A Self Reconfigurable Architecture To Support Multiple Fitness Functions In Genetic Algorithm

Diploma Thesis

By

Effraimidis Charalampos

Submitted to the Department of Electronic Eng & Computer Engineering

(ECE)

Technical University of Crete

in partial fulfillment of the requirements for the ECE Diploma Degree.

Advisor: Professor Dollas Apostolos

Co-advisor: Assistant Professor Papaefstathiou Ioannis

Co-advisor: Associate Professor Pnevmatikatos Dionisios

April 2009

Dedicated to my Family and Friends

I would like to express my thanks and gratitude to my advisor, Prof. Dollas Apostolos, for giving me the honor to work beside such a distinguished scientist and teacher.

I also would like to thank Kyprianos Papadimitriou who has co-authored the paper for his valuable support and tutoring throughout the course of my thesis.

ABSTRACT

Genetic algorithms (GA) are search algorithms based on the mechanism of natural selection and natural genetics. In the past, FPGAs have been widely used for implementing Hardware GAs (HGA) to provide speedups of up to three orders of magnitude as compared to their software counterpart implementation. In this paper, we propose a new reconfigurable implementation of an HGA on a VirtexII Pro FPGA. We use the run time partial reconfiguration (RTR) technology to implement a reconfigurable fitness function and we evaluate the experimental results comparing them to previous HGA implementations.

Keywords — FPGA, Run-Time Partial module-based Reconfiguration, Reconfigurable computing, Genetic Algorithms

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Genetic Algorithms

A genetic algorithm (GA) is a technique which is used to find exact or approximate solutions to optimization and search problems. They have been developed by John Holland in his effort to abstract and explain the adaptive process of natural systems and design artificial software systems based on these mechanisms[1]. These algorithms found application in many areas due to their efficiency in finding exact or approximate solutions to optimization problems. This efficiency is an outcome of the qualities genetic algorithm are based on :

- They start their searching procedure not from a single point but from a population of points. In contrast to other heuristic methods, they span their search space in different directions. This characteristic, gives the genetic algorithms a better chance in avoiding a local optimal.

- They use an objective function - the fitness function - and do not require any auxiliary knowledge on the problem they are deployed to solve. Starting from a population of candidate solutions they make random changes to them and then evaluate the produced population of solutions using the fitness function.

- Work with the coding of the parameter set and not the parameters themselves. This enable the genetic algorithm to manipulate many parameters

simultaneously.

- Finally, they use probabilistic transition rules and not deterministic rules. With propability introduced in the transition rules the GA search process gains more flexibility avoiding local optimums.

Due to these characteristics and their performance, genetic algorithms have been efficiently applied to various NP-complete problems. Recently, the Traveling Salesman Problem(TSP) has become a target of the GA community [2]. The GAs has also been applied to boolean satisfiability (SAT) problems [3]. Those real life problems exploit great interest from scientists as they generate huge search space expoitations, resulting in great complexities and requiring serious amount of calculations. As shown in those researches, for small GA parameters, the required execution time, which provides an acceptable solution, is in the scale of seconds.

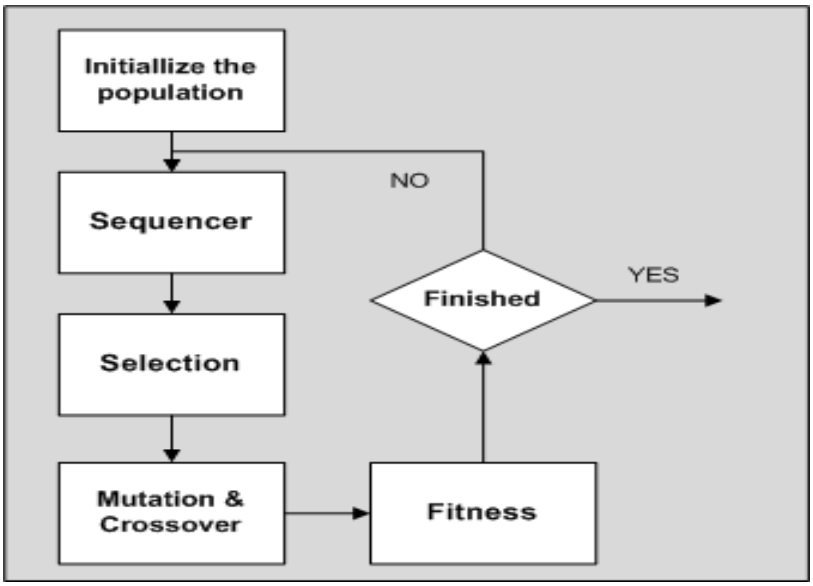

Fig. 1.1: Genetic Algorithm Flowchart

Despite the GA ability to provide good aproximate solutions to NP-complete problems, its algorithmic structure is simple. Fig. 1 illustrates a genetic algorithm flowchart. Each of the shown modules performs a simple operation:

1. The **Initiallize the population** module is responsible for the population initialization. Given the population size and members coding, this unit randomly produces a new population. This is a population of solutions on which the algorithm will start the searching process. The randomness of the initial population is essential for the GA to perform well and span its searching space in different directions.

2. The **fitness** module performs the evaluation of the chromosome. This objective evaluation function is responsible for the representation of the problem. The entire genetic algorithm will operate on the chromosomes evaluation results of the fitness function. The chromosomes that are highly evaluated will be advanced and eventually form a solution while the rest will be destroyed.

3. The **sequencer** module promotes chromosomes to the selection unit. This is not a GA basic module but we included it in the figure in order to make a connection to the hardware GAs.

4. The **selection** module decides which of the sequenced chromosomes will advance. This unit operates according Darwins theory of evolution. Darwin stated that only the best chromosomes should survive and create new offsprings. Therefore, the selection unit, judging the sequenced chromosomes fitness value, should advance to the next unit only the best chromosomes.

5. The **mutation** and **crossover** modules perform the mutation and crossover of the selected parents chromosomes in order to produce their offsprings.

The GA performance greatly depends on those two GA operators. There are many ways how to do crossover and mutation. In our case, concerning crossover, we used a single point random crossover - one crossover point is randomly selected, binary string from beginning of chromosome to the crossover point is copied from one parent, the rest is copied from the second parent.

Fig. 1.2: Crossover

Concerning mutation, we used random bit inversion - the selected bits are inverted.

1.2 Hardware Genetic Algorithms

Historically, various hardware genetic algorithm implementations have been designed for FPGAs. As shown in a recent research [4] published in the FPGA conference, more than 15 different hardware approaches were published the last 10 years. This tendency indicates a rising need for a performance improvement on the software genetic algorithms. This need for an hardware implementation arises from the overwhelming computational complexity of certain problems that causes unacceptable delays in the optimization process of software implementations. The speed advantage of hardware and its ability to parallelize offers great advantages to genetic algorithms overcoming those problems. Speedups up to 3 orders of magnitude have been observed when frequently used software routines were implemented in hardware by way of reprogrammable field-programmable gate arrays (FPGAs)[5].

1.3 Run Time Partial Reconfiguration

The past few years, many vendors have introduced and added the Run Time Partial Reconfiguration (RTR) technology to certain FPGAs. This technology provides great adaptability, adding general purpose features to an instance based architecture, with the trade-off of the reconfiguration overhead. The RTR technology enables the device to be partially reconfigured on the fly, leaving intact placement and routing of the rest of the design and without interrupting its normal operation.

There are two main partial reconfiguration methods supported by xilinx, the difference based and the module based. The difference of these methods is the programmed area of the reconfiguration bitstreams.

- In **module based**, the target area is an entire region. The created partial bitstreams reprogram the entire region even if small changes where required. This increases the reconfiguration overhead as the reconfiguration and introduces large reconfiguration delays. This method is essential for reconfiguring large areas of the FPGA.

- In **difference based**, the target area are the FPGA LUTs. The created partial bitstreams reprogram only the parts that need to be changed. Therefore, this method allows faster reprograming as it is only affecting the logic that requires modifications. This method is essential for reconfiguring small parts of the FPGA.

In our project we used an XUP virtexII pro and depending on the FPGA device there are various reconfiguration features. Specifically, this board doesn't support 2-D reconfiguration as the supported reconfiguration is column based. It

is mandatory to floorplan two reconfigurable regions on the same column. This is because the reconfiguration bitstreams affect the entire columns operation[6].

Partial reconfiguration can either be implemented through the JTAG or with the use of an embedded processor through ICAP. The JTAP reconfiguration requires the presence of an external PC while the other method, refered as self-reconfiguration, doesn't. In the self-reconfiguration method, the partial bitstreams are loaded from a ROM device and are written to Internal Configuration Access Port(ICAP) in order to perform the reconfiguration. The On-Chip Peripheral Bus Hardware ICAP (OPB HWICAP) enables the embedded processor to perform the reconfiguration with the read-modify-write mechanism. It reads the internal memory, modifies the data he read according to the partial bitstream and then writes them back to the internal memory[7].

The theoritical advantages of using dynamic reconfiguration are mostly focused on speed, power and flexibility. Specifically, the advantages are:

1. **Task Speed** - the use of RTR for a co-processor region enables the device to migrate more functions without being constrained by the amount of hardware available. Therefore you have the option to instantiate a more optimized function to obtain a speedup.

2. **Power Consumption** - this is obtained either by loading blank bitstreams in the PRR that are not used and therefore reducing the static leakages of the components or by loading a more optimized design that will reduce the dynamic power consumption.

3. **Survivability** - The reconfiguration can be used to detect and restore or relocate damaged functions.

4. **Mission Change** - The subsystems of the architecture can be changed to

fit to different missions requirements.

5. **Enviroment Change** - The system can adopt to the enviromental changes.

6. **Algorithm adaptive change** - The reconfiguration enables the device to implement algorithms with different constraints and parameters. It makes the necessary changes on the algorithm to fit to the task requirements.

7. **On-line system test and self-healing** - The RTR allows the generation of a Built-In Self Test at runtime to confirm and evaluate the ability to continue operating. In case of fault detection the the entire design or the corrupted module can be reconfigured to correct their operation.

8. **Hardware virtualization** - All functional block are not at the same time instantiated in the device. Virtually, the user has the support of all those functions but physically there are not present.

1.4 A new generation of HGA

With the use of the RTR technology we present a dynamically reconfigurable HGA that solves various problems efficiently exploiting the device resources. Compared to the prior state of art, this architecture dynamically loads the requested fitness function from a ROM device, changing the target problem and overcoming the devices area constrain problem. Furthermore, it has the ability to incrementaly build its fitness function reducing the total reconfiguration overhead. Applying the RTR technology on HGAs, this design can solve almost any problem.

1.5 Contributions of these Work

The contributions of this work are :

- Optimization of an HGA and parameterization of its features for maximum problem support
- Implementation of reconfigurable general purpose mathematical unit (HGA fitness function) that replaces the GAs fitness function providing support to numerous different fitness functions
- Presentation of a new PRM architecture to reduce bus macro usage
- Effective use of RTR for real-life problem
- Illustration of the RTR advantages: better area utilization and speed performance, mission change and power optimization
- Autonomous operation through self-reconfiguration
- This work is the first to exploit RTR on the GA problem

2. STATE OF THE ART

2.1 Hardware Genetic Algorithms

The last twenty years various implementations on hardware genetic algorithms were introduced. The characteristics that distinguished those implementation where their parameters range (i.e. population size, member width) and the instantiated fitness function. Despite the differences of those implementations, they all shared one similarity. All the HGAs solved a single problem with one fitness function type.

Scotts et al [5] proposed and implemented a general HGA on a BORG prototyping board which consisted of 5 Xilinx XC4000 FPGAs. This design exploited effective parallelism and coarse-grained pipelining. Their target problem was to maximize the result of a mathematical function.

Koonar et al [8] developed a reconfigurable GA for VLSI CAD design. This work presents a GA architecture for circuit partitioning in VLSI physical design automation. They used a Virtex board to synthesize the design and to provide the necessary autonomy.

Emam et al [9] proposed a genetic algorithm on non-linear adaptive filters for the purpose of blind signal separation. Their target device was an UniDaq -PCI -PC board, and they report the performance of the designed GA without the presence of a fitness function, claiming that the performance of this implementation is determined by its fitness function.

Narayanan et al [10] developed genetic algorithm modules for intelligent systems. They present the design of two libraries (HDL and VHDL) of hardware modules for a GA system. These libraries were based on a Matlab library. In this work they optimize two polynomial functions with different combinations of modules contained in the library and evaluates the results.

Tommiska et al [11], in a recent project, designed an HGA with Altera Hardware Description Language (AHDL). The GA run on a PC card and was connected to the CPU through PCI bus. This design was also downloaded on an FPGA. They achieved an improvment rate of 200x vs software.

We conducted a research on all up to date HGA FPGA implementations and we present all the intresting information in the following table.

The table information as well as additional information on HGA implementations can be found in a recent research [4].

Hardware Genetic Algorithm FPGA Implementations		
HGA	Problem Type	Supported Fitness Functions
Vavouras [4] et al	arithmetic operations	6
Koonar [8] et al	VLSI circuit partitioning	1
Tang [12] et al	arithmetic operations	1
Aporntewan [13] et al	arithmetic operations	1
Tommiska [11] et al	arithmetic operations	1
Emam [9] et al	blind signal separation	1
Scott [5] et al	arithmetic operations	6
Mostafa [14] et al	arithmetic operations	1
Martin [15] et al	arithmetic operations	1
Koza [16] et al	minimal sorting network	1
Heywood [17] et al	arithmetic operations	1
Perkins [18] et al	signal processing	1
Lei [19] et al	arithmetic operations	1
So [20] et al	video encoding hardware	1
Graham [21] et al	traveling salesman	1
Glette [22] et al	image recognition	1
Shackleford [23] et al	arithmetic operations	1

Tab. 2.1: Hardware Genetic Algorithm FPGA Implementations

2.2 Genetic Algorithm Applications

GAs have been efficiently applied to various NP-complete problems to provide fast aproximate solutions. Recently, the Traveling Salesman Problem(TSP) has become a target of the GA community [2]. The GAs have been applied to boolean

satisfiability (SAT) problems [3]. Both problems generate great interest from scientists as they can address huge search space expoitations. This results in great complexity and requires serious amounts of calculations, hence execution time. As shown in recent literature even for small GA parameters, the required execution time, which provides an acceptable solution, is in the scale of seconds.

The RECOPS[24] was started in 2007 - one of the largest projects in Dynamic Partial Reconfiguration (DPR). This project evaluates the advantages of this technology in military electronics applications. The experiments performed on a wide range of applications such as:

- Image acquisition and transfer
- Electronic Warfare
- Image Processing
- Front end processing
- Short Range Radio Modem
- Software Defined Radio transmitter
- Software Defined Radio receiver

Finally, GAs have been succesfully applied to other engineering and computer sience [25]. They provide solution to problems such as:

- Electromagnetic System Design
- Optimization problems in Aerodynamic Design
- Load Balancing in the Process Industry
- Optimization in computational fluid dynamics

2.3 Run Time Partial Reconfiguration

During the last decade, Run Time Partial Reconfiguration technology has been widely studied as a research topic. A strong example of this tendency is the RECOPS[24] project(Reconfiguring Programmable Devices for Military Hardware). This project began in 2007 and its goal was to examine the potential advantages of the usage of RTR in military applications. This project covered most of the fields of military applications by experimenting in the previously mentioned(paragraph 2.2) applications and emphasizing on the Software Defined Radio (SDR).

The first results are already published [26] and the RTR technology was characterized as promising. The theoritical advantages of the RTR were compared to the practical experimental measurments. Unfortunately, apart from the additional flexibility the RTR brings on to HW, the classical area, speed and power improvements were depended on the application and therefore not deterministically resulting in an imporvement for the application. On the other hand, they claimed that the tested application can benefit from RTR for online system test, survivability, and hardware virtualization.

3. HGA DESIGN

The project is separeted into two parts which at the end are combined to form a new generation self reconfigurable architecture to support multiple fitness functions in genetic algorithms.

3.1 Design issues

For the design implementation we used the Xilinx Design tools ISE 9.1, EDK 9.1 and Planahead 10.1. The design flow was guided by the Early Acces Partial Reconfiguration (EAPR) [27] laboratories documentation. This simplified RTR design flow is available thanks to Planahead [28]. Planahead provides a graphical user-friendly enviroment that simplifies the RTR design flow. In addition, it supports incremental designing. With this feature, the designer can partially modify the design, leaving placement of the rest intact and thereby shortening design iterations, maintaining the required performance.

3.2 Hardware genetic algorithm without PR

Starting the optimization process, the first step was to increase the HGAs supported parameter range. The goal was to fully parametrize the algorithms attributes in order to support the maximum possible number of parameters for the target device [29] features. This was not straightforward, since the increased pa-

New Hardware Genetic Algorithm Attributes	
Attribute	Value Range
Memory	4096*46bits
Population size	32
Max Supported Population Size	2044
Member Width	10 bit
Fitness Width	36 bit
Sum of Fitness Width	46 bit
HGA Max Clock Frequency	127 MHz

Tab. 3.1: New HGA Attributes

rameters introduced a considerable degradation in the clock frequency. This was mitigated by code optimization. We found that the part of the algorithm that caused the clock frequency reduction was the comparison of a members fitness value with a threshold set by the selection unit. We then replaced this comparison by a subtraction and a 1-bit comparison, used to identify whether the subtractions result was positive or negative. Unavoidably, we had to add an extra Finite State Machine (FSM) state to handle the operations.

After the design parametrization, larger chromosomes were tested. Specifically, we tested 10bit member width with 36bit fitness function width as shown in Table II. This was a significant modification compared to the previous implementation [4], which used only 4bit members and 14bits fitness function width.

Furthermore, with these parameters we increased the population distinct members without increasing their population size. This is because with 4-bit members there is a maximum of 16 different members represented, therefore duplicating the same members on a population of 32 members, whereas with 10-bit

members this duplication is avoided. This accretion in the disctinct population members gives the GA increased efficiency as it avoids a premature convergence to a local optimum. The genetic algorithm will perform well and achieve better results as the rate of population convergence will be less than the rate of search space reduction [30].

Additional modifications were made to the selection module. We changed the selection process in order to accelerate its decision. We focused on the selection because it is the unit with worst throughput and affects the efficiency of the entire pipelined architecture. We modified this unit in order to operate faster and changed its selection criteria. The new criteria is the average fitness value of the population members. The selection unit will iterate until either it finds a member which fitness value is above a threshold, or it reaches an iteration limit which will force the unit to select a random member.

Finally, there were some slight deviations from the GA theory in the crossover and mutation module. After the crossover, the second offspring was not an result of the crossover operation, but the second parent.

The analysis of each module structure and operation is explained in the following sections.

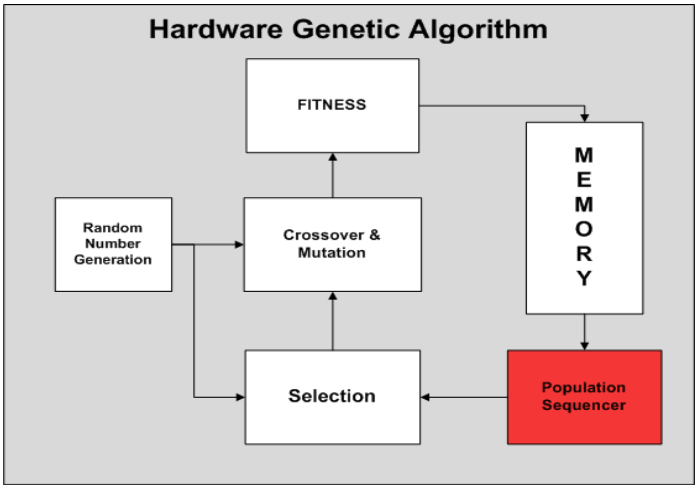

Fig. 3.1: Hardware Genetic algorithm (Population Sequencer)

3.2.1 Population Sequencer

The population sequencer module is responsible for the retrieve of the population members. It has a Finite State Machine (FSM) to handle the required operations of this module. These operations are the initiallization of certain parameters and the sequencing of members to the next module.

Specifically, during the initiallization process, the population sequencer requests from the memory the population size from the previously determined address. After the initiallization, it starts requesting population members from the memory. It has an internal counter that starts counting from 0 to the population size. When the counter reaches the population size limit it resets and starts all over again, requesting from 0.

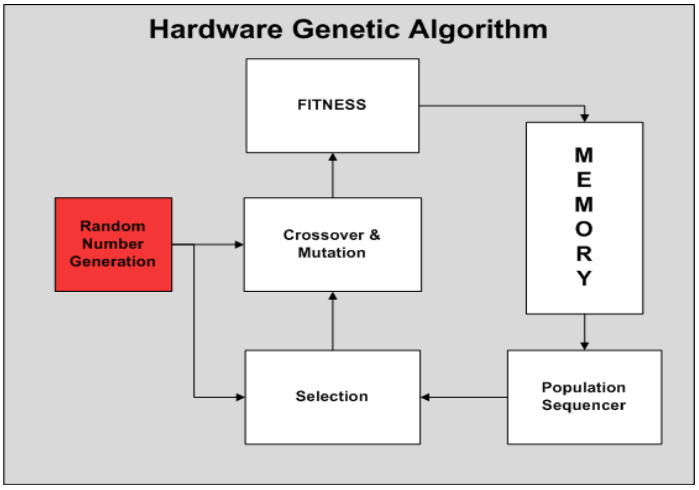

Fig. 3.2: Hardware Genetic algorithm (Random Number Generator)

3.2.2 Random Number Generator

The random number generator module is generating random numbers for the crossover&mutation and selection modules. This module has a more complex FSM than the population sequencer unit. The implemented algorithm for the pseudo random number generation is based on the 150-150-90... hybrid CA described in Serra et al.

Specifically, the operation of this unit requires a random seed. During the initialization process, it requests this seed from the Memory. Afterwards, and based on this number it starts generating random numbers. Those numbers are used for the calculation of the mutation probability and the crossover point.

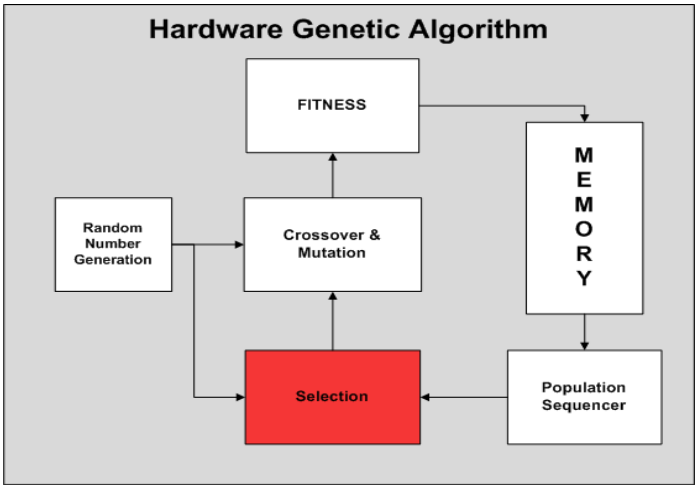

Fig. 3.3: Hardware Genetic algorithm (Selection)

3.2.3 Selection

The selection unit is responsible for the selection of the surviving population members. It has a complex FSM to handle the various calculations needed to choose the surviving parents. The implemented algorithm for the members selection is based on the roulette wheel. The parents that are most likely to be selected are those with the largest fitness value.

The FSM starts with the calculation of the number that it will use as a threshold for the selection process. This threshold is the average fitness value of the entire population. In order to avoid a time consuming division we calculated this number by assuming that the population size is a power of 2. Under this assumption we can find the average number by right shifting the sum of fitness value by $\log_2 (population\ size)$.

After the calculation of the threshold, the selection unit recieves the sequenced members and evaluates them until it finds two members that pass it. The maximum number of evaluated members per choice is constrained by a number in order to constrain the maximum delay and add some randomness to the selection.

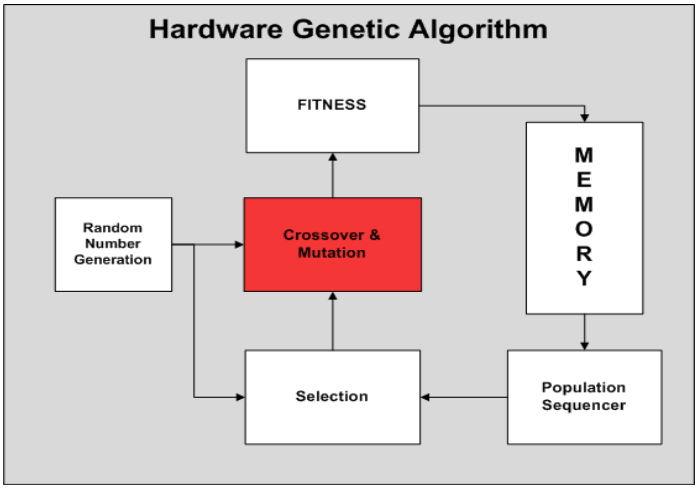

Fig. 3.4: Hardware Genetic algorithm (Crossover and Mutation)

3.2.4 Crossover & Mutation

The crossover& mutation module is responsible for mutating and performing the crossover to the parental members in order to generate the two new offsprings for the new population. The FSM handles the initialization and the application of the above mentioned genetic parameters. The implemented algorithm uses a one-point random place crossover and multiple bit inversion (mutation).

The entire process starts by fetching the mutation and crossover probabilities from the Memory module. When the initialization is complete, this module waits for the selected parents. When the selected parents are recieved it performs the mutation and crossover according to the probabilities. This probabilities recieved formulate a threshold and are compared to the random numbers recieved from the generator to define the crossover point and the mutated bits.

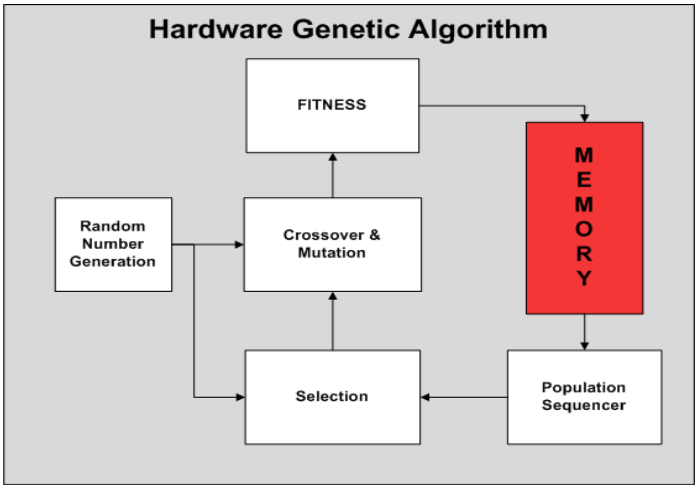

Fig. 3.5: Hardware Genetic algorithm (Memory)

3.2.5 Memory

This unit is the front-end and back-end memory controller of HGA. It handles all the incoming requests of the HGA modules and the PowerPC. To handle those operations it uses an FSM unit giving the highest priority to the modules that perform the least memory transactions.

Specifiacally, this unit recieves a request and an acknoledge signal from the other HGA modules. When a request signal is high and according to the modules priority, the memory module decides and serves the most important request. In our case, the sequencer has the least priority as it is constantly requesting numbers.

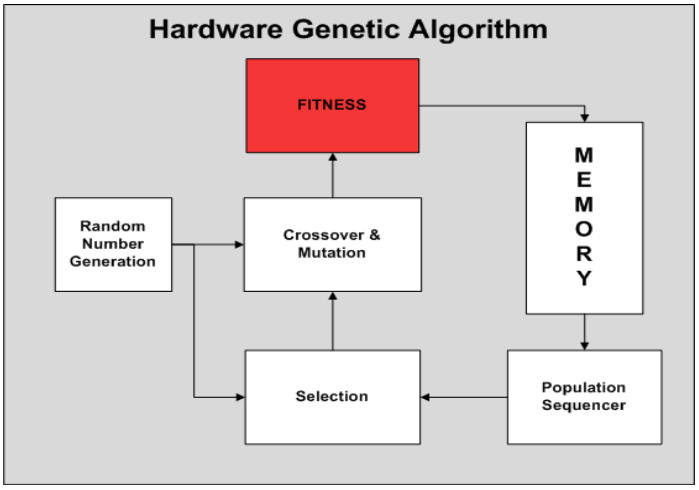

Fig. 3.6: Hardware Genetic algorithm (Fitness)

3.2.6 Fitness

Finally, this unit is the larger unit in terms of resources utilization (see table 3.2). This unit has an FSM to handle the initialization process, the fitness evaluation process of the offsprings and the accumulation of the population fitness values.

The process starts with the initialization of three parameters: population size, sum of fitness of the initial population and the number of generations. This parameters are essential for the operation of the fitness module. After the initialization, it recieves the offsprings from the crossover&mutation module, it evaluates their fitness value and writes them to the memory, accumulating their fitness.

3.2.7 Modules Features

The following table shows certain important features for the hardware genetic algorithm modules retrieved from the ISE synthesis report.

HGA Module Features					
Module	Slices	Flip Flop	LUT	MULT	Max.Frequency
Population Sequencer	76	99	74	0	198.620MHz
Random Number Generator	28	39	37	0	486.665MHz
Selection	309	110	555	0	213.847MHz
Crossover&Mutation	57	58	89	0	305.535MHz
Fitness	505	434	932	10	153.994MHz
Memory	90	129	135	0	248.139MHz

Tab. 3.2: HGA Module Features

3.3 Applying PR to create a reconfigurable fitness function

To begin with, we replaced the HGA fitness function with a reconfigurable mathematical unit. This unit comprises of 4 partially reconfigurable regions (PRR). We were constrained to the specific number of PRRs because the VirtexII Pro FPGA doesn't support 2D reconfiguration. It is mandatory to assign two different PRRs on the same column [6]. In each region a partially reconfigured module (PRM) is loaded. Each module applies simple mathematical equations to the inserted binary number. Furthermore, each module has 32-bit input and output values and it is designed to process numbers ranging from 1-64 bits. It has 32-bit interface but it internally reproduces up to 64 bit number. This affects the mathematical unit performance as it needs 2 clock cycles to create a 64 bit number.

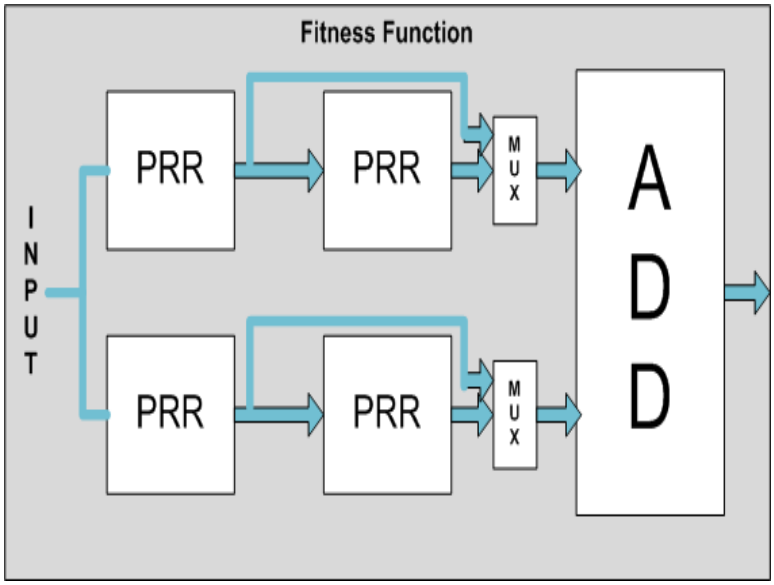

Fig. 3.7: Mathematical Unit Structure

We implemented three PRM for each PRR: multiplier, adder and the zero unit.

All the PRMs route their output to the final adding unit. The ADD unit summarizes the results of the functional PRMs. The first column of PRMs also routes their output to the sequential PRM, as shown in figure 3. This sequential processing produces complex mathematical equations. Considering it performance, the mathematical unit is pipelined and its performance is affected by the downloaded mathematical units. Each PRM needs 2-4 cycles(1 cycles for 1-32 bit input and 2 cycles for 33-64 bit input) to create the I/O numbers + X cycles the function execution. For example the achived throughput for 32bit input, 64 bit output and 4 cycles for the function execution would be: 64bit/((1+2+4)cycles*127MHz) = 719 Mbps.

Partial Reconfiguration Region Resources	
Site Type	Available
LUT	1216
FF	1216
SLICE	608
MULT18X18	4
RAMB16	4

Tab. 3.3: PRR Resources

The internal structure of each PRM is constituted of three modules: The **inputdata** module which recieves 32bit numbers and gradually forms the input number. The **function** module which evaluates our input numbers and the **outputdata** module which dissolves the output number of the function into 32bit numbers, passing them to the next module. The synchronization of the PRMs is achieved through the **write** signals. When the write signal is High the next module stores the 32bit input data. The following figure shows the PRM struc-

ture.

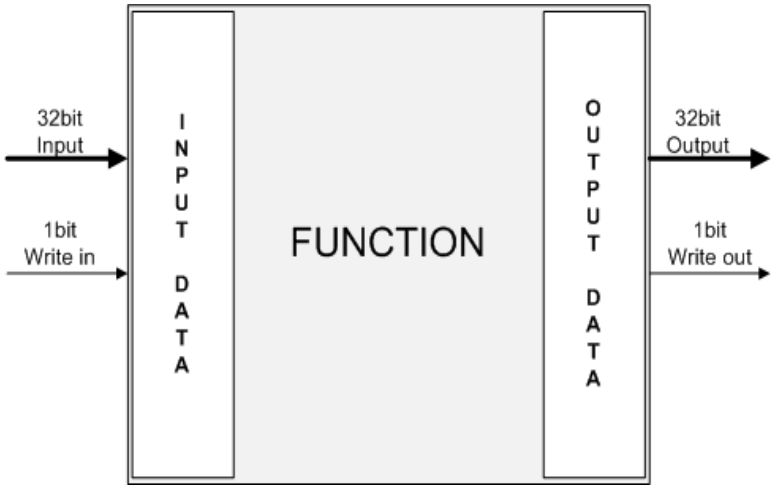

Fig. 3.8: PRM STRUCTURE

The inputdata and outputdata modules are very important for a dynamic reconfiguration project. They minimize the required Bus Macros (BM) for the communication between PRRs. The BM presence in the PRRs constrain their shape, as well as the designers options. A reduction in the BMs provide greater flexibility to the design floorplan and a better routing.

This mathematical unit is fully pipelined and its throughput is limited by the slowest module. It also provides some extra features: 1) a fixed number can be saved in each module, 2) the functioning collumns and the input range are controlled by a register and 3) it has an internal reset signal.

The designing choices of the mathematical unit are made with respect to generality. This architecture is a platform, that thanks to RTR technology, can solve various np-complete problems. Depending on the modeling of the problem, only the necessary fitness evaluation modules need to be designed and downloaded. With the RTR technology, an accurate problem modeling and the implementation of the corresponding fitness evaluation module the FPGA can be transformed

into a multiple np-complete problem solving device.

3.4 Combining HGA and RTR to form the New generation HGA

3.4.1 Design Flow

Our RTR design flow followed the EAPR [27] lab 3 guideline. This guideline was strictly followed to avoid any unwanted problems and succesfully create a dynamically partial reconfigurable project. In the following text the design flow steps and certain tips are introduced:

1. We created a processor system with the EDK design tool. An processor and OPB HWICAP is required in a RTR project to execute the self-reconfiguration operations.

2. Create the custom logic peripherals IP. This peripheral will implement our algorithm, in our case the hardware genetic algorithm, and it is splitted in two parts - the static and the dynamically reconfigurable.

3. Synthesize the previously created IP. In this step it is very important to group our static logic under one module. This is mandatory for the production of an succesfully speed and area globally optimized netlist. The partial synthesised vhdl files produced an partial optimized netlist which cause placement and routing problems in their finally assemply. By synthesizing all the static logic together, you produce an more optimized netlist producing a better placement and routing for the design. On the other hand, the dynamically reconfigurable vhdl files can only be synthesized separately.

The creation of an more optimized netlist can be achieved by changing certain synthesise process options in ISE. Those options are:

$$SynthesisOptions(TAB) -> OptimizationEffort -> High$$
$$XilinxSpecificOptions(TAB) -> PackI/ORegistersintoIOBs -> No$$
$$XilinxSpecificOptions(TAB) -> AddI/OBuffers -> unchecked$$

4. Connect the processor system and the IP under a top level design. We synthesized the top level having instantiated the dynamically reconfigurable logic as black boxes. This will perserve the hierachy for these modules and it will be used in the Planahead design tool for the floorplanning.

5. Create a Planahead project. Import in the project the all the created netlist files , create the are groups and place all the dynamically reconfigured components and their bus macros. The following figure illustrates the placement used in our design.

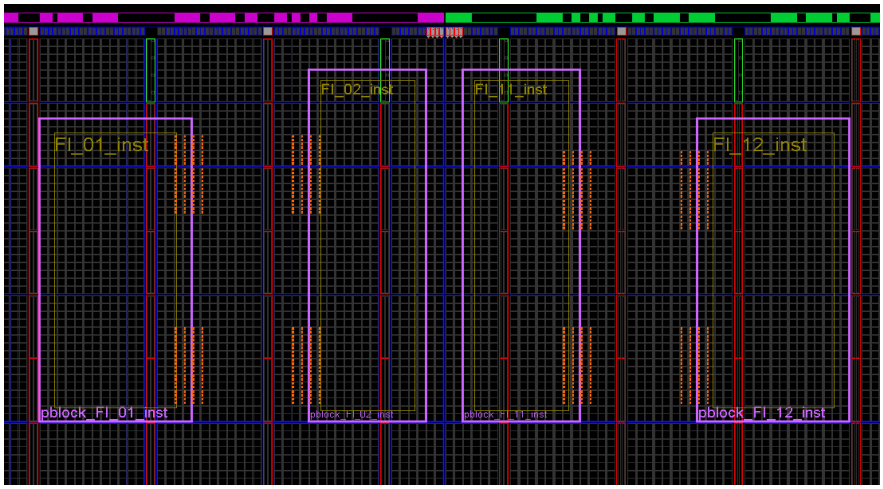

Fig. 3.9: PlanAhead Placement

6. Finally, run a DRC check and execute the PR flow commands to produce the static and partial bitstreams.

3.4.2 Structure and Modifications

Applying the RTR to the modified HGA we created a new generation HGA that supports multiple fitness functions. We replaced the fitness function of the HGA presented in 3.1 with a reconfigurable mathematical unit we introduced in section 3.2. The outcome of this act is shown in figure 3.10.

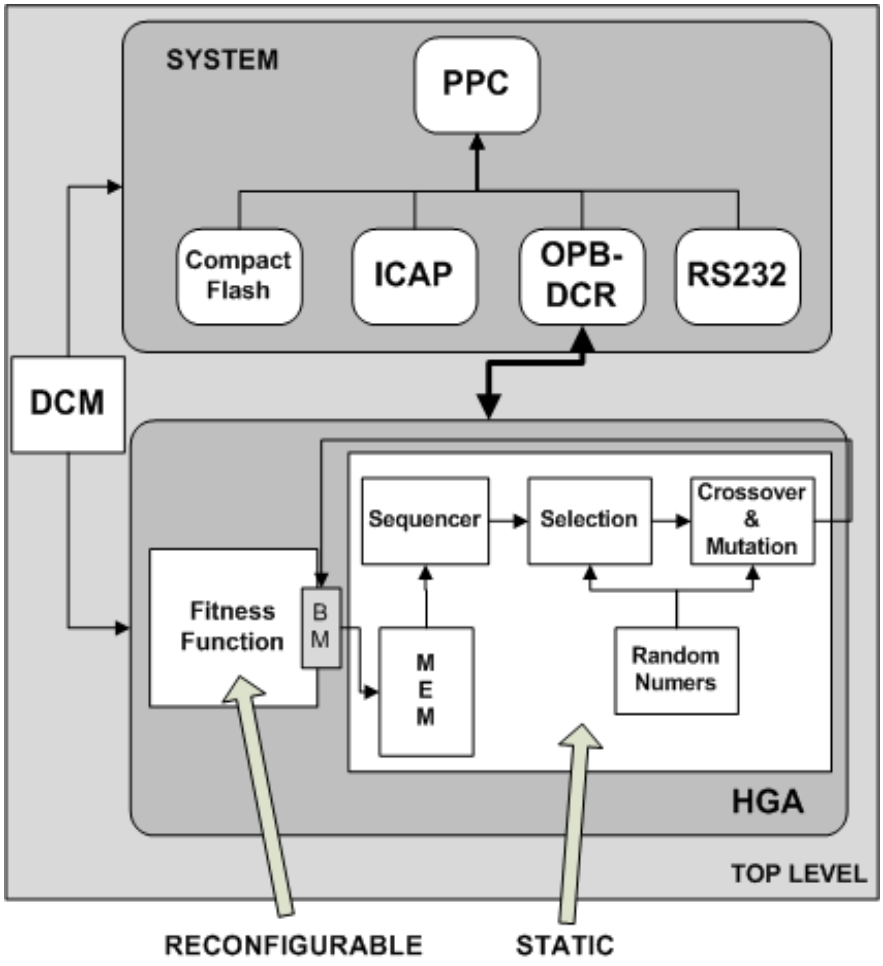

Fig. 3.10: Top Level Structure

As shown in the figure 3.10 various modification were made to the previous HGA architecture. The reasons behind the design decisions are:

- We needed an embedded processor to perform the self reconfiguration and

handle the I/O requirements. This forced us to create an basic system in EDK with the following specifications:

1. A Power PC
2. A RS232 peripheral to support the communication with the PC through the serial port
3. An OPBHICAP peripheral to support the reconfiguration procedure with its simple interface commands
4. A System ACE peripheral to control the CF ROM where the bit-streams are saved.
5. A OPB2DCR bridge and a DCR bus. We utilized DCR registers to make the opb signals external to the EDK project avoiding any problems

- The new generation HGA will be an OPB custom logic peripheral and its reconfigurable regions must be brought to the Top level. For these reasons, this peripheral must be connected with the System under a top level module. The peripheral must be withdrawn from inside the System and externally connect to the DCR bus and therefore connect indirectly to the OPB.
- The DCM must be removed from inside the EDK project and instantiated in the Top level. This is performed for the Design Tools to identify the Global clock. In addition, if this step is not performed, the synthesis operation in Planahead will exit with an error refering to the DCM.

4. IMPLEMENTATION RESULTS OF THE NEW GENERATION HGA

The implementation results are separeted into two section - the static and the partial comparison. We found that it was more convenient to compare the new generation HGA execution results and area features separately in those sections. The HGA we are comparing to is the Vavouras et al HGA [4].

4.1 Static comparison

In this section we compare the execution results and the algorithmic performance of our HGA. In order to accuratelly compare those two implementations we used the following parameteres.

HGA parameters	
Parameter	Value
Member Width	10
Maximux Fitness Width	36
Population Size	32
Clock Frequency	100MHz
Number of Generations	100

Tab. 4.1: HGA parameters

The results indicate an improved execution time for the new generation HGA. Specifically, our implementation required 178000 clock cycles for 100 generations while the vavouras et al [4]HGA required 218000cycles. The calculated speedup is:

Vavouras execution time=218000cycles/90MHz=2.42 msec

Our execution time=51603cycles/100MHz=0.51msec.

Speedup = 4.7.

Apart from that, our implementation acquired a faster convergence to the global optimum. As shown in the figures our generation convergence point ranged from 7 generations while in Vavouras et al HGA the convergence point ranged from 22-28.

Generations Convergence Points (Population Size 32)		
HGA	Function	Generations Convergence Point
Vavouras	x	22
Vavouras	$x + 5$	25
Vavouras	$2 * x$	28
Vavouras	x^2	22
Our	x	7
Our	$x + 5$	7
Our	$2 * x$	7
Our	x^2	7

Tab. 4.2: Generations Convergence Points

4.2 Partial comparison

The new generations HGA area requirements where increased in comparison to the previous HGA. These differences are shown in the following table.

XCV2P30 Device Utilization					
HGA	LUT	FF	SLICE	MULT18x18	RAMB16
Vavouras	11%	7 %	16 %	11 %	7 %
Our	36.9%	37.5 %	37.5 %	11.7 %	37.5 %

Tab. 4.3: XCV2P30 Device Utilization

This accretion in the device utilization features was the result of the PRRs and the bus macros. The partially reconfigurable regions we commited for reconfiguration purposes increased the device constrained features. The static design could not utilize those regions even if the loaded PRM where blank bits. So it is more appropriate for the comparison of the HGAs utilized device resources to add the constrained features(shown in table 4.4) of the each PRR to the total amount of utilizated resources.

Partial Reconfiguration Region Resources	
Site Type	Available
LUT	1216
FF	1216
SLICE	608
MULT18X18	4
RAMB16	4

Tab. 4.4: Table IV. PRR Resources

The size of those reconfigurable regions was determined by two factors - the needed resources of the largest PRM and the bus macro placement.

5. EXPERIMENTAL RESULTS

5.1 Genetic Algorithm Performance

In this section we show the experimental results after exhaustive experiments with different genetic parameters. We illustrate how the size and variety of the initial population affects its performance, reducing the generations needed for the genetic algorithm convergence.

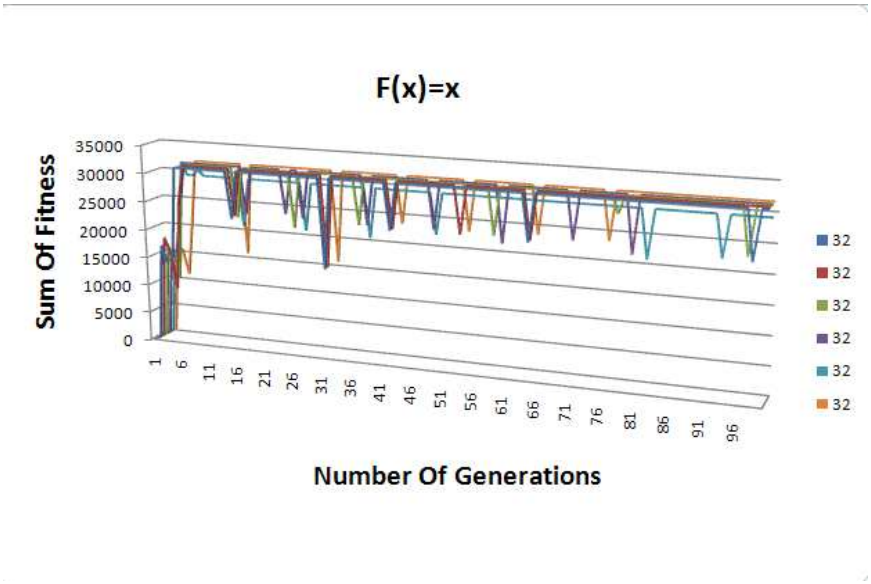

Fig. 5.1: GA Results Population size 32 Function = x

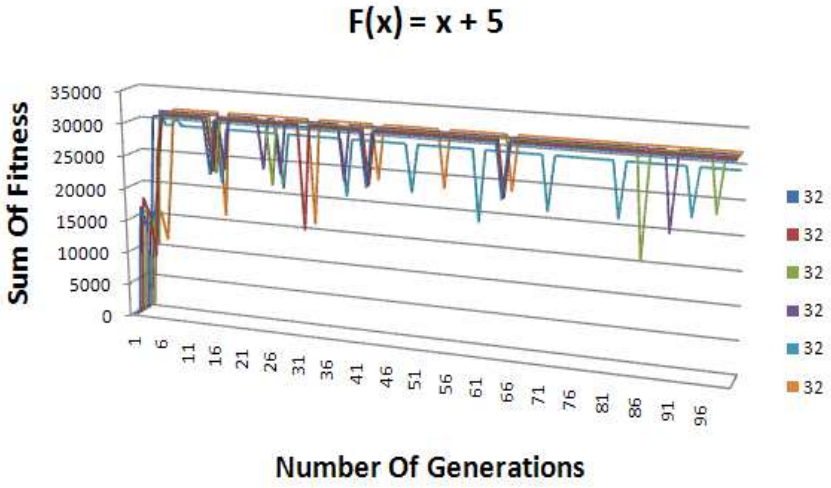

Fig. 5.2: GA Results Population size 32 Function = x + 5

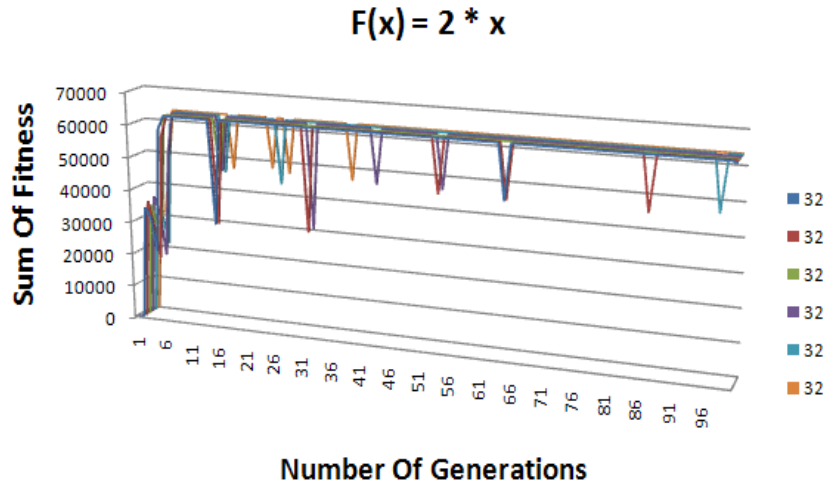

Fig. 5.3: GA Results Population size 32 Function = 2*x

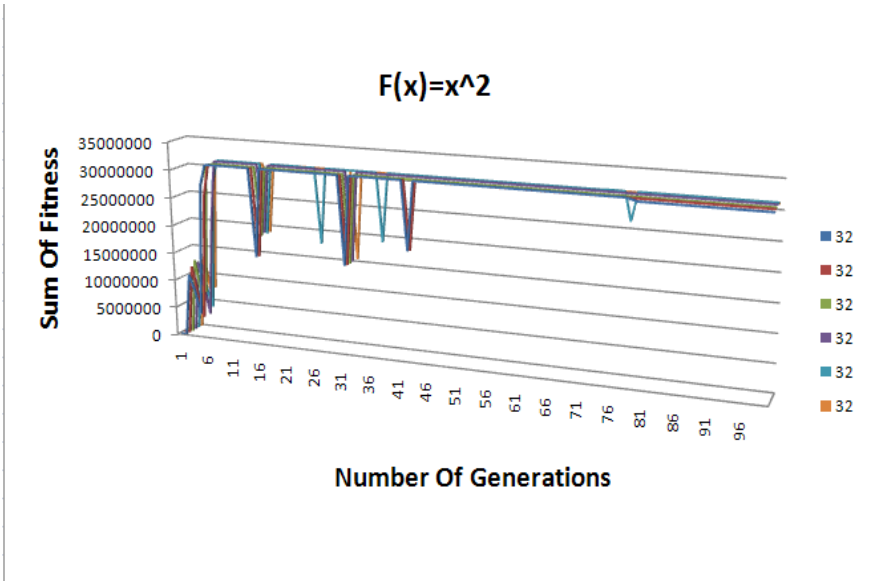

Fig. 5.4: GA Results Population size 32 Function = xˆ 2

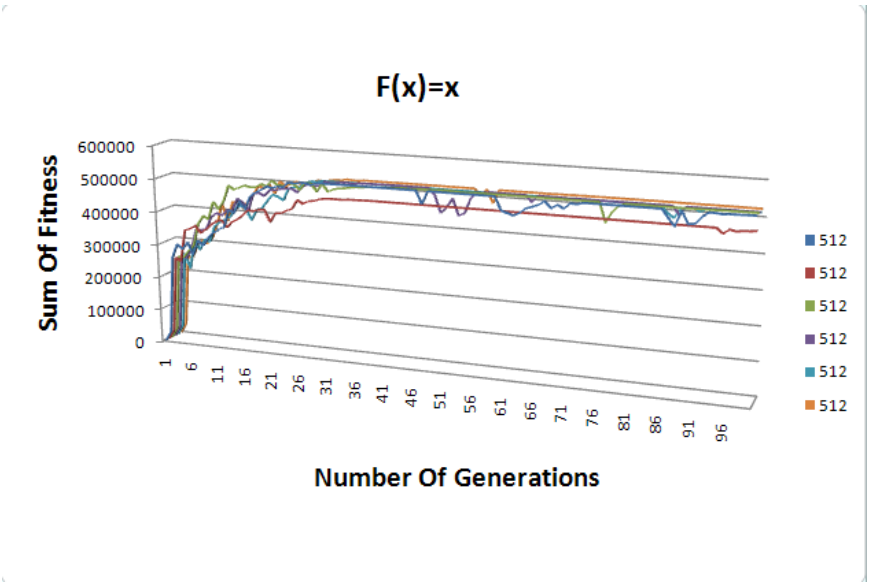

Fig. 5.5: GA Results Population size 512 Function = x

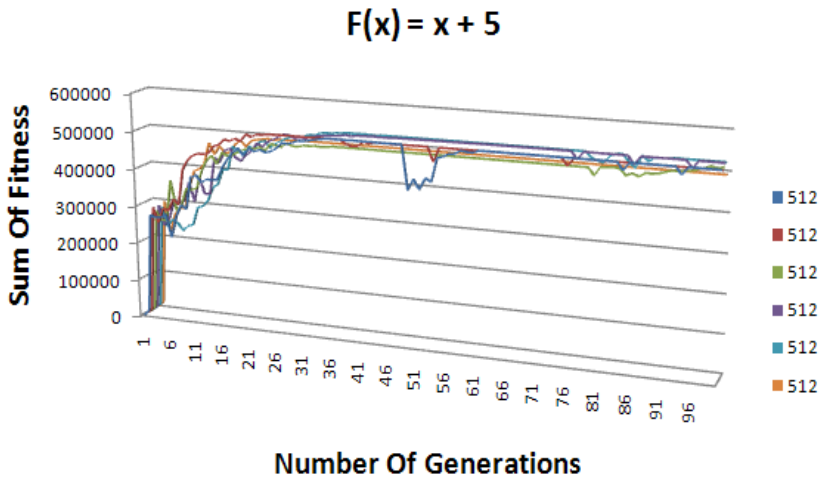

Fig. 5.6: GA Results Population size 512 Function = x + 5

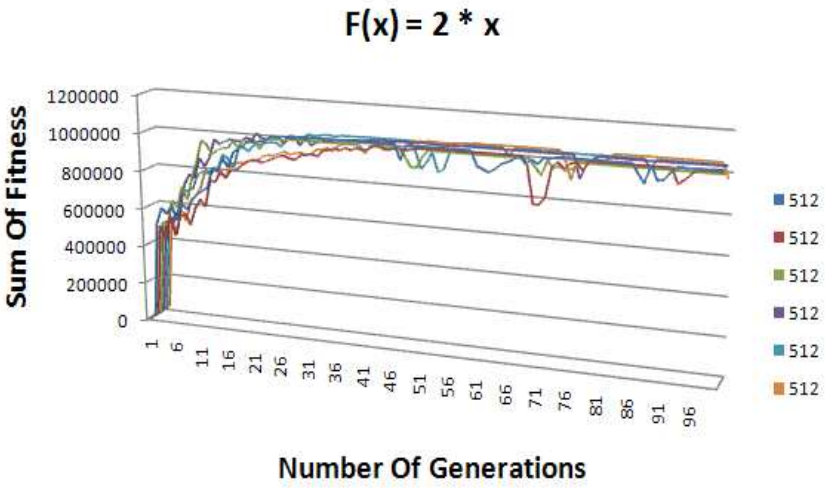

Fig. 5.7: GA Results Population size 512 Function = 2*x

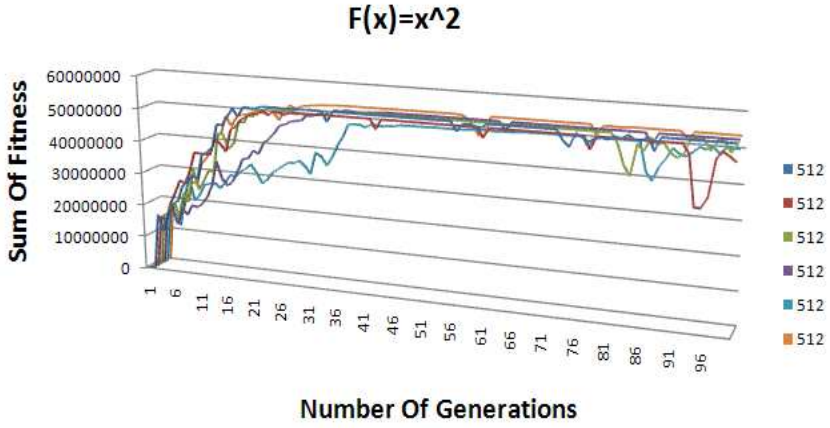

Fig. 5.8: GA Results Population size 512 Function = xˆ 2

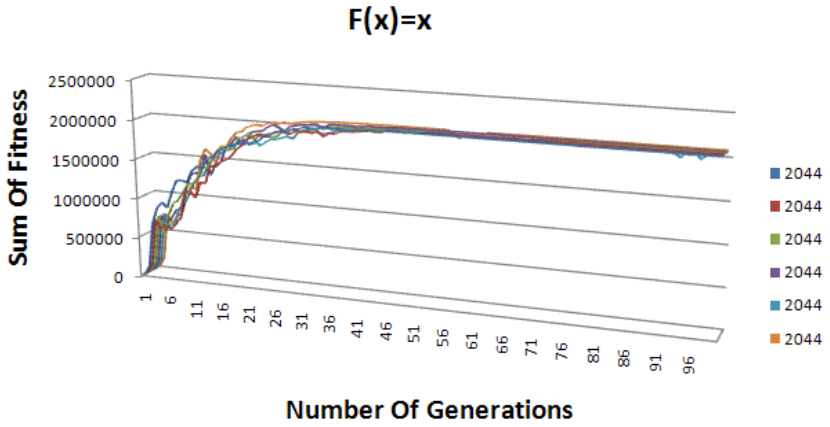

Fig. 5.9: GA Results Population size 2044 Function = x

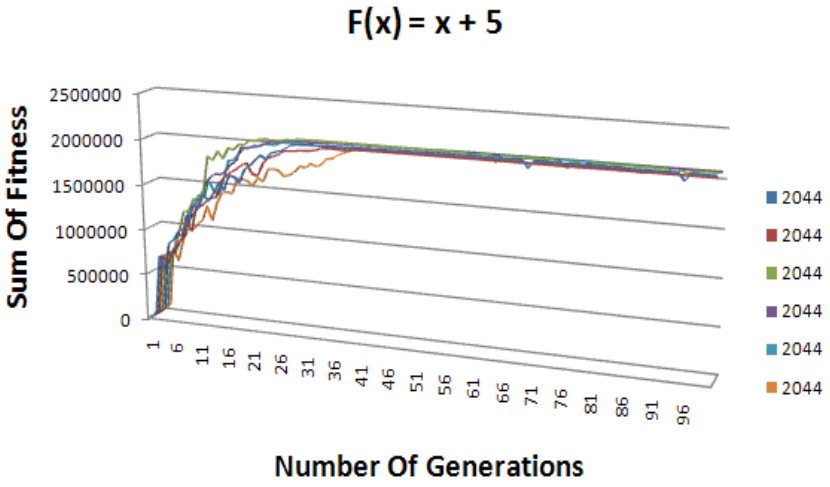

Fig. 5.10: GA Results Population size 2044 Function = x + 5

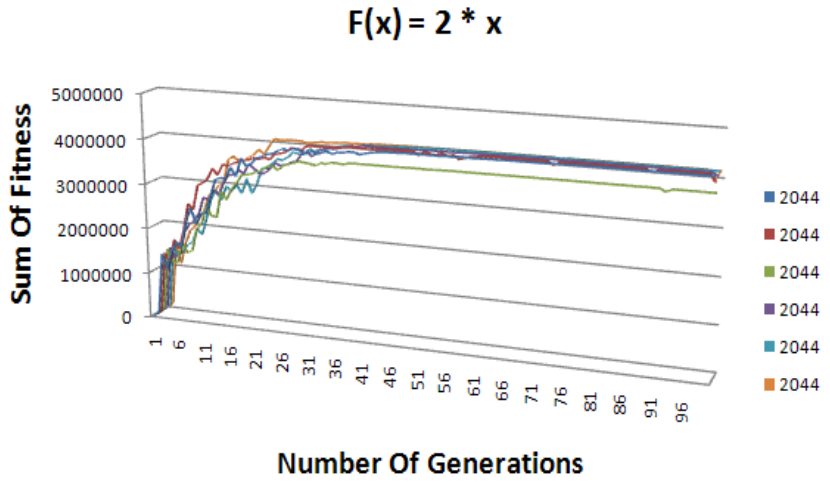

Fig. 5.11: GA Results Population size 2044 Function = 2*x

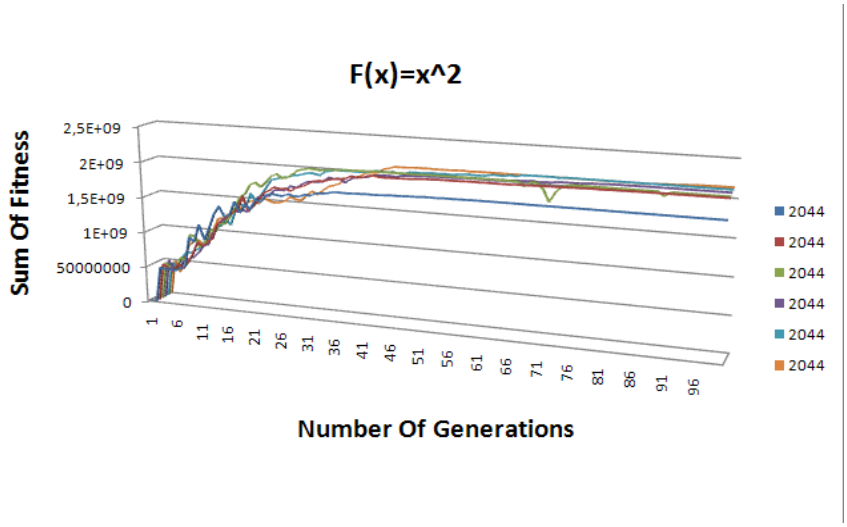

Fig. 5.12: GA Results Population size 2044 Function = xˆ 2

As shown in the above figures, in all cases the genetic algorithm converged to the global optimum after approximately 6-26 generations. The generations need for the convergence where different among the tested population sizes. The larger population sizes caused a slower but more mature congvergence.

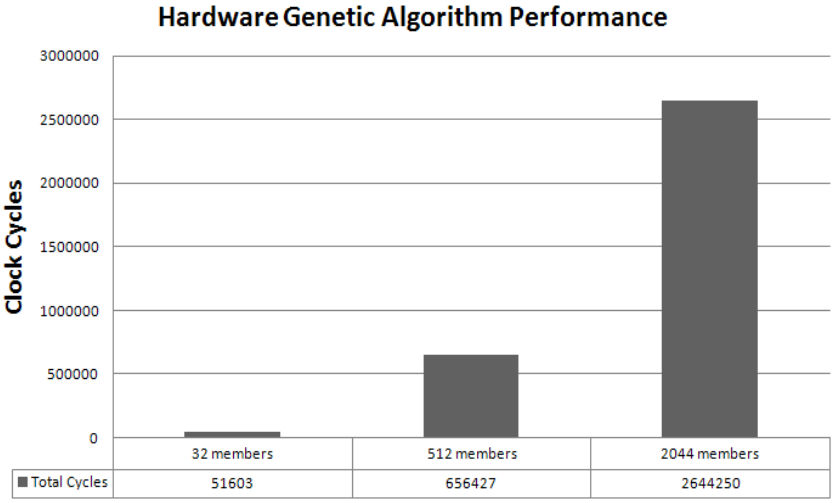

Fig. 5.13: Hardware Genetic Algorithm Performance

5.2 Reconfiguration Overhead

One of the most important tradeoffs of the partial reconfiguration technology is the reconfiguration overhead. This is the time needed for the reconfiguration process of an partially reconfigurable block. The hardware components that are used in the reconfiguration process are the Compact Flash and OPB ICAP. Both components introduce unacceptable delays for an real time applications reaching up to hundreds msec.

In addition to those problems, a recent research article [26] shows that the theoritical throughput of the OPB ICAP differs from the practical. After careful experiments they measured the following throughputs for OPB ICAP:

ICAP throuputs		
ICAP MODE	Practical	Theoritical
8 bit	32.4Mbit/s	0.75 Gbit/s

Tab. 5.1: ICAP throuputs

The reconfiguration timings of our design will be measured according to the results of a relative work [31] and the Virtex-II Pro FPGA user guide manual [29]. Firts we calculate the time needed from one partial bitstream to be loaded from the Compact Flash to the PowerPC BRAM. Second we calculate the time needed for the reconfiguration of one partial bitstream. Finally we summarize those times to conclude for the total reconfiguration time.

Concerning the CF the, each transaction with it is memory hungry and the SysAce controller performance deviates from the theoritical. The theoritical trhoughput is 8 MB/s but the measured practical throughput was 0,3 MB/s [31]. In our setup, for the dynamic reconfiguration procedure, we firstly load the bitstream from the CF into the PowerPC bram. The CF transaction time is:

Bitstream Size = 141728 Bytes = 0,000135162 MB

SysAce Throughput = 0.3 MB/s

Transaction Time = 45,054 msec

Afterwards, the PowerPC write the loaded bitstream to the ICAP. According to their timing results [31] for the reconfiguration of each frame, our calculated time is:

Frame Length = 206 (32-bit words) = 824 Bytes

Number of Frames = Bitstream Size / Frame Length = 172

Reconfiguration Time (8 first frames) = 75 msec

Reconfiguration Time of each following frame = 6ms

Reconfiguration Time (172 frames)= (8/8)*75 msec + (164/1)*6 msec

= 1.059 sec

From the above calculation we found that the total reconfiguration time is:

Total Reconfiguration Time = 45.054 msec + 1.059 sec = 1.103 sec

6. CONCLUSION & FUTURE WORK

This paper presents new approach towards HGAs with the application of RTR. It shows a way to create an autonomous self-reconfiguring system that supports a large variety of fitness functions. Due to the generality of the HGA architecture we had a performance disadvantage towards other instance specific HGAs. This is because the instance specific architectures utilized maximum hardware parallelization. Nonetheless, this trade-off is relatively small vs the advantages gained.

Furthermore, this work is part of a larger problem to be solved, such as VLSI optimization, image processing and others. We propose a general intergrated solver that doesn't require the pre-loading of all functions in the FPGA and has the ability to deal with different type of problems. In addition to this we demonstrate certain changes that can be made on the previous static HGA implementation [4] to not only increase its performance but also make it a dynamically reconfigurable general purpose HGA.

In the future, we can increase the reconfigurable module support of the Compact Flash bitstream library. This would integrate this system and supply the user with additional reconfiguration options. The user can select a bitstream from a larger variety of modules to download in the PRRs. This would increase the support of the problems the HGA can solve.

We can also evaluate changes in the fitness function during a single run, as it

is already supported in hardware.

Further acceleration of the selection process of the HGA to increase the throughput rate and acquire more competitive results to other implementation.

BIBLIOGRAPHY

[1] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, January 1989. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0201157675

[2] A. Baljit Singh, Arjan Singh, "Havrda and charvat entropy based genetic algorithm for traveling salesman problem," *International Journal of Computer Sience and Network Security, 2005.*, vol. Vol. 8, no. No. 5, pp. pp. 312–319, 2005.

[3] K. A. De Jong and W. M. Spears, "Using genetic algorithms to solve np-complete problems," in *Proceedings of the third international conference on Genetic algorithms.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 124–132.

[4] M. Vavouras, K. Papadimitriou, and I. Papaefstathiou, "Implementation of a genetic algorithm on a virtex-ii pro fpga," in *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays.* New York, NY, USA: ACM, 2009, pp. 287–287.

[5] S. D. Scott, A. Samal, and S. Seth, "Hga: a hardware-based genetic algorithm," in *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays.* New York, NY, USA: ACM, 1995, pp. 53–59.

[6] M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto, "Core allocation and

relocation management for a self dynamically reconfigurable architecture," *VLSI, IEEE Computer Society Annual Symposium on*, vol. 0, pp. 286–291, 2008.

[7] *OPB HWICAP*.

[8] G. Koonar, "A reconfigurable hardware implementation of genetic algorithms for vlsi cad design," Master's thesis, Computer Science University of Cincinnati, Cincinnati, USA, 2005.

[9] H. Emam, M. Ashour, H. Fekry, and A. Wahdan, "Introducing an fpga based genetic algorithms in the applications of blind signals separation," *System-on-Chip for Real-Time Applications, 2003. Proceedings. The 3rd IEEE International Workshop on*, pp. 123–127, June-2 July 2003.

[10] S. Narayanan and C. Purdy, "Hardware implementation of genetic algorithm modules for intelligent systems," *Circuits and Systems, 2005. 48th Midwest Symposium on*, pp. 1733–1736 Vol. 2, Aug. 2005.

[11] M. Tommiska and J. Vuori, "Hardware implementaion of ga," 1996.

[12] W. Tang and L. Yip, "Hardware implementation of genetic algorithms using fpga." The 47th IEEE International Midwest Symposium on Circuits and Systems, IEEE 2004.

[13] C. Aporntewan and P. Chongstitvatana, "A hardware implementation of the compact genetic algorithm," *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, vol. 1, pp. 624–629 vol. 1, 2001.

[14] A. I. K. Hossam E. Mostafa and Y. Y. Hanafi, "Hardware implementation of genetic algorithm on fpga," 21 National Radio conference March 2004.

[15] 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, 2001.

[16] J. R. Koza, F. H. Bennett, 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 '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1998, pp. 209–219.

[17] M. I. Heywood and A. N. Zincir-Heywood, "Register based genetic programming on fpga computing platforms," in *Proceedings of the European Conference on Genetic Programming*. London, UK: Springer-Verlag, 2000, pp. 44–59.

[18] S. Perkins, R. Porter, and N. Harvey, "Everything on the chip: a hardware-based self-contained spatially-structured genetic algorithm for signal processing," in *Proc. Of the 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES 2000*. Springer-Verlag, 2000, pp. 165–174.

[19] T. Lei and W. J.-x. Zhu Ming-cheng, "The hardware implementation of a genetic algorithm model with fpga," WO02 IEEE.

[20] M. F. So and A. Wu, "Hardware implementation of four-step genetic search algorithm."

[21] P. Graham and B. Nelson, "A hardware genetic algorithm for the travelling salesman problem on splash 2," in *Field-Programmable Logic and Applications*. Springer-Verlag, 1995, pp. 352–361.

[22] K. Glette, J. Torresen, and M. Yasunaga, "Online evolution for a high-speed image recognition system implemented on a virtex-ii pro fpga," Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007) 2007 IEEE.

[23] R. J. C. Barry Shackleford, Greg Snider, "A high-performance, pipelined, fpga-based genetic algorithm machine," 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

[24] P. Manet, D. Maufroid, L. Tosi, M. Di Ciano, O. Mulertt, Y. Gabriel, J.-D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, and V. La Barba, "Interactive presentation: Recops: reconfiguring programmable devices for military hardware electronics," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*. San Jose, CA, USA: EDA Consortium, 2007, pp. 994–999.

[25] G. Winter, J. Periaux, M. Galan, and P. Cuesta, *Genetic Algorithms in Engineering and Computer Science*. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[26] P. Manet, D. Maufroid, L. Tosi, G. Gailliard, O. Mulertt, M. Di Ciano, J.-D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, V. La Barba, P. Cuvelier, B. Rousseau, and P. Gelineau, "An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications," *EURASIP Journal on Embedded Systems*, vol. 2008, November 2008.

[27] *Early Acces Partial Reconfiguration Lounge*.

[28] *Planahead Tutorial 10.1*.

[29] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide v4.2*.

[30] Y. Tsoy, "The influence of population size and search time limit on genetic algorithm," *Science and Technology, 2003. Proceedings KORUS 2003. The 7th Korea-Russia International Symposium on*, vol. 3, pp. 181–187 vol.3, June-6 July 2003.

[31] A. Anyfantis, K. Papadimitriou, and A. Dollas, "Performance analysis of dynamic reconfiguration in modern integrated reconfigurable logic circuits."