flip-flop transparent latch mode is disabled. Using the FPGA device in this manner, the circuits produced operate consistently across multiple devices and behave consistently when reloaded onto the same device. Removing these synchronous constraints produces circuits much like their analog predecessors. Circuits tend to behave inconsistently across multiple devices and even when reloaded onto the same device.

Evolved circuits are analyzed using the *BoardScope* FPGA debug tool. *BoardScope* allows graphical inspection of a circuit's behavior as it is operating on the FPGA device. State changes in flip-flops are viewable in the main display, and control such as clock single stepping is also provided. It is also possible to probe individual CLBs and view the configuration, including LUT values and internal CLB

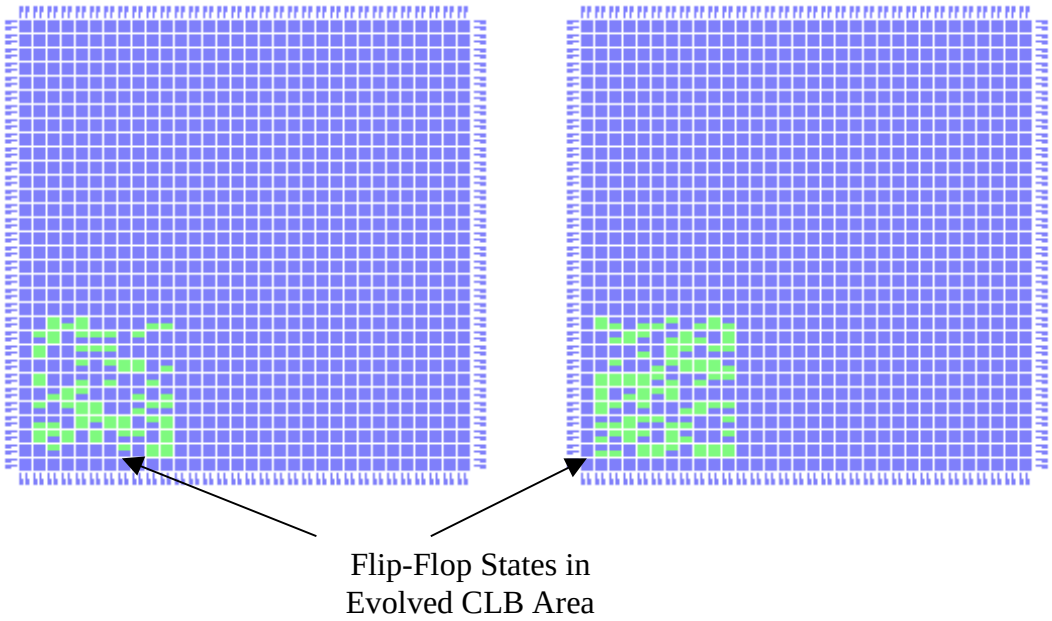

Figure 4 - Asynchronous Circuit on 2 FPGAs

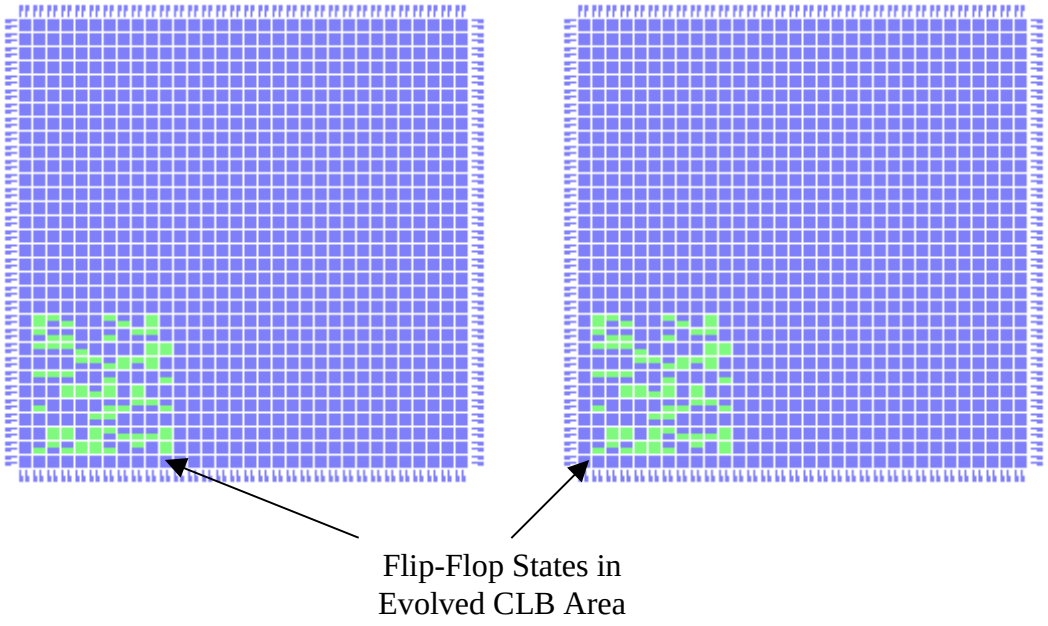

Figure 5 - Synchronous Circuit on 2 FPGAs

routing. In typical usage, the best circuit is saved to a file by *GeneticFPGA* and then downloaded and examined using *BoardScope*. For more detailed analysis, the best individuals of each generation can be examined to see how the circuits evolved. Multiple circuits can be downloaded to multiple FPGAs and their behavior compared side-by-side.

Figure 4 shows *BoardScope* viewing two circuits evolved using the standard asynchronous mode. Note the differences in the flip-flop output states. Clearly these circuits are behaving differently on different devices. Figure 5 shows the same circuit evolved using synchronous mode, again on two different devices. Note that both devices contain circuits behaving identically. It is not surprising that eliminating asynchronous feedback and using flip-flops in digital circuits enable stable circuits to evolve; these are the same techniques that engineers use to produce stable circuits manually.

As proof of concept, several circuits were evolved: 4-bit one-hot counter, 8-bit 1-hot counter, and several frequency dividers. These macros proved challenging to produce. The 8-bit counter required 15 hours of processing to evolve a correct solution and multiple tries until the correct parameters and scoring functions were found. A 16-bit 1-hot counter was attempted, but a solution was not found in a reasonable amount ot time.

An attempt at evolving a pattern recognizer was also made. A 7-bit value was fed to a 10x10 array of CLBs operating in the digital domain. The output (on the opposite side of the matrix) was supposed to present a 1 if the input had more 1s than 0s, otherwise present a 0. The correlation between the input and output remained at roughly 50%, the initial value. The nearly static 50% correlation indicated that the evolved circuits were just generating random patterns on the output, and were not generating computational data paths between the inputs and outputs.

The limited results are attributed to the exclusive use of local interconnect. Because local routing on XC4000EX/XL FPGAs have a single driver, it provided the simplest way to avoid contention. Perhaps the use of non-local routing will increase circuit evolvability and functionality.

6 Conclusions and Future Work

The *GeneticFPGA* toolkit evolves circuits at the gate and flip-flop level. By forcing the evolutionary process to follow synchronous design principles, *GeneticFPGA* may be used to create stable digital circuits. Although an FPGA architecture that accepts random configuration data at the hardware level is convenient for evolutionary hardware, *GeneticFPGA* demonstrates that this is not a requirement. Illegal and undesirable configurations can easily be avoided using software techniques. It is interesting that these techniques require only RAM-based configurable FPGA devices. Given software for producing legal circuit configurations, this approach would work on nearly any SRAM FPGA architecture, including those such as the Xilinx XC2000 - the first FPGAs developed 14 years ago.

New mainstream architectures, like the Virtex FPGA should allow more complex evolutionary circuits to evolve. The configuration and readback ports, the primary speed bottlenecks on XC4000EX/XL FPGAs, are approximately 50 times faster on Virtex architecture. In addition, the greater gate densities should allow larger evolved circuits and test circuits. The richer and more regular routing in Virtex, coupled with the use of local and global lines, should also allow greater flexibility for non-trivial circuits to arise.

References

[1] A. Thompson, "Silicon Evolution", in *Proceedings of Genetic Programming 1996 (GP96)*, J.R. Koza et al. (Eds.), pp. 444-452, MIT Press, Cambridge, Ma, 1996.

2 A. Thompson, "On the Automatic Design of Robust Electronics Through Artificial Evolution", in *Proc. 2nd Int. Conf. on Evolvable Systems: From Biology toHardware (ICES98)*, M. Sipper, D. Mange & A. Péres-Uribe (Eds.), pp13-24, Springer-Verlag,1998.

3 T.C. Fogarty, J.F. Miller, and P. Thomson., "Evolving Digital Logic Circuits on Xilinx 6000 Family FPGAs", in *Soft Computing in Engineering Design and Manufacturing*, P.K. Chawdhry, R. Roy, and E.K. Pant (Eds.), pp. 299-305. Springer-Verlag, 1998.

[4] M. Korkin, H. de Garis, F. Gers, and H. Hemmi, "CMB (CAM-Brain Machine): A Hardware Tool which Evolves a Neural Net Module in a Fraction of a Second and Runs a Million Neuron Artificial Brain in Real Time", in Proceedings of Genetic Programming 1997 (GP97), J.R. Koza et al. (Eds)., pp. 498-503, Morgan Kaufmann Publishers, San Francisco, Ca, 1997.

5 J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, Ma, 1992.

6 L. Davis, *Handbook of Genetic Algorithms*, International Thomson Computer Press, London, 1996.

7 D. E. Goldberg, *Genetic Algorithms in Search Optimization and Machine Learning*, Addison-Wesley Publishing Company, Inc., Reading, Ma, 1989.

8 S. A. Guccione and D. Levi, "XBI: A Java-based interface to FPGA hardware", in *Confiurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering, Proc. SPIE Photonics East*, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, Bellingham, WA, November 1998.

9 D. Levi and S. A. Guccione, "BoardScope: A Debug Tool for Reconfigurable Systems", in *Confiurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering, Proc. SPIE Photonics East*, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, Bellingham, WA, November 1998.

10 Xilinx, Inc., *The Programmable Logic Data Book*, 1998.

11 N. Sitkoff, M. Wazlowski, A. Smith, and H. Silverman, "Implementing a Genetic Algorithm on a Parallel Custom Computing Machine", in *IEEE Symposium on FPGAs for Custom Computing Machines*, Peter Athanas and Kenneth L. Pocek (Ed.), pp. 180-187, IEEE Computer Society Press, Los Alamitos, CA, April 1995.

12 P. Graham and B. Nelson, "Genetic Algorithms in Software and in Hardware - A Performance Analysis of Workstations and Custom Computing Machine Implementations", *in IEEE Symposium on FPGAs for Custom Computing Machines*, Kenneth L. Pocek and Jeffrey Arnold (Ed.), pp. 216-225, IEEE Computer Society Press, Los Alamitos, CA, April 1996.

13 I. M. Bland and G. M. Megson, "The Systolic Array Genetic Algorithm, an Example of Systolic Arrays as a Reconfigurable Design Methodology", in *IEEE Symposium on FPGAs for Custom Computing Machines*, Kenneth L. Pocek and Jeffrey Arnold (Eds.), pp. 260-261, IEEE Computer Society Press, Los Alamitos, CA, April 1998.

14 P. Graham and B. Nelson, "A Hardware Genetic Algorithm for the Travelling Salesman Problem on SPLASH 2" *in Field-Programmable Logic and Applications*, Will Moore and Wayne Luk (Eds.), pp. 352-361, Springer-Verlag, Berlin, August/September 1995. Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications, FPL 1995. Lecture Notes in Computer Science 975.

15 M. Mitchell, J. P. Cructhfield, and R. Das, "*Evolving Cellular Automata to Perform Computations*", in *Handbook of Evolutionary Computation*, T. Back, D. Fogel, and Z. Michelwicz (Eds.), Oxford University Press, Oxford, 1997.

16 T. Toffoli and N. Margolus, *Cellular Automata Machines: A New Environment for Modeling*, MIT Press, Cambridge, Ma, 1987.

17 R. Dawkins, *The Blind Watchmaker*, W. W. Norton & Company, New York , NY, 1996.

18 J. H. Holland, *Hidden Order*, Addison-Wesley Publishing Company, Reading, Ma, 1995.

Patents Pending