Proceedings of the 2007 IEEE Workshop on
Evolvable and Adaptive Hardware (WEAH 2007)

Evolutionary Multiobjective Optimization on a Chip

Stefano Bonissone and Raj Subbu

General Electric Global Research, One Research Circle, Niskayuna, NY 12309

bonisson@research.ge.com, subbu@research.ge.com

***Abstract*—The majority of research in Evolvable Hardware is focused on evolving logic for deployment on reconfigurable hardware. There are far fewer reports concerned with the implementation of Evolutionary Algorithms (EAs) in hardware. The focus of our research is directed toward using reconfigurable hardware as a means to speed up evolutionary search, and in particular Evolutionary Multiobjective Optimization (EMO). Evolutionary Multiobjective Optimization utilizes an evolutionary search to find solutions to difficult multiobjective optimization problems. We present an implementation of an EMO algorithm in reconfigurable hardware, and discuss how it may be utilized in practical deployment situations.**

***Index Terms*—Evolutionary Multiobjective Optimization, Evolvable Hardware, Field Programmable Gate Array, Neural Networks.**

I. Introduction

Evolutionary algorithms (EA) have been traditionally deployed on general-purpose computational systems. Parallel and distributed computing techniques have been employed on general-purpose computational systems to improve the computational efficiency of an evolutionary algorithm. Since evolutionary algorithms work with a population of solutions, parallelizing the fitness computation has the benefit of significant speedup. When the problem solving may be sped up by problem decomposition, distributed evolutionary computing techniques have been employed [1], [2]. However, for the efficient execution of applications requiring high-frequency multi-objective optimization constrained by the size of the computational unit, it is desirable to develop a multi-objective evolutionary technique that enables high optimization speed-ups with a small computational footprint. An example of such an application is in missile or unmanned vehicle control, where a high optimization speed is required, the computational hardware footprint and weight constraints are severe, and the domain demands simultaneous consideration and optimization of multiple conflicting objectives such as thrust and range given varying mission needs while operating with a finite fuel resource. Another example is in medical image reconstruction based on projections [3], where a high reconstruction speed and high quality image output are requirements. In this medical imaging domain application, the computational hardware footprint requirements are typically very severe.

Most of the Evolvable Hardware (EH) research is concerned with evolving circuits for various functions. Extrinsic EH, Intrinsic EH, and Complete Hardware Evolution (CHE) are the three principal research areas, and are categorized based on the magnitude of the algorithm computation that is performed in hardware. Extrinsic EH uses no special hardware, but instead circuit simulations for the fitness evaluations of the evolved circuits. Intrinsic EH typically applies to situations when either a simulator is not available, not accurate enough, or not practical. So, each evolved circuit configuration is physically implemented and tested in its real operational environment. Executing the individuals directly in hardware also allows for the exploitation of the physics of the devices, as Thompson [4] has shown. CHE [5] utilizes reconfigurable hardware for all aspects of the evolution, including the EA in hardware. There has been considerably less work done in this area, as one of the principal practical bottlenecks has been the speed of the fitness evaluation in practical applications when the evolutionary process is implemented in hardware. If the fitness evaluation is compute-intensive and slow, that reduces or eliminates the speedup benefits of the evolutionary operational computations in hardware. What is needed is a framework wherein fast evolutionary operational computations are interfaced with fast and reliable fitness evaluations, to realize the benefits of evolutionary computation in hardware.

Our work does not fall into either the Extrinsic or Intrinsic EH categories, but more closely follows the ideas of Complete Hardware Evolution. Our goal is not to evolve circuits, but rather to utilize a Field Programmable Gate Array (FPGA) as a means for implementing an Evolutionary Multiobjective Optimizer (EMO) in hardware. Our current goal is not the implementation of one particular EMO algorithm on hardware, but the demonstration of the deployment potential of a basic but generic EMO algorithm on hardware. An EMO in hardware would allow for much faster execution and open the door for applications in many new domains that require compact and fast real-time processing. In addition to the speedup benefit, the reduced size and power consumption will also allow for on-board real-time deployments such as in unmanned vehicles or in medical scanners.

Section II presents a brief overview of some recent and relevant EH research and shows how our research fits into the current literature. Section III describes the EMO on FPGA hardware concept, including a description of the language used, algorithm modules, and the overall algorithm design. Section IV describes the test problem used and experimental results. Also shown is a performance comparison to a corresponding software implementation of the EMO algorithm. Section V presents conclusions and future work for deployment-level demonstrations.

II. BACKGROUND

A. *Evolvable Hardware*

Research in the field of Evolvable Hardware is typically concerned with evolving circuits for reconfigurable hardware devices. Complete Hardware Evolution, as described earlier, refers to every aspect of the evolutionary approach existing and executing on a hardware device. While we are not concerned with evolving circuits, we chose to use an FPGA as a means to speed up the evolutionary search. There has been less research done in this area.

Tufte and Haddow [5] implemented a complete hardware evolution scheme in an attempt to evolve logic solutions directly on the hardware device. Kramer et al. [6] implemented different versions of a Compact Genetic Algorithm (CGA) [7] to compare performance on a series of benchmark problems. CGAs have a very compact representation and small memory footprint. Rather than attempting to maintain an entire population of individuals, the CGA methods represent a population as a bit probability chromosome. A probability exists for each bit in the chromosome, which represents the likelihood of a 1 residing in that bit for the entire population. Members of the population are created on the fly utilizing this distribution of bits. This representation of the population makes these algorithms attractive for hardware implementation. So and Wu [8] implemented a four-step genetic search algorithm for use in video processing. Hamid and Marshall [9] have implemented a Genetic Algorithm (GA) in FPGA hardware for grey-scale soft morphological filters (SMF). Their motivation was one of speeding up the search algorithm, which would otherwise take considerably longer to run. Perkins et al. [10] present another instance of implementing all components of an EA in hardware. In this work, an EA is implemented in hardware to design a stack filter that alters a corrupted signal in an attempt to reconstruct the original signal as closely as possible. A speedup in execution was the main motivation for these authors to implement such an EA in hardware.

Scott et al. [11] appear to be one of the first to implement a GA in hardware. In their proof of concept, the VHDL language was used to implement Goldberg's Simple Genetic Algorithm (SGA) in an FPGA. Several simple test functions were used to compare the hardware implementation and a software implementation of the same algorithm. Glette and Torrensen [12] also attempted to perform on board intrinsic evolution of logic. Slightly different from Tufte and Haddow's approach [5], Glette and Torrensen utilized the PPC microprocessor on the Xilinx FPGA board to run the GA, while configuring the FPGA logic to evaluate the individuals of the population. Similar to Scott et al. [11], Glette and Torrensen utilize an SGA style algorithm and representation for the search process.

There are a few important characteristics that unify all of these above on-chip EAs. The most important unifying characteristic is the binary representation. A binary representation was used as an encoding for a configuration bit stream in some implementations, and in others it was used to represent the value of a variable. There are a few exceptions where logic elements were evolved, and even an instance of a hardware-based Genetic Program (GP) evolving a function and terminal set [13].

What has not been attempted thus far was to create a hardware implementation of a real-valued EA, and in particular a real-valued multiobjective EA. In software implementations, real-valued EAs show better performance across many types of optimization problems than compatible binary encoded EAs.

B. *Evolutionary Multiobjective Optimization*

Many interesting and important real world problems cannot be well characterized by a single measure of fitness. These problems require multiple, sometimes conflicting, measures of fitness that need to be optimized simultaneously. Often when solving these problems, the multiple criteria are condensed into a single function to be optimized. These singleton functions can then be minimized or maximized using a classical GA. Condensing multiple objectives results in a loss of information and is often inappropriate, subsequently leading to a reduced or inaccurate space of solutions. By leaving the objectives separate, an EA can optimize solutions for all objectives and identify those that lie on or close to the Pareto frontier. Non-domination can be used as a comparison measure between two candidate solutions. Using the definition of non-dominance defined in (1), *a dominates b* given a minimization problem.

$$\begin{aligned} \forall i \in \{1,\ldots,n\} &: f_i(a) \le f_i(b) \;\wedge \\ \exists j \in \{1,\ldots,n\} &: f_j(a) < f_j(b) \end{aligned} \tag{1}$$

In most EMO algorithms, dominance is the criteria used for the selection function since a single fitness value cannot describe the quality of a solution. The NSGA-II algorithm that Deb et al. proposed in [14] sorts the population of solutions first based on dominance, and then again based on crowding within a neighborhood. This non-dominated sorting selection method ensures that the best solutions based on all fitness functions advance to the next generation. The secondary sort based on crowding is an attempt to reduce the gaps in the non-dominated front that the algorithm produces. The set of non-dominated solutions from the final population is what the NSGA-II algorithm returns. Two other EMO algorithms in the literature that incorporate non-domination as part of their selection or archival functions are the SPEA2 [15] and PAES [16]. The use of dominance as a selector is a principal component of our algorithm implementation on hardware. Our current implementation is however restricted to encapsulating the basic characteristics of an EMO, and does not seek to implement some of the more intensive computations of these above referenced algorithms.

III. Evolutionary Multiobjective Optimization on Field Programmable Gate Arrays

A. *Implementation Language*

Many of the program implementations in FPGA hardware are designed in VHDL, a hardware description language. There has been one published implementation of Genetic Programming (GP) in FPGA hardware created by Martin [13] programmed in a higher-level language called Handel-C. Handel-C is a subset of the ANSI-C language with special constructs for parallelism and FPGA hardware access.

There are aspects of the Handel-C language that make it possible to exploit the inherent parallelism of reconfigurable devices. The language has implicit timing whereby each assignment takes 1 clock cycle to execute. This allows for a quick and simple analysis of the algorithm during the design process. The parallel construct of the language allows each operation and assignment in that block of code to be executed in the same clock cycle. The compiler/synthesizer creates all the necessary logic in order to perform these operations in parallel.

B. *Representation*

The structure that is chosen to represent candidate solutions is perhaps the most important feature of an EA. The method of representation determines the type of random variation operators to be implemented and affects the manner in which the fitness landscape is traversed. The fitness landscape is also dependent on the representation scheme based on the decoding procedure used to evaluate an individual. Binary representations of real valued numbers tend to perform worse for EAs. It is for this reason that a real valued representation is considered. Many EAs when solving for a function that requires several real valued variables encode the genome as a series of real valued numbers. The binary alternative is to encode all numbers as a lengthy bit string. This bit string representation is unwieldy, increases the search time, and can even prevent the EA from converging to a good solution.

Solutions to many multiobjective optimization problems can be represented using a real valued chromosome. However, real valued numbers have certain limitations when represented in hardware. The precision of the real valued numbers is the main concern, as more bits are required for a given precision. Utilizing floating point numbers is avoided due to the complex logic necessary for their implementation. This increased logical complexity would also be the limiting factor for the overall clock speed of the device. The alternative to floating point numbers is fixed point numbers. Fixed point requires much less logic, but is unable to represent solutions with as much precision. If the problem requires a large range of numbers, more bits can be used to represent the integer portion. If on the other hand more fractional precision is necessary, more bits can be utilized in the fractional portion of the numbers. This is a problem dependent portion of the EMO algorithm, in the same way the fitness function formulation is problem dependent.

C. *Algorithm Modules*

In order to implement an EMO algorithm in hardware, one must take into consideration how to construct different modules. A random number generator and non-dominance filter are two necessary modules. The random number generator must be able to produce both uniformly distributed random numbers and Gaussian distributed random numbers for use in mutation. A simple but elegant way of generating uniformly distributed random numbers is to use several Cellular Automata (CA). Many researchers choose to use Logical Feedback Shift Registers (LFSRs) for their implementations within a small logic footprint. An LFSR however, does not produce very good random numbers—the numbers generated become predictable and sequential. Martin [17] performed a comparison of different random number generators taking into account simplicity in implementation and randomness of numbers generated. His analysis determined that although LFSRs are simple to implement in hardware, they do not produce “random enough” numbers for evolutionary search. A single cellular automaton also yields similar performance, but several CAs or several LFSRs produced much better random numbers and improved the evolutionary search. To extract a random number from a bank of CAs, one composes the number by taking a single bit from each CA. For example, if there are 10 CAs of 16 bits each, then 16 different 10 bit numbers are generated. The advantage of using multiple CAs is that each cell’s new value can be computed independently of any other cell’s new value. This parallel computation allows for one to generate many random numbers in a single clock cycle given sufficient logic.

Apart from generating uniformly distributed random numbers, Gaussian distributed random numbers are also desirable for mutation. Many techniques to generate Gaussian distributed random numbers utilize multiple uniformly distributed random numbers that are additively combined based on the *Central Limit Theorem*. Software techniques that utilize two uniform random numbers to generate one or more Gaussian distributed random numbers require the use of trigonometric functions and the square root function. The implementation of trigonometric functions in an FPGA requires a large logic footprint, and reduces the overall clock speed of the final design. It is for this reason that we choose to use a different approach. Since the bank of CAs generate many uniformly distributed random numbers each clock cycle, we exploit the *Central Limit Theorem* to combine them into a single Gaussian distributed random number. Four numbers from the bank of CAs are summed together and divided by four in order to obtain a number that is of approximate Gaussian distribution. Four numbers are utilized because a division by four can occur with very little logic by shifting by two bits. Further, utilizing the Handel-C language constructs specific to FPGA hardware, we are able to reduce the cycle cost of averaging four numbers to a single clock cycle.

The second major module necessary for an EMO implementation is a dominance/non-dominance filter. A simple test for non-dominance has a computational complexity of $O(n^2)$ in a sequential programming language. In the worst case, each individual needs to be compared to every other individual in order to establish non-dominance. Fortunately, this is one area that can be greatly reduced in complexity through parallelization. The non-dominance filter compares each individual in parallel to every other individual. The individuals that remain are the non-dominated individuals of the population. This filtering process takes place on the parent and child populations so that the non-dominated solutions from a generation are stored in the archive. This non-dominated set is what comprises the new population for the subsequent generation.

Crossover is implemented using an arithmetic crossover scheme with a α value of 0.5, which allows for very simple implementation in device logic. The general definition of arithmetic crossover is given in (2) below.

$$\begin{aligned} c_1 &= \alpha * p_1 + (1-\alpha) * p_2 \\ c_2 &= (1-\alpha) * p_1 + \alpha * p_2 \end{aligned} \tag{2}$$

With $\alpha = 0.5$, each parent is divided by two, and then summed. These two operations can be completed in two cycles, with the entire crossover operation occurring in 5 cycles. Utilizing an α value of 0.5 produces a single individual instead of two distinct individuals from crossover. This fits well into the non-dominated selection scheme that is utilized in the EMO algorithm.

The mutation of individuals is performed through a Gaussian mutation procedure. The selected individual to be mutated is multiplied by a Gaussian random number to introduce a perturbation in value. Unfortunately, multiplication is expensive in hardware, so parallelism is avoided for this aspect of the EMO and the logically expensive multiplier is not duplicated many times in the hardware. Fortunately, fixed precision is used for individuals so the logic for an even more complex and expensive floating-point multiplier is avoided.

D. *Algorithm Overview*

The overall basic EMO algorithm is depicted in Figure 1 below. The algorithm incorporates an archive that stores the non-dominated solutions from the parent and solution populations from each generation. The archive size is fixed due to implementation limits, and so once the archive stores its last individual, the next non-dominated solution to be included in the archive replaces the first individual in the archive. The reason for the archival replacement is one of implementation on the FPGA hardware. To create an arbitrarily large archive is much more difficult and most likely not worth the resulting significant slowdown. This replacement continues in a sequential manner. This archival strategy also retains the most recently found individuals, which are known to generally dominate the individuals found earlier in the evolutionary search.

The multi CA random number generator depicted in Figure 1 constantly generates new bit strings each cycle independently of the rest of the EMO. These bit strings are accessible from the different modules since they reside in RAM on the device. This allows the crossover and mutation operations access to randomly generated numbers for use in their operation.

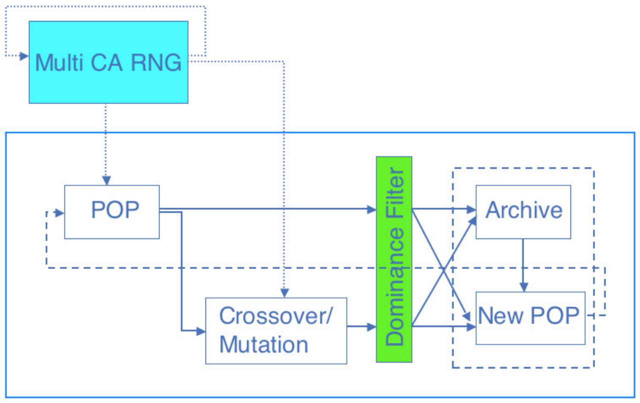

Figure 1: Architecture of the basic EMO algorithm

IV. Experimental Results

To test the EMO on a chip concept, a simple multiobjective optimization problem is used as a benchmark. At the time of this research, we did not have access to FPGA hardware to synthesize the algorithm, so we were limited to running it in a realistic simulation environment. We were constrained mostly by the complexity of the test problem because of the speed of the simulator compilation. Due to this simulation constraint, we chose to use Schaffer's test function [18]. It is a two-criteria test function that requires a single search variable to be constrained to a specified range. The problem definition is given in (3) with the overall search space depicted in Figure 2 below.

$$f_1(x) = x^2$$
$$f_2(x) = (x-2)^2 \qquad (3)$$
$$-8 \le x \le 8$$

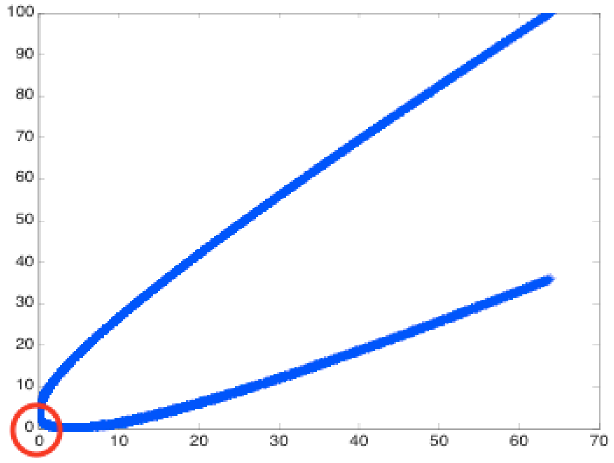

Figure 2: Schaffer's two-objective one-variable function, with the Pareto frontier shown circled.

The EMO design takes a relatively small percentage of the logic and memory in the Xilinx family of FPGA devices. This suggests that a much more complex fitness function could be utilized and the hardware EMO could be used to tackle a real world problem. The EMO design takes 10519 logic cells and 145Kb of distributed RAM. Based on an FPGA specification sheet for the Xilinx Virtex-4 family of devices [19], this design would fit on the second smallest member of the Virtex-4 family, namely the XC4VLX25 device. Should a more complex fitness evaluator be necessary, a device with more logic would most likely meet the requirements. On the largest device of the family, the XC4VLX200, the logic blocks of the EMO take up only 5% of the device and only 10% of the total distributed memory. At the time of the research, this was the top of the line that Xilinx offered. However, now the Virtex-5 family exists which allows access to even greater resources at possibly a faster clock speed.

Table 1: Performance benchmarking of the EMO on a Chip versus a high-level language implementation.

	Software	FPGA
Population	100	100
Resolution	N/A	16 bits
Archive size	500	500
CA size	N/A	400
Clock (MHz)	3,000	41
Cycles	N/A	42,203
Time (sec)	0.3381	0.001029
Speedup	1	328.46

In order to benchmark the hardware EMO, it is compared against an equivalent Matlab version of the algorithm applied to the same test problem. This comparison serves as a measure of the speed benefit of the hardware implementation versus a software implementation. Though the EMO is implemented in a hardware simulator, the simulator accurately tracks clock cycles, and the corresponding run-time in a hardware deployment may be calculated. Table 1 summarizes the differences between the two implementations.

All algorithm parameters are kept the same across the two implementations in order to allow for a fair comparison. Only the resolution of variables differed—a floating-point representation in Matlab and fixed-point representation in hardware. The population size, number of generations and archive size were all made smaller for the simulation. This was to reduce the compilation time for the hardware simulator. Though the hardware implementation utilizes a much lower clock speed compared to the software implementation, the hardware implementation is able to leverage the parallelization very well. The execution time in seconds for the FPGA EMO is calculated by measuring the number of cycles from the simulator's cycle counter. This cycle count is divided by the maximum clock speed that the deployment "place and route" algorithm determined that the EMO design could run at. Based on this calculated execution time and measured execution time for the software version, there is a speedup of over 300 times.

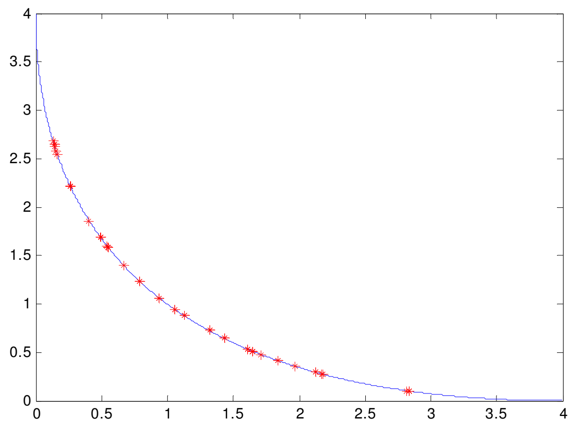

Figure 3: EMO generated Pareto frontier shown using star symbols superimposed over the mathematically defined Pareto frontier.

The EMO converged well to the Pareto frontier for this particular problem, something that is certainly desirable. Figure 2 shows the overall search space for the test problem with the location of the Pareto front circled. Figure 3 shows the mathematically defined Pareto front as a line and solutions in the archive as superimposed star symbols for a given run of the EMO. Although much larger designs would easily fit onto the FPGA hardware, the simulator required simple designs in order to be able to compile in a reasonable amount of time. The compilation time for the simulation of a population size of 10 individuals, 32 generations and an archive size of 64 took over 2 hours.

V. Discussion and Conclusions

Our research is directed toward using reconfigurable hardware as a means to speed up evolutionary search, and in particular Evolutionary Multiobjective Optimization. Evolutionary Multiobjective Optimization utilizes an evolutionary search to find solutions to difficult multiobjective optimization problems. We presented an implementation of a basic EMO algorithm in reconfigurable FPGA hardware.

One of the principal practical bottlenecks in the hardware implementation of evolutionary algorithms has been the speed of the fitness evaluation when the evolutionary process is implemented on hardware. If the fitness evaluation is compute-intensive and slow, that reduces or eliminates the speedup benefits of the evolutionary operational computations performed in hardware. In this paper, we outlined an initial method and framework to implement an EMO on an FPGA device, and demonstrated its potential for the efficient execution in applications requiring high-frequency multi-objective optimization constrained by the size of the computational unit. However, what is needed is a flexible representation for a fitness function that can accommodate efficient hardware implementation and simultaneously be utilized to represent arbitrary fitness function models. What is also needed is a fitness function representation that is flexible enough to rapidly adapt should the model of the domain change. Implementing the fitness functions as a set of neural network models has these beneficial characteristics, and further the combination of an EMO with an onboard bank of neural network models may be used to successfully model and optimize complex dynamic systems [20].

There is much work that can be done to elaborate on this hardware EMO implementation. Most importantly, synthesizing the simulated design is the next immediate step. The cycle accurate simulation and analysis of the compiled design show that it is possible to implement an EMO in an FPGA. Synthesis onto hardware will allow for large population and archival sizes to be used. It will also allow for the EMO to tackle harder, more interesting problems. Our current implementation demonstration is restricted to encapsulating the basic characteristics of an EMO, and did not seek to implement some of the more intensive computations such as fitness sharing in algorithms such as the NSGA-II [14]. This is a topic for future work.

References

[1] R. Subbu and A. C. Sanderson. "Modeling and Convergence Analysis of Distributed Coevolutionary Algorithms", *IEEE Transactions on Systems, Man, and Cybernetics (Part-B),* 34(2), 2004.

[2] R. Subbu and A. C. Sanderson, "Network-Based Distributed Planning using Coevolutionary Agents: Architecture and Evaluation," *IEEE Transactions on Systems, Man, and Cybernetics (Part-A),* 34(2), 2004.

[3] X. Li, T. Jiang, and D. J. Evans, "Medical Image Reconstruction Using a Multi-objective Genetic Local Search Algorithm," *International Journal on Computer Mathematics,* 74:301—314, 2000.

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

[5] G. Tufte and P. Haddow, "Prototyping a GA pipeline for complete hardware evolution," In A. Stoica, D. Keymeulen, and J. Lohn, editors, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, IEEE Computer Society, 1999.

[6] G. R. Kramer, J. C. Gallagher, and M. Raymer, "On the Relative Efficiencies of *cGA variants for intrinsic Evolvable Hardware: Population, Mutation, and Random Immigrants, *Proceedings of the 2004 NASA/DoD Conference on Evolution Hardware*, 2004.

[7] G. Harik, F. Lobo, and D. Goldberg, "The compact genetic algorithm," *IEEE Transactions on Evolutionary Computation*, 3(4), 1999.

[8] M. So, and A. Wu, "FPGA Implementation of Four-Step Genetic Search Algorithm," *Proceedings of the 6th IEEE International Conference on Electronics, Circuits and Systems*, 1999.

[9] M. Hamid, and S. Marshall, "FPGA Realisation of the Genetic Algorithm for the Design of Grey-Scale Soft Morphological Filters," *Proceedings of the International Conference on Visual Information Engineering*, 2003.

[10] S. Perkins, R. Porter, and N. Harvey, "Everything on the chip: a hardware-based self-contained spatially-structured genetic algorithm for signal processing," In J. Miller, A. Thompson, P. Thomson, and T. Fogarty, editors, *Proceedings of the 3rd International Conference on Evolvable Systems: From Biology to Hardware*, 2000.

[11] S. D. Scott, S. Sharad, and S. Ashok. *A hardware engine for genetic algorithms*, Technical Report UNL-CSE-97-001, University of Nebraska-Lincoln, 1997.

[12] K. Glette and J. Torresen, "A Flexible On-Chip Evolution System Implemented on a Xilinx Virtex-II Pro Device," *Proceedings of Sixth International Conference on Evolvable Hardware*, Springer LNCS 3637, 2005.

[13] P. Martin, "A Hardware Implementation of a Genetic Programming System using FPGAs and Handel-C," *Genetic Programming and Evolvable Machines*, 2(4):317—343, 2001.

[14] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, *A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II*, KanGAL Technical Report 200001, Indian Institute of Technology, Kanpur, India, 2000.

[15] E. Zitzler, M. Laumanns, and L. Thiele, *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*, Technical Report 103, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, May 2001.

[16] J. D. Knowles, and D. W. Corne, "The Pareto Archived Evolution Strategy: A New Baseline Algorithm for Pareto Multiobjective Optimisation," *Proceedings of the Congress on Evolutionary Computing*, Washington DC, 1999.

[17] P. Martin, *An Analysis of Random Number Generator for a Hardware Implementation of Genetic Programming using FPGAs and Handel-C*, www.celoxica.com/techlib/files/CELW0307171J2F -23.pdf, 2002.

[18] J. D. Schaffer, "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms," *In Genetic Algorithms and their Applications: Proceedings of the First International Conference on Genetic Algorithms*, Lawrence Erlbaum, 1985.

[19] Xilinx, Inc. *Xilinx Virtex-4 specification sheet*, http://direct.xilinx.com/bvdocs/publications/ds112.pdf.

[20] R. Subbu, P. Bonissone, N. Eklund, W. Yan, N. Iyer, F. Xue, and R. Shah, "Management of Complex Dynamic Systems based on Model-predictive Multi-objective Optimization," *Proceedings of the 2006 IEEE International Conference on Computational Intelligence for Measurement Systems and Applications*, 2006.