CIRCUIT SYNTHESIS EVOLUTION USING A HARDWARE-BASED GENETIC ALGORITHM

Rami Abielmona, Voicu Groza

University of Ottawa
School of Information Technology Engineering
161 Louis-Pasteur St., CBY-A516
Ottawa, ON, K1N 6N5, Canada

ABSTRACT

In this paper, we propose a scheme based on a hardware implementation of a genetic algorithm, to evolve the minimized logic solution of a defined input function. The minimization will be one of resource usage, more precisely of **look-up tables (LUTs)**. The design aids in the difficult issue of technology mapping, as well as multi-level logic synthesis. The approach undertaken in this research involves intrinsic hardware evolution, where the circuit solution is evolved ”online”, and the output is a minimized structure of the circuit. Our architecture is outlined and briefly discussed, while our current results are presented and analyzed.

1. INTRODUCTION

In the age of cloning and genome mapping, we have reached a point in time, where with a little research and innovation, we can achieve things previously thought impossible. With today's advances in reconfigurable computing, it is not beyond the realm of our imagination that circuits, that previously required a large knowledge database as well as human resources, are being ”evolved” by machines. These circuits are simplistic in nature, compared to the man-made complex structures, but do provide us with a solid foundation to expand our research upon. Evolvable hardware has become a rapidly-growing field, in which new discoveries are being made at an astounding pace.

The advent of such a technology has enabled us to think of new ways to approach the research and development of both hardware and software systems. The main design issues have been shifted from a structural view to a functional view. The internals architecture is being overshadowed by the overall functionality of the system. To that effect, the designer has the more daunting task of thinking of new functionalities, without having to pay too much attention to the intricacies of the system architecture. This novel approach is not as simple as it is portrayed here. A lot of work has to be done to ensure that the evolved solution matches the characteristics that make the designer solution so effective, be it robustness, scalability or fault tolerance, to name a few. Other major advantages, such as speed of execution or resource utilization are considered the main drivers of success for designer solutions.

The main problem faced when designing new chips these days is the shortage of available silicon space. To bypass this issue, an extension of the underlying instruction set can be implemented by providing the programmer with an *aide* to the main processor or *Central Processing Unit (CPU)*. The assembly-level programmers are thus provided with a substantially greater domain of instructions, allowing them to gain access to a myriad of deemed integral operations. This solution is a typical one these days, as *coprocessors* provide that aforementioned extension. Another feasible solution involves re-studying the design that is being attempted. This requires the designers to repeat several tasks which have already been performed for the original design, but that need to be optimized in a new and previously unforseen manner. This optimization will require a new methodology for *logic synthesis*, when speaking of general purpose chips.

The main goal of the research reported in this paper is to propose a scenario that involves the introduction of an innovative logic minimization scheme. The use of artificial intelligence to produce a minimal (according to some function) solution is not new, but in this research, artificial intelligence will be used to map a design to a technology, while minimizing the resources utilized. The scheme will be referred to as **Genetic Algorithm Synthesis (GAS)**.

GAS, as the name precludes, is a synthesis scheme based on a standard genetic algorithm. It is accompanied by an inherently powerful advantage: **multi-level multi-output (MLMO)** logic minimization. The latter feature has been a fuzzy topic in the circuit design milieu for a long time. This research attempts to extend the work of [1], [2], [3] and others on this topic, by offering a hardware-oriented minimization mechanism. Along with MLMO minimization, GAS's inputs, consisting of truth tables and GA parameters, pro-

vide the designer with an easy interface to programmable logic. In other words, GAS has a simple system requirement: "To evolve a minimal working solution for a defined truth table".

2. PRELIMINARIES

Developed by John Holland, his colleagues and students at the University of Michigan, GAs are search techniques modeled after natural selection, including the genetic operators that play an important role in the fashion that we survive in our environment [4]. GAs are stochastic algorithms with very simple operators that involve random number generation, and copying and exchanging string structures. The aforementioned operators act on a *population of members*, represented by *chromosomes*. Members are mated by the *crossover* of their appropriate chromosome, the latter undergoing a *mutation* operator after crossover. The next *generation* of members is selected from the current one according to how well (*fitness value*) the members react to their present environment [5]. GAs have been efficient in their "evolutionary" runs because as new generations are made of older ones, one can easily select the members that are fairing quite adequately in the environment they have been placed in. The crossover operator ensures that the "good" chromosomes have more children in the generations to come, while the "bad" ones have less or no children at all. The mutation operator provides a stabilizing random input of error, to emulate the corresponding phenomenon that occurs in nature [6].

Evolvable hardware is a term coined by Hugo De Garis in 1992 [6]. *Evolware*, as it is often referred to, consists of reconfigurable hardware that can undergo a number of evolutionary cycles, producing a suitable solution to the application at hand [7]. The introduction of FPGAs has permitted the use of GAs in *very large scale integration (VLSI)* chips and hence, the creation of evolware.

The use of reconfigurable hardware for the design of a GA was seen in projects such as [8], [9] and [10]. In Stephen Scott's behavioral-level implementation of a GA [8], the targeted application was the optimization of an input function. In [9], a GA was designed and implemented on a PLD, using the *Altera hardware description language (AHDL)*. In [10], a number of GAs were designed and implemented in a text compression chip.

The first major breakthrough in the field of intrinsic hardware evolution was accomplished by Adrian Thompson in 1996, when he successfully used completely unconstrained evolution techniques to configure the logic inside of a FPGA [11]. The findings have propelled a wide area of research in intrinsic (as well as extrinsic, or off-chip) evolvable hardware methodologies [6], [12] and [13].

Other related work includes the work done on logic minimization through the use of GAs. Louis [1] has made use of genetic algorithms on design structures to attempt to solve the combinational circuit design problem. His use of a *masked crossover* genetic operator proved to be very advantageous, but the GA itself became less of a powerful search technique. Coello et all [2] extended the research done by Louis, and used a GA to automate the design of combinational logic circuits. They modeled logic circuits with matrices, with each element representing a gate and the inputs from previous elements. The gate was selected from a list of five fundamental gates : AND, NOT, OR, XOR and WIRE, where the first four are self-explanatory, and the last representing a physical wire (and thus, the absence of a gate). Logic minimization was thus solved by a constraint on the matrix elements: maximizing the WIRE element, and hence minimizing the total number of gates necessary for the implementation of the circuit. Koza [3], on the other hand, used genetic programming to assist him in the design of combinational circuits. The main goal of Koza's research was the successful generation of the circuits, and not their minimization, thus does not completely apply to the task at hand.

In our current work, we combined the topics to create a scheme, which attempts to minimize the logic resources used within a *programmable logic device (PLD)*, with particular emphasis on the design of a *register transfer logic* level hardware implementation of a GA.

3. SYSTEM ARCHITECTURE

The overall system is composed of two PLDs, appropriately called the *optimizer* and the *host* (refer to figure 1). The optimizer is where the input function logic is "evolved", whereas the host is where the input function solution is actually instantiated and tested. The host will be included in the design as a sub-block of the optimizer (please see the ECLBs discussion below). The optimizer's main objectives are to interface with the CPU and the host, start off the **Hardware-Implemented Genetic Algorithm (HIGA)** on its evolutionary run, store the solutions to each input truth table and evaluate the fitness of each candidate solution. The host is used in order to instantiate the functional units that each logical function needs for execution. Hence, the optimizer performs the evolutionary work, writes to memory the best solution and instantiates the solution in the host, the latter being an originally empty PLD. As an implementation note, all PLDs in this research are actually *Field-programmable gate arrays (FPGAs)*.

The optimizer is made up of the following (refer to figure 2): a HIGA subsystem, a memory subsystem, an **Evolvable Configuration Logic Blocks (ECLBs)** subsystem and a CPU interface subsystem.

HIGA This subsystem provides the optimizer with the "nu-

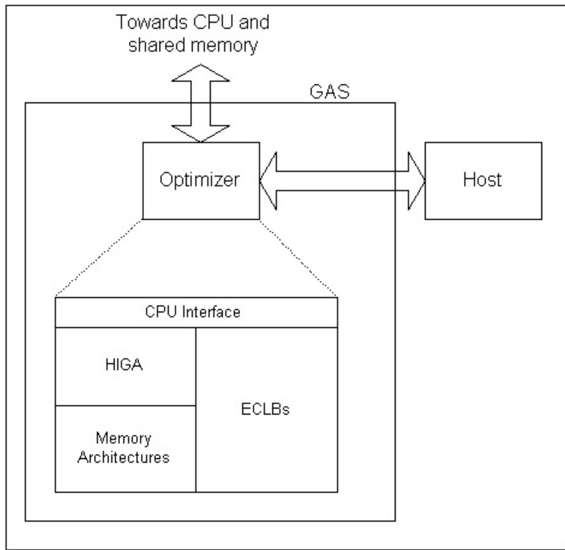

Fig. 1. System Diagram

cleus", or brain, of the operation. According to the input function (represented by a truth table) presented to the optimizer, the HIGA begins to run in order to determine, through intrinsic hardware evolution, which "chromosome" (or solution) is best suited for the operation. The HIGA is implemented in the *very high speed integrated circuit hardware description language (VHDL)* and contains the following modules: Random Number Generator (RNG), Input Instantiation, Selection, Crossover, Mutation, and Fitness modules.

Memory This subsystem is what is referred to as the **"hardware cache" (hcache)** of the system. It sits on the optimizer as memory cells that, upon optimizer programming, could be loaded from a ROM device which contains the previously stored circuits of logic functions. This subsystem stores the "solutions" to all the already-encountered truth tables, and thus acts as a common storage facility for designs that could be utilized in the future. This is initially external memory, as opposed to what is shown in figure 2.

ECLBs These logic blocks are empty on boot up, but are used to evaluate the chromosomes in order to provide the HIGA subsystem with feedback on how well the candidate chromosome performed in the current environment. This is initially an empty logic device, as opposed to what is shown in figure 2. This subsystem is viewed as the current replacement for the host. It has the same functionalities as the latter, but also acts as an area for the actual evolution of each circuit.

CPU Interface This subsystem provides the interface to the CPU. It consists of data, control and status registers, as well as a myriad of **special purpose registers (SPRs)**. The interface integrates the programmer's model of the optimizer and the host communication functions, in order to easily and successfully allow for the hardware/software co-execution and control.

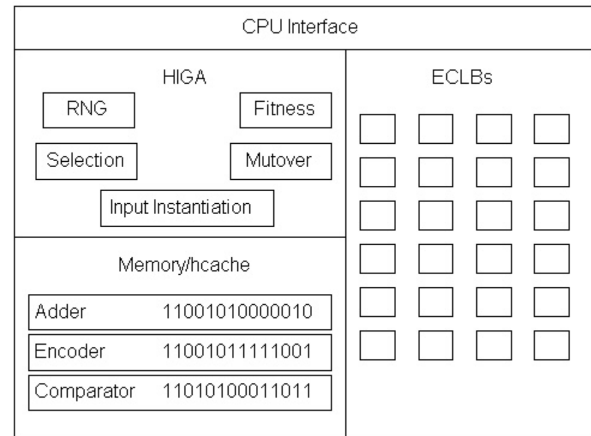

Fig. 2. Optimizer module design

As seen in figure 3, the HIGA receives its inputs from on-board shared memory. Currently, the user manually enters the input truth table, as well as the input GA parameters (maximum population, crossover and mutation probabilities, etc...). The HIGA proceeds to iteratively evaluate chromosomes in the ECLBs subsystem. This real-time, on-board evaluation constitutes the "intrinsicity" of this project. The output of the HIGA is a best-fit function-specific solution, which is stored in the memory/"hcache" subsystem. The stored solution can then either be kept in the hcache or moved to external memory, for future reference.

As previously mentioned, the HIGA subsystem is composed of the following modules:

Random Number Generator This module periodically provides the input instantiation, selection, crossover and mutation modules with pseudo-random numbers.

Input Instantiation This module provides the output, of each truth table input combination, stored in external memory. This output feeds into the fitness module for comparison to the experimental output. This module accesses the on-board memory to read the input truth table values.

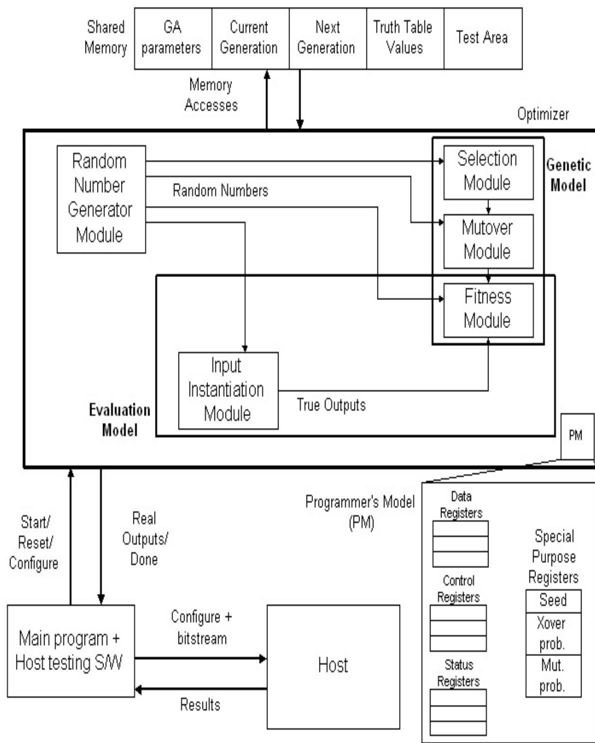

Fig. 3. Overall architecture

Selection This module accepts a member and a random number as inputs. It uses a hybrid algorithm (roulette wheel selection [5] and tournament selection [4]) to decide whether the member is to be selected for the next generation or not.

Crossover/Mutation These modules accept members as input, and use the input pseudo-random numbers to decide whether crossover/mutation are to be performed upon the individuals or not. Typical single-point crossover and single-bit mutation are used.

Fitness This module is used to evaluate the fitness of each individual. It accepts the ideal output from the input instantiation module, compares it to the experimental output generated from the ECLBs subsystem, and calculates the fitness of that individual. The output, consisting of a member/fitness combination as well as the address to write the structure at, is passed on to software for further processing. This module also dubs as a control module, as it handles the transfer of individuals between the modules and memory. It stores the new individuals in memory and passes them on to the selection module.

To summarize the operation of the HIGA, the truth table and GA parameter inputs are stored in shared memory. The initial generation of chromosomes is generated pseudo-randomly, and stored in shared memory. Each member undergoes the three major genetic operators, processed in the selection, crossover and mutation modules, and is either wholly or partially picked to move on to the next generation. Once picked, the new member is evaluated in order to receive a fitness value, which is compared to the ideal output produced by the input instantiation module. The modules coordinate all the transfers between each other, thus ensuring a successful sequential operation.

4. RESULTS

The HIGA has been completely designed and developed. The data and control paths of the modules have been designed and functionally simulated. For shortage of space, the paths will not be introduced here, as an extensive discussion must follow for clarity. The current remaining work involves the overall architecture of the system, including the real-time evaluation of each chromosome in the ECLBs area, and its eventual feedback into the fitness module for assignment. Presently, the random number generator module generates an input train that feeds the input instantiation module, and random numbers that feed each of the selection, crossover, mutation and fitness modules. A chromosome is fed into the selection module. The latter performs the selection algorithm:

1. Select a member using the roulette wheel selection algorithm;

2. Select another member using the roulette wheel selection algorithm;

3. Compare a random number r to a preset parameter k, and if r is smaller than k, then select the member with the higher fitness, otherwise select the member with the lower fitness;

4. Repeat this procedure until the population is full.

Once a member has been selected, it is passed on to the crossover module. The latter awaits the arrival of two members from the selection module, and proceeds to perform single-point multiple-bit crossover on both members. The results are forwarded to the mutation module, where single-bit mutation is performed. The crossover and mutation modules perform their operations according to the GA probabilities which are stored in shared memory, and have actually been recently merged into one module, called the **mutover** module, for implementation efficiency purposes. Finally, The fitness module receives the member and forwards it to the controlling software, along with an assertion of a signal

to indicate that a member is ready for evaluation. The current state of the system does not allow us to respond back with real-time data as both the software and the feedback paths are being developed.

In these preliminary designs, the software being utilized consists of an innovative set of APIs, developed at Xilinx Inc.[11], labelled *JBits*. This tool provides the designer with a fast and simple interface to the Xilinx Virtex family of FPGAs. The reconfiguration aspect of this research dictates the use of a tool that allows for fast reconfiguration times, as well as an ease and simplicity of use. Both aspects are provided by JBits. The controlling software will have to interface with JBits, as the latter will only be used in the "live" phase, where the bitstream is actually being configured on the device. For that, the *Java Native Interface (JNI)* will be utilized to establish the communications path between the controlling software (which interfaces to the optimizer) and JBits.

A GA was designed, implemented and tested in the C high level language. A roulette wheel selection algorithm was used to select a member for the next generation. Single-point crossover and single-bit mutation were used, along with a simple fitness function. The latter consisted of obtaining an optimal or near-optimal solution to a linear equation. The purpose of this research is to set the foundation for further exploration of the simple GA in software. The undertaken research (GAS) will be compared to a modified version of the GA in software.

Upon successful completion of the system (as seen in figure 3), the user will be able to input the initial GA parameters, as well as the truth table to be implemented, and the corresponding optimal, or near-optimal, circuit solution will be stored in the shared memory for the user to access.

5. NOVELTIES AND APPLICATIONS

This is an RTL-level hardware implementation of a GA, as well as one of the few attempts to combine evolware with traditional hardware as part of a complete system. The interaction between the two forms a powerful hybrid of systems that can be easily expanded (ease of scalability and parallelism). The research falls under the third *generation* of evolvable hardware classes, according to Tomassini and Sipper [14]. All of the operations of the GA, including the fitness evaluation are performed "online" in hardware. The research also introduces another hidden novelty: the hardware cache. The hcache subsystem stores the circuit solutions of numerous logical functions.

The applications for this design include MLMO logic minimization, discussed early on in this paper. According to John Wakerley [15], it is impractical to deterministically find a minimal cover of variables, and thus, heuristic approaches have to be undertaken, such as Espresso II [15] for example. Heuristic methods, by definition, are ones that use a solution of several to feed into the next stage of the algorithm [16]. Thus, GAs are heuristic by nature, no pun intended, and are used as an approach to MLMO logic minimization. GAS also attempts to combine GAs and functional decomposition in order to introduce a novel logic synthesis method for LUT based FPGAs. The approach searches for functions that are useful in the realization of other functions, thus allowing for sharing of common functions.

The overall system is also suited for many additional applications, one of which is the development of a system whose main advantages are:

I the provision of a massively parallel structure, composed of many solution-searching areas for predefined functions;

II the inherent property of adaptation, that allows the system to change its optimization goals according to the provided cost function; and

III the fault-tolerance issue, in that most of the deemed non-working solutions, or subfunctions for that matter, are progressively eliminated through the evolutionary runs.

GAS is targeted for use in all hardware chip design projects. The applications of the chip-based systems are numerous, ranging from robots used for chip layout and fabrication, to VLSI circuits used for telecommunication systems.

6. CONCLUSIONS

GAS provides a test bed for evolutionary circuit design. Once the optimizer has been completely implemented and tested, then it is trivial to incorporate a PLD in order to realize the entire system. This combination is the epiphany for reconfigurable computing platforms, because of its ability to implement any logical function at any point in time. Upon the correct modeling of the overall system, the user is able to input this model into GAS, and receive a minimized logic circuit, without having to be concerned with implementation or minimization algorithms or details. GAS also distinguishes itself by the amount of functionality that is built into hardware, instead of software, allowing for faster reconfiguration and processing times.

Contrary to current methodologies, the design power, with the help of the host, now resides in the software designer's hands, as the hardware is actually a tool that can be modified to fit the designer's needs. All that is required from the software designer in order to harness that power is, a correct model of the new functionality to start the evolutionary run, and a working host to evolve the circuit.

7. REFERENCES

[1] Sushil J. Louis and Gregory J. Rawlins, "Using genetic algorithms to design structures," Tech. Rep. 326, Computer Science Department, Indiana University, Bloomington, Indiana, Feb. 1991.

[2] Alan D. Christiansen Carlos A. Coello Coello and Arturo Hernandez Aguirre, "Towards automated evolutionary design of combinational circuits," .

[3] John R. Koza, *Genetic Programming. On the Programming of Computers by Means of Natural Selection*, The MIT Press, 1992.

[4] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company, Incorporated, Reading, Massachussetts, 1989.

[5] Melanie Mitchell, *Introduction to Genetic Algorithms*, MIT Press, Cambridge, Massachussetts, 1996.

[6] Hugo De Garis, "Evolvable hardware: Printciples and practice," Last viewed on September 20, 2000, http://www.hip.atr.co.jp/ degaris/papers/CACM-E-Hard.html.

[7] T. Gordon, "Introduction to evolvable hardware," Last viewed on September 20, 2000, http://www.cs.ucl.ac.uk/staff/T.Gordon/ehwnote.html.

[8] Stephen Donald Scott, "Hga: A hardware-based genetic algorithm," M.S. thesis, University of Nebraska, August 1994.

[9] Tommi Rintala, "Hardware implementation of ga," Last viewed on September 20, 2000, http://www.uwasa.fi/cs/publications/2NWGA/node60.html.

[10] Loring Wirbel, "Compression chip is first to use genetic algorithms," *Electronic Engineering Times*, p. page 17, December 1984.

[11] Xilinx Incorporated, *The Programmable Logic Data Book*, San Jose, California, 2000.

[12] Isamu Kajitani Tetsuya Higuchi, Masaya Iwata and Masahiro Murakawa, "Hardware evolution at gate and function levels," .

[13] Didier Keymulen Hidenori Sakanashi, Masaya Iwata and Masahiro Murakawa, "Evolvable hardware chips and their applications," .

[14] Marco Tomassini and Moshe Sipper, "An introduction to evolvable hardware," Last viewed on January 25, 2001, http://evonet.dcs.napier.ac.uk/evoweb/evonews/news3/ehard.htm.

[15] John Wakerley, *Digital Design Principles and Practices*, Prentice Hall, 1994.

[16] *The American Heritage Dictionary of the English Language*, Houghton Mifflin Company, 3rd. edition, 1996.