High-speed FPGA-based Implementations of a Genetic Algorithm

Michalis Vavouras Kyprianos Papadimitriou Ioannis Papaefstathiou
Department of Electronic and Computer Engineering
Technical University of Crete
GR73100 Chania, Crete, Greece
{vavouras, kpapadim, ygp}@mhl.tuc.gr

***Abstract*—One very promising approach for solving complex optimizing and search problems is the Genetic Algorithm (GA) one. Based on this scheme a population of abstract representations of candidate solutions to an optimization problem gradually evolves toward better solutions. The aim is the optimization of a given function, the so called fitness function, which is evaluated upon the initial population as well as upon the solutions after successive generations. In this paper, we present the design of a GA and its implementation on state-of-the-art FPGAs. Our approach optimizes significantly more fitness functions than any other proposed solution. Several experiments on a platform with a Virtex-II Pro FPGA have been conducted. Implementations on a number of different high-end FPGAs outperforms other reconfigurable systems with a speedup ranging from 1.2x to 96.5x.**

I. Introduction

Genetic Algorithms (GAs) are search optimization techniques based on Darwin's theory about evolution. They form a particular class of evolutionary algorithms that utilize techniques inspired by evolutionary biology such as inheritance, selection, crossover and mutation. Their main advantage is that they can solve more complex problems, in contrast to other heuristic algorithms, because of their strong characteristics which are mainly the following [1]: (a) they search starting from a population of points and not from a single point. If one path turns out to be a dead end, they eliminate it and continue working on more promising avenues, giving them a greater chance of finding the optimal solution, (b) they use the inherited fitness function information and not other auxiliary knowledge. Instead of using previously known domain-specific information when taking decisions and making changes, they make random changes to their candidate solutions and then use the fitness function to determine whether those changes produce an improvement or not, (c) they use probabilistic transition rules and not deterministic ones, and (d) they work with the coding of the parameter set and not the parameters themselves.

The motivation for implementing GAs in hardware stems from the fact that they are very CPU intensive while they are also intrinsically parallel algorithms. Moreover, the basic operations of a GA can be executed in a pipelining fashion. In addition, the pipelined datapath can be replicated in a way that many datapaths can process different population members, in parallel.

A hardware-based genetic algorithm (HGA) implemented on an FPGA board was first published in 1995 [2]. This design, the so called original HGA, was implemented in a board with eight FPGAs. In present work we implemented the HGA in the XUPV2P platform. In order to create a flexible and low-cost design we have utilized the embedded PowerPC of this FPGA as well as built-in special-purpose cores like multipliers. We fine tuned the architecture of the HGA so as to be more parallel, and then we parameterized it and added more complex fitness functions. The resulted architecture was implemented in a number of high-end FPGAs. Real-world experiments carried out on a fully functional prototype demonstrate that our system outperforms any existing or proposed solution.

Our architecture has been extended in order to be flexible so as to support a wide range of different genetic parameters. We did this because there exist a few promising publications from the Genetic Algorithms field proposing the examination of the GA behavior when utilizing different fitness functions, and examination of the GA performance when altering the population size [3]. In this paper we explore the GA's behavior for different types of functions by conducting different experiments and introduce the ability to vary the genetic parameters. The main contributions of our approach can be summarized in the following:

- our system has significantly higher performance compared to other hardware systems,
- it supports more fitness functions than any existing system, and it also supports parameterized values for the population size, the member width and the fitness value width. Thus a flexible design is available, more than any other existing design allowing to explore the impact of genetic parameters in the GA performance,
- our real-world experiments and results cover a different viewpoint regarding the efficiency of a reconfigurable hardware system implementing GAs

The paper is structured as follows: In Section II the basics of the genetic algorithms along with the existing hardware designs are presented. Section III describes the original HGA [2], and our initial implementation on a XUPV2P platform. Section IV presents our optimizations. Section V has implementation details regarding the resources utilization and the execution frequency when the optimized system was implemented in a number of high-end FPGAs. Section VI contains the experimental results and comparisons with existing and proposed reconfigurable systems. Finally, Section VII discusses the present status and the future directions of this work.

TABLE I
INITIAL POPULATION AND FITNESS VALUES BEFORE THE GA OPERATION

	i	Chromosome x_i (in bin)	$f(x_i) = 3x$	$f(x_i) \div \sum f(x_i)$	Expected Count $f(x_i) \div \overline{f(x_i)}$	Actual Count
	1	11010	78	0.412	1.65	2
	2	01101	39	0.206	0.82	1
	3	00010	6	0.031	0.12	0
	4	10110	66	0.35	1.39	1
$\sum f(x_i)$			189	1	4.00	4
$\overline{f(x_i)}$			47.25	0.25	1.00	1
max $f(x_i)$			78	0.412	1.65	2

TABLE II
THE POPULATION AND THE FITNESS VALUES AFTER ONE GENERATION

	i	Chromosome x_i (in bin) After Selection	Mate	Crossover Point	After Crossover	$f(x_i) = 3x$
	1	11/010	x_3	2	11101	87
	2	1/1010	x_4	1	10110	66
	3	01/101	x_1	2	01010	30
	4	1/0110	x_2	1	11010	78
$\sum f(x_i)$						261
$\overline{f(x_i)}$						65.25
max $f(x_i)$						87

II. RELATED WORK

In this section we introduce the basics of genetic algorithms and we present the most efficient existing hardware designs; some of those systems have been implemented in real hardware whereas other have been simulated.

A. Genetic algorithm operation

A GA is implemented as a computer simulation in which a population of abstract representations (called chromosomes) of candidate solutions (called individuals or phenotypes) to an optimization problem gradually evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The basic operations of a GA are selection, crossover, and mutation. Initially, a population of candidate solutions of the problem is randomly generated. A *fitness function* is then applied on each member of the population and produces a fitness value. The members are evaluated according to their fitness values and the most promising ones are *selected* as parents. The 'fittest' member has the more chances to be selected as a parent. Then the following process is repeated until a certain termination criterion is met: The selected members, i.e. parents, are randomly mated via *crossover*. This means that they exchange part of their body. Then, *mutations* are performed to preserve population diversity in order to avoid convergence to local optima, and the new individuals called offsprings are generated. Afterwards, the fitness value of each offspring is evaluated which affects the selection process for the next generation; in every step the worst part of the population is replaced with the newly generated offsprings. The termination criterion might be a maximum number of generations, or, a satisfactory fitness value. If m is the population size, and g is the number of generations, a typical GA would execute each of its operations $m \times g$ times. If the termination criterion is a predefined number of generations, a satisfactory solution may or may not be reached. Recapitulating, the GA operation involves random number generation, copying, and partial string exchange and lies on the survival of the fittest when searching for optima. The use of a population of points helps the GA to avoid converging to false peaks (local optima) in the search space.

The exact GA operation is better illustrated with the example shown in Tables I and II. First, an initial population of four chromosomes each encoded with a string of 5 bits is given. The values in the other cells of the Table are decimal. The fitness function to be optimized is $f(x) = 3x$. The scope is to maximize the fitness function over the domain $0 \leq x \leq 31$. Table I has the initial population along with the corresponding fitness values and percentages as derived from the fitness function. The "$f(x_i) \div \sum f(x_i)$" column contains the probability of the chromosomes selection. The "Actual Count" column has the selection results, which correspond with the values of the "Expected Count" column.

After selecting the chromosomes, these are randomly paired and each pair is examined separately. For each pair, the GA decides whether to perform crossover or not. If it does not, then both chromosomes are placed into the population with possible mutations. If it does, then a random crossover point is selected and the crossover task proceeds. Table II illustrates the situation after the completion of the GA operation. It is observed that the selection process left out the chromosome with the lowest probability, i.e. x_3 of Table I. In our example, we have assumed that the selected chromosomes to be randomly paired are A=11010 and B=01101. After two point crossover, i.e. 11/010 and 01/101, they become A'=11101 and B'=01010. These bit strings are then placed in the population with possible mutations. The GA operation invokes the mutation operation on the new bit strings very rarely, i.e. with a low probability, generating a random number for each bit and flipping this bit only if the random number is less than or equal to the mutation probability. In the present example, the mutation operation has not affected the bit strings.

As shown in Table II, after the selection, crossover, and mutation are complete, the new strings are placed in a new

population. In this example, just after only one generation, the sum of fitness values $\sum f(x_i)$ was increased from 189 to 261 and the average fitness value $\overline{f(x_i)}$ was increased from 47.25 to 65.25, which gives for both an increase of 38%; the maximum fitness value max $f(x_i)$ was increased from 78 to 87 which gives an increase of 11%. These metrics can be monitored throughout GA execution as the quality criteria of the population after each generation. The above operation continues until the termination criterion is met.

B. Hardware implementations of GAs

Scott et al. [2] published one of the first works on hardware-based implementation of a genetic algorithm. It forms the basis of our work and we describe it in detail in the next Section.

Koonar et al. [4] designed a genetic algorithm for circuit partitioning in VLSI physical design automation. All GA operations are carried out by the hardware. The fitness function to be optimized performs a combination of bit-wise AND and OR operations. The synthesis results reported a maximum frequency of 123 MHz for a Xilinx Virtex FPGA. However, no actual hardware was implemented and simulations were conducted at 50 MHz.

Martin et al. [5] implemented a GA that evaluates populations for fitness functions that correspond to the regression problem and to a certain boolean logic problem. The former problem uses the fitness function $x = 2a + b$ with integer values. The boolean logic problem has the fitness function $x = a \oplus b$. The authors experimented with different designs utilizing different numbers of parallel fitness evaluation modules. The scope was to find the optimal degree of parallelism, and thus evolution speed, when the silicon cost was taken into account. The design was implemented on a Xilinx Virtex FPGA running at 25 MHz.

Glette et al. [6] designed a system for image recognition using a genetic algorithm. Only the fitness function is implemented in hardware, whereas all other GA operations are executed on an embedded PowerPC. The fitness function to be optimized is the summation $\sum x$. The design was implemented in a XUPV2P platform. The synthesis reported 131 MHz; however, due to the clock of the platform the hardware executed at 100 MHz and the PowerPC at 300 MHz.

All the above designs conform to the simple GA style described in [1]. Also, they all contain adequate experimental results which allows for comparisons with our work, and they experimented with moderate population sizes as we did at the present phase. However, none of them has examined the convergence of the different functions to optimums when the number of generations changes, which is explored in the present work. With respect to the generality of the architecture our work is the first to incorporate six fitness functions and variable population size, member width and fitness value width.

III. The Initial System

Our design is based on the HGA system first presented in [7], [2].

A. The original HGA

The HGA system consists of two parts, a software application and the hardware design. The software running on a PC, generates the initial population along with the genetic parameters and writes them to a memory located on the hardware platform. The hardware is the HGA core which reads the contents of the memory, executes the GA and writes the results back to the same memory for the user to view.

Using the software application the user controls the following genetic parameters: the population size m, the sum of fitnesses of the initial population $\sum f(x_i)$, the maximum number of generations g, a seed for the pseudorandom generator, the crossover probability and the mutation probability. More specifically, the user enters the population size, the fitness function to be optimized, the maximum number of generations - which is the termination criterion -, and the crossover and mutation probabilities. Then, the software produces randomly the initial population, calculates the sum of fitnesses of the initial population according to the given fitness function, and generates a seed with a pseudorandom number generator. Moreover, the software using the fitness function calculates the fitness value of each member. Then, it concatenates each member with its fitness value and forms a 'quantity'. The above are sent to the SRAM of the prototyping board through the PCI interface. A "Go" signal from the software indicates that the data have been written in the SRAM. The HGA core then starts execution, and when done it activates a "Done" signal indicating to the software that the final population has been generated and written in the SRAM. The software reads the SRAM contents, and writes them into a file for the user to view.

The HGA core comprises the following modules shown in Figure 1: the Memory Interface Module (MIM), the Population Sequencer (PS), the Selection Module (SM), the Random Number Generator (RNG), the Crossover and Mutation Module (CMM) and the Fitness Module (FM). The HGA modules have been designed according to the GA operators described in [1]. They operate concurrently and they form a coarse-grained pipeline. The Shared Memory (MEM) is the external SRAM which is directly accessed by the software application and by the HGA core. After the genetic parameters and the initial population with the fitness values have been loaded into the shared memory, a "Go" signal notifies the memory interface module that the GA can start execution. In turn, the MIM notifies the population sequencer, the pseudorandom number generator, the fitness module, and the crossover and mutation modules that they should start execution. Each of these modules requests its parameters from the MIM, which fetches them from the appropriate locations of the MEM. The PS starts the pipeline by requesting population members from the MIM and passing them to the SM. The latter receives a member and decides whether it will be selected according to a selection algorithm, called roulette wheel selection algorithm [7]. More specifically, the SM compares the fitness value of each member with a random number. If a member is selected, then another member is examined until a pair of sufficiently fit members is found. Then the pair of members is passed to the CMM, and at the same time, the SM resets itself and restarts the selection process. The CMM decides whether to apply crossover and mutation based on random number values sent from the RNG. Then, the new members are sent to the FM for evaluation by the fitness function. When done, the two new members are written to the MEM through the MIM.

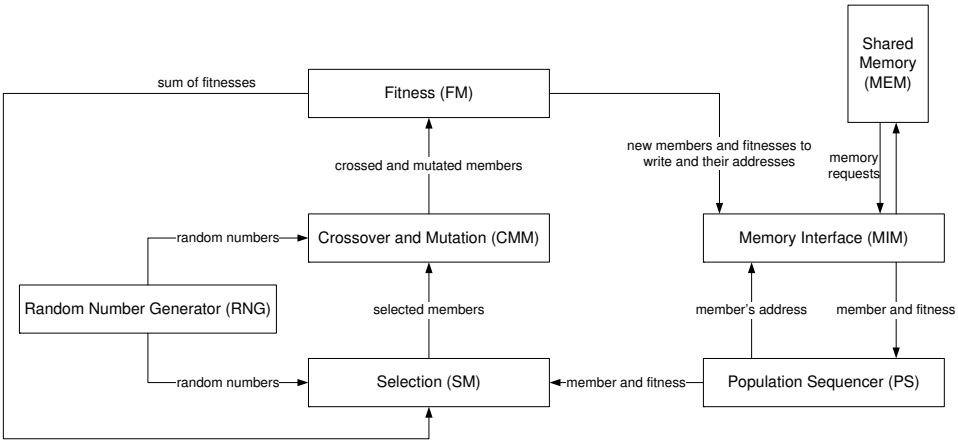

Fig. 1. Block diagram of the original HGA system.

The FM also keeps some records regarding the current state of the HGA that are used by the SM to select new members. The above steps are repeated until the FM determines that the current HGA run is completed. When that happens the FM notifies the MIM, which in turn shuts down the HGA modules and activates the "Done" signal.

The original design was implemented in a board with eight Xilinx FPGAs. Due to capacity limitations the member's width was $n = 3$ bit and only the function $f(x) = 2x$ was optimized. The width of the fitness value produced by the fitness function was $f = 4$ bits, which accommodates the maximum fitness value of a member. The experiments were conducted for a population size of $m = 16$ members, and simulation were reported for $n = 4$, $f = 5$ and $m = 32$. Finally, although the synthesis result reported a frequency of 12 MHz, the prototype operated at 2 MHz clock due to the wire-wrapping connection between the FPGAs.

B. Porting the design to the XUPV2P platform

In the original implementation the software communicated with the hardware through the PCI interface. In our case, this is performed using the embedded PowerPC which provides ready-to-use routines. Moreover, the external SRAM has been replaced by a dual port memory implemented with FPGA Block RAMs (BRAMs). Thus the Shared Memory is now considered as part of the HGA core. Its size is 128 entries x 9 bits. Its depth accommodates the populations of two successive generations, i.e. $2 \times 32 = 64$, and the six genetic parameters. More specifically, two distinct parts of the memory locations have been assigned to two successive generations. During an active generation, the new population members are written in the memory part that contains the oldest members; it should be noted that only the members of the previous generation are needed during an active generation. An appropriate memory width was selected such as to accommodate the concatenation of the members with their fitness values, i.e. n=4 and f=5 respectively.

Our initial HGA system and some improvements have been introduced in [8]. It operates as follows: When starting, only the initial population and the genetic parameters are sent from the PC to the PowerPC; those parameters are not at all processed by the PC, they are just entered by the user. Those values are written to the memory, i.e. MEM, which is implemented with BRAM resources. Then, the PowerPC activates a "Go" signal indicating to the HGA core to start execution. After the GA execution completion, the HGA core sends a "Done" signal to the PowerPC indicating that the final population is available in the MEM. The PowerPC reads the MEM contents and it sends them to the host PC for the user to view. The system enters an awaiting state which allows for the user to send new data. It should be noted that the initial population and the genetic parameters could also be sent to the PowerPC by other means (e.g. a simple keyboard attached to the platform) making our system a fully embedded one. The only reason for using an external PC was due to a Graphical User Interface to enter the initial population and the genetic parameters and getting the resulting data.

The communication between the PowerPC and the HGA core is performed through the OPB bus; it is infrequent, i.e. happens only at the initial and the final stages of system execution, and thus it is conducted through memory mapped registers. A signal activation controller, implemented in hardware, stands between the PowerPC and the HGA core for supervising their communication. This module undertakes the task of holding a signal activated, which is driven from a register written by the PowerPC, only for the necessary number of cycles. For example, the disabling of the clock enable signal when memory port access is not required, provides the best technique to minimize the embedded memory dynamic power consumption as described in [9]. This operation is depicted in Figure 2.

We verified the correct operation by using the $\overline{f(x_i)}$ of the final population as the validation criterion. This is also used as a criterion for optimization. We observed that for the same fitness function, the resulted values after different generations were almost equal to the ones published for the original HGA. At this stage, only the $f(x) = 2x$ was included, and the design was not parameterized.

IV. The New HGA System

In the new system the type of fitness function to be optimized is given as an input parameter from the user to the

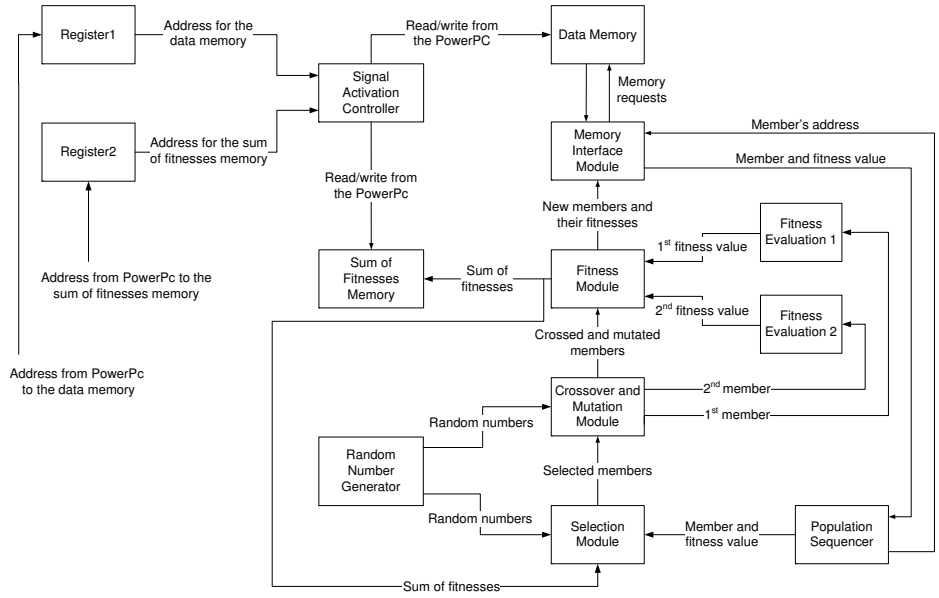

Fig. 2. Block diagram of the new HGA system

HGA core. All fitness functions are implemented in the HGA core and their outputs are connected to a multiplexer. The multiplexer outputs the result of the appropriate fitness function which is controlled with a selector driven by the above parameter. The following six fitness functions are supported:

- $f(x) = 2x$
- $f(x) = x$
- $f(x) = x + 5$
- $f(x) = x^2$
- $f(x) = 2x^3 - 45x^2 + 300x$
- $f(x) = x^3 - 15x^2 + 500$

In order to increase the performance of our design, we implemented them using the built-in resources of the FPGA. This is the first version of the improved HGA denoted as HGAv1. As we gradually improve the design and we plan to explore the HGA performance for various parameters [3], in the next version we have parameterized the population size, the member width and the fitness value width. In this system, the so called P-HGAv1 (P stands for parameterized), we have increased the depth of the Shared Memory (MEM) so as to support large population sizes. Specifically, its depth has been changed to 4096 so as to accommodate a population of up to 2044 members, i.e. $2 \times 2044 = 4088$, plus the six genetic parameters. In addition, we support larger members and fitness values, i.e. we increased the member width to $n = 10$ and the fitness value to $f = 36$. This affected other parameters, such as the widths of the shared memory and the sum of fitnesses. The former's width was changed from 9-bits to 46-bits, so as to accommodate the member and its fitness value. Consequently, the width of the sum of fitnesses was changed to 47-bits.

The block diagram of the P-HGAv1 design is shown in Figure 2. The HDL code was significantly changed to make it parameterized, and to increase the parallelism. A new module has been added, the fitness evaluation module, which consists of built-in multiplier blocks and adders, a controller and a multiplexer. The multipliers are implemented in a pipeline manner. Thus the fitness value of each member is calculated in more than one cycle. In the original design, the calculation was done in a single clock cycle but for one simple function only. In present system each multiplier indicates with a *RDY* signal when its result is ready. The function controller handles the *RDY* signals and activates a signal when the appropriate fitness value is ready. Then, the function multiplexer drives the appropriate fitness function to the output. Hence, the fitness value of a member is calculated in more than one cycles, and it finishes when the *RDY* signal is activated. The elapsed time between a multiplier's initiation and assertion of its *RDY* signal varies depending on the fitness function.

As the operation is applied on pairs of members, we decided to instantiate two components of the fitness evaluation module. Thus when the CMM outputs the two new members, i.e. the offsprings, it sends them to the FM module and the two fitness evaluation modules. More specifically, both are sent to the FM, and in parallel each of them is sent to one fitness evaluation module. This allows for parallel evaluation of the two members as soon as they are given by the CMM. In the original design the two new members were evaluated by the FM in a serial manner.

In order to increase the transparency during the experiments, we added a dual port memory for storing the sum of fitnesses of each population after successive generations; the sum of fitnesses serves as a quality criterion for the generated populations. After the algorithm's completion, the memory contents are given to the PowerPC and then to the PC for the user to view. The results of Section VI were collected using this memory.

V. Implementation Details

Table III has the area costs of the initial and the optimized HGA systems covering the BRAMs, built-in multiplier blocks, slices, flip-flops and LUTs utilization. For both systems, the HGA core is clocked with the 100 MHz clock of the OPB bus,

TABLE III
INITIAL AND PARAMETERIZED HGA CORE RESOURCES UTILIZATION OF THE VIRTEX-II PRO

Resources	available	HGA Porting	P-HGAv1
# MULT18x18	136	1 (1%)	11 (8%)
# BRAMs	136	4 (3%)	5 (3%)
# Slices	13,696	830 (6%)	1,193 (8%)
# Flip Flops	27,392	792 (3%)	1,140 (4%)
# 4-In LUTs	27,392	1,520 (5%)	2,055 (7%)

while the PowerPC is clocked at 300 MHz. The development, was conducted within the Xilinx EDK and ISE ver. 9.1 tools.

The initial implementation occupied 6% of the FPGA slices and one built-in multiplier block. The latter is not occupied by the function $f(x) = 2x$; this function is implemented with logic resources and not built-in blocks so as to create a design which is identical to the original one. The built-in multiplier block is used within the SM module for multiplying the sum of fitnesses with a random value; the low clock rate achieved when this multiplier was implemented with CLBs forced as to use a built-in multiplier. Finally, the BRAM resources implement the main memory of the PowerPC and the shared memory.

In the optimized P-HGAv1, the fitness functions occupy 8% of the built-in multiplier blocks. One more BRAM was added for storing the sum of fitnesses. The slices increased to 8% mainly due to the control for the fitness evaluation modules and the sum of fitnesses memory.

The above illustrate the differences in the implementation between the initial HGA design, and our optimized P-HGAv1, both as implemented in the Virtex-II Pro FPGA. It would have not been reasonable to compare any new implementations with the original HGA [2] due to the huge differences in the technology since then. A first remark about the P-HGAv1, as it has low silicon requirements while supporting multiple fitness functions and it is parameterized, is that it can perfectly fit into larger applications/systems in which the genetic algorithms are used for optimization, like scheduling and image recognition systems [6].

During the development of the fitness evaluation module we experimented with different configuration for the multipliers implementation using the Xilinx Core Generator. The parameters to be configured concern the depth of the pipelining, and whether the input and the output of the multiplier are registered. In Table IV, the checked cells correspond to the selected configurations for the generation of each multiplier. Each of them was incorporated into the P-HGAv1 design, and the results after place and route are shown in the Table. Based on these results, we created all fitness functions using the configuration that produces the fastest multipliers. We believe that Table IV clearly demonstrates the effect of the designer's decisions on the actual performance of such a system.

As reported in Section IV we continuously evolve the architecture and at present we have incorporated six fitness functions, parameterized the population size and increased the member and the fitness value widths. Moreover, we experienced with newer FPGAs of similar volume with the Virtex-II Pro, which just illustrates the benefits when implementing the design in a newer FPGA technology. The results were produced after the placement and routing and are shown in Tables V and VI. The algorithm operates at 136 MHz and 160 MHz, when targets the Virtex-4 and the Virtex-5 respectively.

TABLE V
P-HGAV1 CORE RESOURCES UTILIZATION OF THE VIRTEX-4

Resources	Available	P-HGAv1
# DSP48s	128	10 (7%)
# BRAMs	232	5 (2%)
# Slices	25,280	1,223 (4%)
# Flip Flops	50,560	1,139 (2%)
# 4-In LUTs	50,560	2,276 (4%)

TABLE VI
P-HGAV1 CORE RESOURCES UTILIZATION OF THE VIRTEX-5

Resources	Available	P-HGAv1
# DSP48Es	48	10 (20%)
# BRAMs	60	3 (5%)
# Slices	7,200	760 (10%)
# Flip Flops	28,800	1,124 (3%)
# 6-In LUTs	28,800	1,714 (5%)

VI. EXPERIMENTAL RESULTS AND COMPARISON

In this Section we present real-world results, and a performance comparison with the existing hardware approaches discussed in Section II.

A. Experimentation with different parameters

In our experiments we evaluated the optimization of the fitness functions according to the number of generations. This type of experiments is different from the ones conducted for the original HGA [2], but we believe it demonstrates in a more concrete way the efficiency of our approach. The fixed parameters were $m = 32$, $n = 4$, and $f = 14$. Six different functions were optimized and the P-HGAv1 ran for 1 up to 100 generations. Each experiment was repeated for several initial populations that were randomly generated. The sum of fitnesses serves as a quality criterion for the populations produced after successive generations. The results are depicted in Figure 3 in logarithmic scale.

As observed in the Figure all the fitness functions are optimized, meaning that better solutions are found, just after a few generations. This is due to the small population size and the member's width. However, the function $f(x) = x^3 - 15x^2 + 500$ is optimized in less generations than the other ones, i.e. it converges faster. This occurs as the specific function has two optimums, 0 and 15. The rest of the fitness functions converge in about 20 number of generations. In Figure 3, an arrow pointing to the series of each fitness function indicates the number of generations after which each function converges.

B. Comparison with other hardware implementations

In order to accurately examine the performance of the P-HGAv1 we inserted a module that counts the clock cycles for each generation. The I/O timings were removed because we are interested only in the execution time of the P-HGAv1. This takes approximately $0.021msecs$ for each generation, and it growths almost linearly with the number of generations. This

TABLE IV
THE EFFECT OF PARAMETERS DURING MULTIPLIER CONFIGURATION ON THE P-HGAV1 DESIGN

i	Maximum Pipeline	Minimum Pipeline	Register Input	Register Output	Output Latency	Maximum P-HGAv1 Frequency(MHz)
1	√				1	101
2	√		√	√	2	102
3	√		√		1	102
4	√			√	1	108
5		√			0	60
6		√	√	√	2	94
7		√	√		1	98
8		√		√	1	99

TABLE VII
COMPARISON BETWEEN HARDWARE IMPLEMENTATIONS OF SIMPLE GAS

Ref.	Year	Device	MHz	m	g	Execution time (ms)	Execution time (ms) for g=1,000
[5]	2001	Xilinx XCV2000	25	16	511	14	27
[6]	2007	Xilinx XC2VP30	100	16	1,000	1,313	1,313
[4]	2002	Xilinx XCV50	50	20	100	7.2	72
P-HGAv1	2009	Xilinx XC2VP30	100	32	100	2.19	21.9

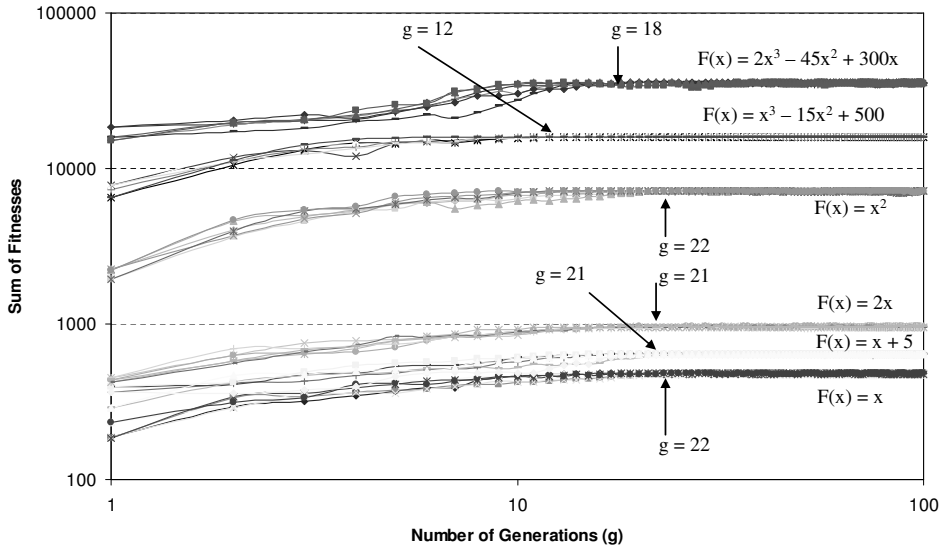

Fig. 3. Fitness functions optimization and convergence over the number of generations

TABLE VIII
SPEEDUP OF THE P-HGAV1 IMPLEMENTED IN V-II PRO OVER OTHER HARDWARE DESIGNS (ACTUAL IMPLEMENTATION)

Ref.	V-II Pro
[5]	x1.2
[6]	x60.0
[4]	x3.3

TABLE IX
EXECUTION OF THE P-HGAV1 IN HIGH-END FPGAS. NO ACTUAL IMPLEMENTATION YET

Device	MHz	m	g	Execution time (ms)
Xilinx XC2VP30	107	32	1,000	20.4
Xilinx XC4VFX60	136	32	1,000	16.1
Xilinx XC5VLX50T	160	32	1,000	13.6

linearity was observed in [7] as well. In our case, we observed that the above time varies slightly for different fitness functions and initial populations.

Tables VII and VIII have the comparison of the P-HGAv1 implementation in Xilinx Virtex-II Pro with other hardware approaches targeting small population sizes. Although the existing approaches optimize different fitness functions, we

TABLE X
SPEEDUP OF THE P-HGAV1 DESIGN IN HIGH-END FPGAS OVER OTHER HARDWARE DESIGNS (PAR RESULTS)

Ref.	V-II Pro	V-4	V-5
[5]	x1.3	x1.6	x2.0
[6]	x64.3	x81.5	x96.5
[4]	x3.5	x4.4	x5.3

believe that this has a minor impact in the comparison results as the module that executes the calculation of the fitness function, does not consume much of the total execution time; no matter if the fitness evaluation is a combination of bit-wise AND and OR [4], XOR [5], summation [6], or polynomial [2], we saw that it does not consume a large portion of the total execution time. Specifically, in our case during one generation which requires approximately 2100 cycles, 160 cycles are consumed for the heavier fitness function when calculating the fitness values for 32 population members; this entails 8% of the total execution time. Finally, the execution time is normalized for $g = 1,000$ which is easily done due to its linear relation with the generation counts.

The contents of Table VII have been sorted according to the population size m. It is clear that the P-HGAv1 is significantly faster when compared with the existing approaches. It is also worth noticing that the populations supported by the existing systems are smaller than the one supported by our system. Since, as it is widely stated, the smaller the population size, the faster the execution time, it is clear that our system would be even faster for population sizes of $m = 16$ or 20. It should also be stressed that the devices used for all those designs are very similar, if not identical, to the one in which we have implemented the P-HGAv1. Therefore, we strongly believe that the comparisons are fair.

Another interesting observation concerns the implementation in [6] which uses the same XUPV2P platform as well. In that work only the fitness function module is implemented in hardware, whereas all other operations are carried out by the PowerPC. This incurs significant delay to the algorithm execution. The frequency in Table VII refers to the hardware part, whereas the Power PC operates at 300 MHz.

As Table VIII shows, our system achieves a speedup of 1.2 up to 60 over the existing solutions, while it also supports larger population sizes, member and fitness value widths, and more fitness functions. We believe that the P-HGAv1 is one of the most promising approaches for systems implementing GAs. In addition, Tables IX and X have the performance improvements when the design targets the Virtex-4 and Virtex-5 FPGAs. They also have the performance improvement achieved when the Virtex-II Pro executes at its maximum frequency and not when it is clocked with the 100 MHz clock of the OPB bus.

Present paper does not include area comparisons with other works as our target was to implement a high-speed GA. However, in the previous Section we showed that the new HGA consumes a small portion of the resources. Moreover, we do not provide comparison with software implementations as due to the high parallelization and pipelining of the genetic algorithm the existing hardware solutions are the stronger competitors. However, this information is available in the references.

VII. Conclusions and Future Work

A fully functional prototype has been developed that supports the optimization of six different fitness functions, more than any other existing design, with variable population size, member and fitness value widths. It outperforms the existing systems in terms of the processing speed, while it also supports larger population sizes. This system is fully operational and ready-to-use on the XUPV2P platform.

In order to further increase the efficiency of our approach, the role of the PowePC will be extended in order to undertake the reconfiguration of the fitness function. Moreover, even though we have achieved better results as compared to the existing FPGA designs, we plan to extend the hardware with further parallelization of the selection modules, and replication of the SM/CMM/FM pipeline. Also, we plan to identify and report the optimal parallelism level when the cost of the silicon resources is taken into account.

To the best of the authors knowledge, other works on hardware designs for different genetic algorithms optimized for larger populations have conducted experiments for up to 100,000 members [10] and for up to 1,000,000 generations [11]. The impact of the population size increase to the GA performance and the number of generations for the search of the solution are of special interest to the scientific community and thus we plan to conduct such experiments [3]. Moreover, we saw that the convergence point depends on the fitness function to be optimized; thus we will continue investigating the behavior of the system regarding the convergence point for more fitness functions and for larger population sizes.

VIII. Acknowledgment

The authors acknowledge Prof. Stephen D. Scott of the University of Nebraska-Lincoln for his help on porting the HGA design, and Charis Eufraimidis for transferring the design to the Xilinx Tools ver. 9.1 and implementing it in the Virtex-4 and Virtex-5 FPGAs. Also, they thank the reviewers for their comments that helped to improve the paper.

References

[1] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1989.

[2] S. D. Scott, A. Samal, and S. C. Seth, "Hga: A hardware-based genetic algorithm," in *FPGA Conference Proceedings*. ACM, March 1995, pp. 53–59.

[3] Y. R. Tsoy, "The influence of population size and search time limit on genetic algorithm," in *The 7th Korea-Russia International Symposium on Science and Technology*. KORUS, June-July 2003, pp. 181–187.

[4] G. Koonar, S. Areibi, and M. Moussa, "Hardware implementation of genetic algorithms for vlsi cad design," in *Computer Applications in Industry and Engineering*. ISCA, November 2002, pp. 197–200.

[5] P. Martin, "A hardware implementation of a genetic programming system using fpgas and handel-c," *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 317–343, December 2001.

[6] K. Glette, J. Torresen, and M. Yasunaga, "Online evolution for a high-speed image recognition system implemented on a virtex-ii pro fpga," in *Second NASA/ESA Conference on Adaptive Hardware and Systems(AHS)*. IEEE Computer Society, August 2007, pp. 463–470.

[7] S. D. Scott, *HGA: A Hardware-Based Genetic Algorithm*. University of Nebraska, 1994.

[8] M. Vavouras, K. Papadimitriou, and I. Papaefstathiou, "Implementation of a genetic algorithm on a virtex-ii pro fpga," in *FPGA Conference Proceedings*. ACM, 2009, p. 287.

[9] R. Tessier, V. Betz, D. Neto, and T. Gopalsamy, "Power-aware ram mapping for fpga embedded memory blocks," in *FPGA Conference Proceedings*. ACM, February 2006, pp. 189–198.

[10] J. R. Koza, F. H. B. III, J. L. Hutchings, S. L. Bade, M. A. Keane, and D. Andre, "Evolving computer programs using rapidly reconfigurable field-programmable gate arrays and genetic programming," in *FPGA Conference Proceedings*. ACM, February 1998, pp. 209–222.

[11] T. Tachibana, Y. Murata, N. Shibata, K. Yasumoto, and M. It, "General architecture for hardware implementation of genetic algorithms," in *FCCM*. IEEE Computer Society, 2006, pp. 291–292.