A SELF-RECONFIGURING ARCHITECTURE SUPPORTING MULTIPLE OBJECTIVE FUNCTIONS IN GENETIC ALGORITHMS

Charalampos Effraimidis, Kyprianos Papadimitriou, Apostolos Dollas, Ioannis Papaefstathiou

Department of Electronic and Computer Engineering
Technical University of Crete
Chania, Greece
{ceufraimidis@isc, kpapadim@mhl, dollas@mhl, ygp@mhl}.tuc.gr

ABSTRACT

Genetic algorithms (GA) are search algorithms based on the mechanism of natural selection and genetics. FPGAs have been widely used to implement hardware-based genetic algorithms (HGA) and have provided speedups of up to three orders of magnitude as compared to their software counterparts. In this paper, we propose a parameterized partially reconfigurable HGA architecture (PPR-HGA). The novelty of this architecture is that it allows for the objective function to be updated through partial reconfiguration, and supports various genetic parameters.

1. INTRODUCTION

Genetic algorithms (GA) are techniques used to find exact or approximate solutions to optimization and search problems [1]. Despite the GA ability to provide good solutions to various problems, its algorithmic structure is simple. In Figure 1 each of the block modules performs a simple operation: (i) the *fitness* module performs the evaluation of the chromosome, (ii) the *sequencer* module randomly selects the chromosomes, i.e. an aspect of the model under study, and passes them to the selection module, (iii) the *selection* module decides which of the sequenced module should advance, and (iv) the *mutation* and *crossover* modules mutate and mate the selected chromosomes.

The need for hardware implementation of GAs arises from the overwhelming computational complexity of problems that cause delays in the optimization process of software implementations. The speed advantage of hardware and its ability to parallelize, offer great advantages to genetic algorithms to overcome those problems. Speedups of 1 to 3 orders of magnitude were achieved when frequently used software routines were implemented in hardware with Field Programmable Gate Arrays (FPGAs) [2]. However, those implementations were focusing on solving one specific problem due to the hardware resources constraints.

Over the past few years, many vendors have incorporated the Partial Reconfiguration (PR) technology to certain FPGAs. This technology provides great adaptability, as part of the device can be reconfigured on-the-fly without affecting the rest of the device. We present a parameterized partially reconfigurable HGA (PPR-HGA) that solves various problems by efficiently exploiting the device resources. As opposed to the prior state of the art, this architecture can load the requested objective function - also called fitness function - from an external device, and thus changing the target problem while overcoming the device area constraint problem. Furthermore, it has the ability to incrementally build its fitness function in order to reduce the total reconfiguration overhead. The contributions of this work are:

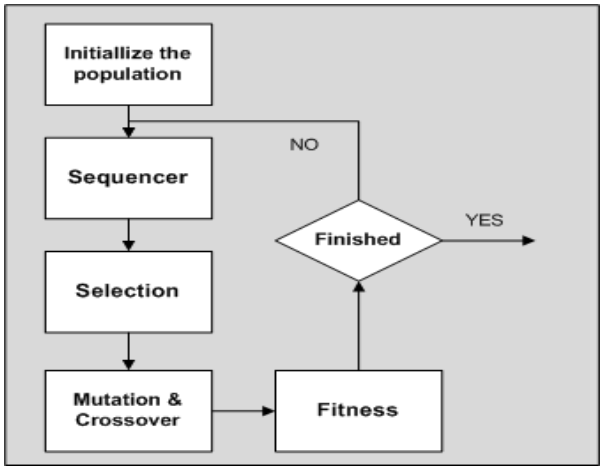

Fig. 1. Genetic algorithm flowchart

- Optimization of an HGA and parameterization of its features for maximum problem support
- Implementation of a reconfigurable mathematical unit to support multiple fitness functions
- A PRM architecture that reduces bus macro usage
- This work is the first to use partial reconfiguration on the GA problem

The rest of this paper is organized as follows: Section 2 presents previous works on HGA. Section 3 has the PPR-HGA architecture. Section 4 illustrates the modifications we made to the previous work on HGA concerning the static part [3, 4]. The reconfigurable HGA part is presented in Section 5. Section 6 has the experimental results, and Section 7 concludes the paper.

2. STATE OF THE ART

Based on the work by Scott et al. [2], Vavouras et al. implemented an HGA in several FPGAs [3, 4]. The design forms a coarse-grained pipeline and implements six different polynomial fitness functions. Emam et al. [5] proposed an HGA on non-linear adaptive filters for the purpose of blind signal separation. The performance was reported without the presence of a fitness function, claiming that the performance of this implementation is determined by its fitness function. The architecture is static, with fixed parameters and solves a low complexity problem. Tommiska et al. [6] designed an HGA with Altera Hardware Description Language (AHDL). Their architecture combines the simplicity, pipeline and structure of our HGA [3] with a more instance-based architecture. Their HGA parameters are fixed and so are their processing components.

The present work is based on our previous implementation [3, 4] which was proven to be very efficiently designed. It has a coarse-grain pipeline that provides a good throughput rate. The architecture is modular and the modules communicate through a handshaking protocol, thus increasing robustness.

3. PPR-HGA ARCHITECTURE

We used the Xilinx partial reconfiguration design flow and designed the PPR-HGA incrementally. This way the designer can partially modify the design, leaving the placement of the rest intact and thereby shortening design iterations, and maintaining the required performance.

The design was implemented on a Virtex-II Pro FPGA. Figure 2 shows the components of the design as connected in the top level module: the system, the HGA and the DCM. The system is composed of the PowerPC (PPC) and four peripherals: the Internal Configuration Acces Port (ICAP), the Compact Flash (CF) controler peripheral, the serial port controler, and the OPB-DCR - which through the OPB2DCR bridge implements the interface between the On-Chip Peripheral Bus (OPB) and the Device Control Register (DCR). The ICAP enables the PPC to read and write partially the FPGA configuration memory. The PPC reads the stored bitstreams from the Compact Flash and passes them to the ICAP peripheral, which in turn reconfigures the device using a read-modify-write mechanism. The HGA is composed of two parts, the static HGA and the reconfigurable HGA, which are discussed in the next Sections.

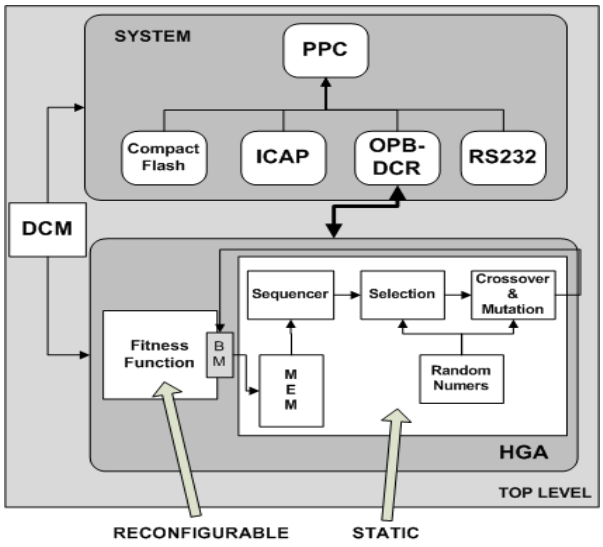

Fig. 2. Top level architecture

4. STATIC HGA

Starting the optimization process, the first step was to increase the HGAs supported parameter range which was initially discussed in [4]. The goal was to fully parameterize the algorithm attributes in order to support the maximum possible number of parameters for the target device features [7]. This was not straightforward, since the increased parameters introduced a considerable degradation in the clock frequency. We mitigated this by code optimization. We found that the part of the algorithm that caused the clock frequency reduction was the comparison of the member's fitness value with a threshold set by the selection module. We then replaced this comparison with a subtraction and a 1-bit comparison, used to identify whether the subtraction result was positive or negative. An extra Finite State Machine (FSM) state was added to handle the operations.

After the design parametrization, shown in Table 1, larger parameter values were tested, i.e. up to 10-bit members with 36-bit fitness function width. With these parameters the number of distinct members was increased without increasing the population size. This is because with 4-bit members a maximum of 16 different members is represented, which entails duplication of the same members among a population of 32 members; with 10-bit members this duplication is avoided. This increase in the distinct 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 [8].

Finally, additional modifications were applied to the mutation/crossover module and to the selection module. Concerning the selection module, we modified the selection pro-

Table 1. PPR-HGA attributes

Attribute	Value Range
Memory	4096 × 46-bits
Max. population size	2044
Member width	10-bit
Fitness function width	36-bit
Sum of fitness width	46-bit
HGA max. clock frequency	127 MHz

cess in order to accelerate its decision. The new selection criterion is the average fitness value of the population members. The selection module 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 randomly a member.

Figure 3 has the static HGA modular architecture. The output of the Crossover and Mutation unit is routed to the reconfigurable mathematical unit that implements the Fitness Function, and the latter's output is routed to the Memory.

5. RECONFIGURABLE HGA

5.1. Partial Reconfiguration Advantages

The PR technology allows the reconfigurable HGA to support more fitness functions. This offers better utilization of the device features and better speed performance to the HGA. By allocating the fixed amount of resources shown in Table 2 we can solve many different problems, without having to place them simultaneously within the same device. This also offers a speed advantage as a result of the fact that only the operating functions are routed, hence reducing the logic and utilized area of the device and minimizing the relative delays.

Another advantage of PR is power consumption optimization. A reduction in the static leakage of the components can be achieved by loading blanking bitstreams to the

Table 2. Resource utilization in a XC2VP30 FPGA

Resource Type	Required	Percentage
LUT	10118	36.9 %
FF	9235	33.7 %
SLICE	5144	37.5 %
MULT18x18	16	11.7 %
RAMB16	51	37.5 %

corresponding PRRs and keeping only the functional components instantiated.

5.2. Mathematical Unit Implementation

We replaced the static HGA fitness function of our previous implementation [3, 4] with a reconfigurable mathematical unit. This unit comprises 4 partially reconfigurable regions (PRR) each of which occupy the resources of Table 3. We were constrained to the specific number of PRRs because the Virtex-II Pro FPGA doesn't support 2D reconfiguration. It is mandatory to assign two different PRRs on the same column. In each region a partially reconfigurable 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 to 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 form a 64-bit number.

We implemented three PRMs for each PRR, a multiplier, an adder and a zero unit. All the PRRs statically route their output to an ADD unit. The latter summarizes the results of the functional PRMs. The first column of PRRs also routes their output to the next PRR, as shown in Figure 4. This sequential processing produces complex mathematical equations. Considering its performance, the mathematical unit is pipelined and its performance is affected by the downloaded mathematical units. Each PRM needs 2 to 4 cycles(1 cycle for 1 to 32-bit input and 2 cycles for 33 to 64-bit input) to

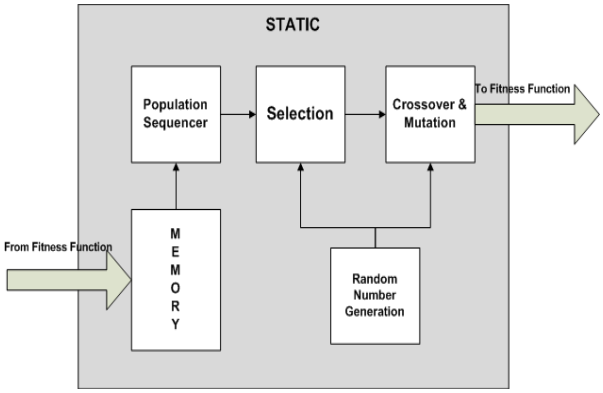

Fig. 3. Static HGA

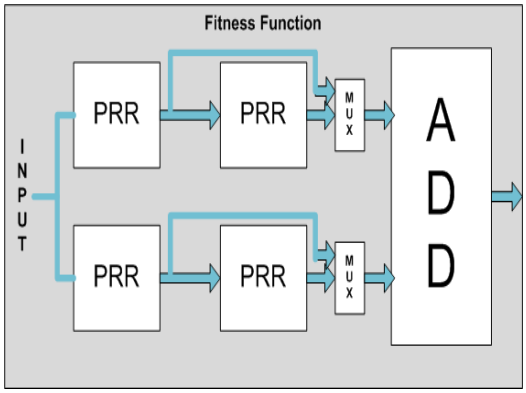

Fig. 4. Mathematical unit architecture

Table 3. PRR resources allocation

Resource type	Allocated
LUT	1216
FF	1216
SLICE	608
MULT18X18	4
RAMB16	4

create the I/O numbers plus a few cycles for the function execution. For example the achieved throughput for 32-bit input, 64-bit output and 4 cycles for the function execution would be: 64-bit/((1+2+4)cycles $\times$ 127MHz) = 719 Mbps.

Each PRM contains three modules: The *inputdata* module which receives 32-bit numbers and gradually assembles them to form the input number. The *function* module which evaluates the input numbers, and the *outputdata* module which segments the output number of the function into 32-bit numbers, and then passes them to the next module. The synchronization of the PRMs is achieved through the *write* signals. When the write signal is asserted the next module stores the 32-bit input data. Figure 5 shows the PRM structure.

The inputdata and outputdata modules allow for the reduction of the required bus macros that implement the communication between the PRRs. The bus macros presence in the PRRs constrain their shape, as well as the designer's options. A reduction in the bus macros provides greater flexibility to the floorplanning process and a better routing.

The design choices of the mathematical unit have been made with respect to generality. Depending on the modeling of the problem, only the necessary fitness evaluation modules need to be designed and downloaded.

6. EXPERIMENTAL RESULTS AND COMPARISON

The experimental results of the new PPR-HGA are compared to the previous HGA implemented in the Virtex-II Pro FPGA [3, 4]. The new architecture executes at 127 MHz, and we tested it for member width equal to 10-bits, population size 32-bits and simple fitness functions, i.e. x, $2x$ and x^2. In comparison with the initial HGA this architecture achieved better results. The GA not only converges to the global optimum faster but the required clock cycles are greatly reduced. The average convergence point of the PPR-HGA for the above fitness functions is 7 generations as opposed to an average of 22 generations needed for the previous HGA. There is also a clock frequency improvement which accounts for the decrease of the average execution time:

$$previous\ HGA\ exec.time = 218000 cycles \div 107 MHz = 2ms$$
$$PPR{-}HGA\ exec.time = 178000 cycles \div 127 MHz = 1.4ms$$

Therefore, the new architecture offers a speedup of 1.4.

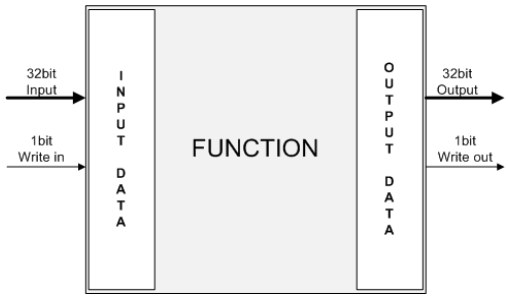

Fig. 5. PRM structure

7. CONCLUSIONS AND FUTURE WORK

This paper presents a new approach towards HGAs with application of PR. 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 might have a performance disadvantage as compared with other instance-specific HGAs. This is because the instance-specific architectures utilize maximum hardware parallelization. In the future, we can increase the reconfigurable module support of the Compact Flash bitstream library which will supply the user with additional reconfiguration options.

8. 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. Seth, "Hga: a hardware-based genetic algorithm," in *Proceedings of the ACM third international symposium on Field Programmable Gate Arrays*.

[3] M. Vavouras, K. Papadimitriou, and I. Papaefstathiou, "Implementation of a genetic algorithm on a virtex-ii pro fpga," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*.

[4] ——, "High-speed fpga-based implementations of a genetic algorithm," in *Proceedings of the IEEE International Symposium on Systems, Architectures, Modeling and Simulation, to be presented*.

[5] H. Emam, M. Ashour, H. Fekry, and A. Wahdan, "Introducing an fpga based genetic algorithm 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.

[6] M. Tommiska and J. Vuori, "Hardware implementaion of ga," Aug 2006.

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

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