

Low Cost Platform for Evolvable-Based Boolean Synthesis

César Pedraza Bonilla Facultad de Ingeniería de Telecomunicaciones.
 Universidad Santo Tomás - Bogotá.
 cesarpedraza@usantotomas.edu.co Carlos Iván Camargo Facultad de Ingeniería
 Universidad Nacional de Colombia - Bogotá.
 cicamargoba@unal.edu.co

Abstract—Evolutionary algorithms are another option for combinational synthesis because they allow for the generation of hardware structures that cannot be obtained with other techniques. This paper shows a parallel genetic programming (PGP) boolean synthesis implementation based on a low cost cluster of an embedded platform called SIE, based on a 32-bit processor and a Spartan-3 FPGA. Some tasks of the PGP have been accelerated in hardware and results have been compared with an HPC implementation, resulting in speedup values up to approximately 180.

Index Terms—Embedded systems, Evolutionary algorithms, boolean synthesis, cluster.

I. INTRODUCTION.

One of the main aims in combinational synthesis consists of finding compact boolean expressions in the sum of products (SOP) form with the smallest possible number of variables and terms. Boolean algebra offers a way to find compact expressions, but this process depends on the designer's experience, resulting in non-optimal or inadequate expressions. There are other techniques available for combinational synthesis such as the Karnaugh maps, the Quine-McCluskey algorithm, the Reed-Muller algorithm, etc. In general terms, these algorithms have disadvantages like exponential complexity, lack of restrictions management, and multiple solutions. As an alternative to the traditional design of combinational circuits, some authors have proposed bio-inspired techniques based on simple genetic algorithms (SGAs), variable-length genetic algorithms (VGAs), tree-based genetic programming (GP), simulated annealing, ant colony algorithms and others. These strategies allow to create combinational blocks that cannot be obtained with the traditional methods, and to add some restrictions to the design such as delay, area, etc. These designs have a very low limited number of variables [1] and they are mainly oriented to obtain a few basic structures.

In order to use parallel genetic programming (PGP) it has been developed an FPGA cluster-based architecture to solve the combinational synthesis problem on-chip. The fitness coprocessor unit (FCU) on each FPGA helps to accelerate the convergence of the algorithm, as well as provide an appropriate support for 12-variable synthesis problems. The success of the system is mainly due to the capability of evaluating a chromosome in the FPGA through a virtual LUT-oriented architecture without using high-latency partial reconfiguration

techniques, and determining the fitness value for an individual faster than other references.

II. GENETIC PROGRAMMING TO COMBINATIONAL SYNTHESIS

This section describes some of the most important aspects of the evolutionary algorithm for the combinational synthesis problem, such as chromosome representation, the fitness problem and the genetic operators.

A. Chromosome representation.

The codification is the way a logic circuit is represented using a bit array in order to be managed in the evolution process [2]. This representation must be able to represent all the different solutions to the problem, also the crossing and muting operators should not generate unreal individuals, and it must cover all the solution space so that the search is really random. There are different ways of representing combinational hardware for a genetic algorithm: tree-based representation in 2-D, PLD-like structures and cartesian representation are some of them [3] [4] [5]

The 2-D tree representation is appropriated for implementing parallel systems because it allows splitting the chromosomes for balancing the computational load [6]. Figure 1 shows the selected cell-based structure representation. Each cell has 3 functions f and 4 input variables v coded in binary. This representation allows for the addition of more cells to represent larger circuits. It must be mentioned that the chromosome length has to be variable because the length of the solution to the synthesis problem is unknown a priori.

B. Fitness function.

Finding the appropriate fitness function is important because it is responsible for quantifying the way a chromosome or individual meets or does not meet the requirements.

$$fitness = \omega_1 \cdot \left[\sum_{j=1}^m \sum_{i=1}^n Y(j, i) - X(j, i) \right] + \omega_2 \cdot P(x) + \omega_3 \cdot L(x) \quad (1)$$

In equation 1 the fitness function for our GA is shown. Constants ω_1 , ω_2 and ω_3 are used for establishing the weights

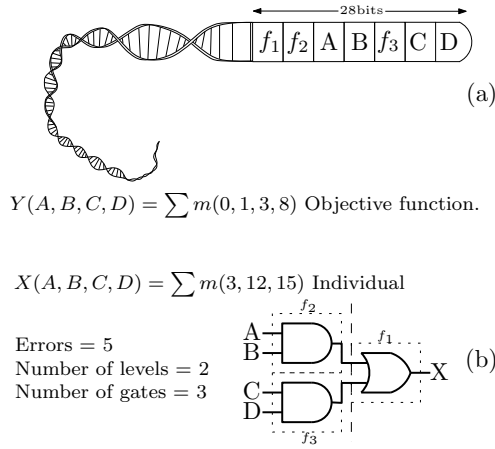


Fig. 1. Cell-based structure representation.

of each of the parameters that will determine the fitness function. The double-summation term calculates the number of coincidences of the individual X for all the possible combinations at the output with the target functions Y . The $P(X)$ function is used for calculating the number of logic gates of a chromosome taking into account some of the *introns* or segments of the genotype string that will not have any associated function and that do not contribute to the result of the logic circuit that they represent. The function $L(X)$ is used for determining the number of levels the circuit has, or in other words the number of gates that the critical path crosses. The m constant means the number of outputs in the circuit and n the number of possible combinations of inputs in the circuit.

C. Genetic operators.

The selection operator is responsible of identifying the best individuals of the population taking into account the exploitation and the exploration [6]. The first one allows the individuals with better fitness to survive and reproduce more often, and the second one means searching in more areas and making it possible to find better results. In the other hand, the mutation operator modifies the chromosome randomly in order to increase the search space. It can change: 1) an operator or variable and 2) a segment in the chromosome. Both are executed randomly and with a certain probability. A variable mutating probability during the execution of the algorithm (evolvable mutation) [7] is more effective for evolvable systems. Finally, the crossing operator combines two selected individuals for obtaining two additional individuals to add to the population. A crossing system with one or two crossing points randomly selected has been implemented because it is more efficient for evolvable systems [8].

III. COPYLEFT SIE PLATFORM.

The hardware project copyleft SIE [9] allows the generation of commercial applications under the Creative Commons BY-SA license [10], which allows the distribution and modification of the design (including commercial applications), with

only the requirements that the derived products be under the same license and that the proper credit be given to the author of the original work. This is what defines the base of the *hardware copyleft* products.

A. Hardware copyleft.

Since they are inspired by the FOSS movement, the *hardware copyleft* devices share the same philosophy [11], and they are the perfect complement. The requirements for a HW devices to be reproducible and modifiable are: availability of the schematics and files of the printed board circuit, using the proper format for opensource tools; the toolchain for compilation and debugging to develop applications; the source code of the bootloader, the tool that loads such a program in the non-volatile memory, the filesystem, and the applications; complete documentation showing how it was designed, built, how to use it, design applications, and also tutorials that explain the functionality of the different components.¹. Additionally, you must have a way to build and assemble this hardware, which is the main difference between free software and free hardware. This is a big difference compared to the free software initiative, for which there is no investment required to modify and build an existing project. This is the reason why there can be different levels of freedom, since a project that uses expensive and hard-to-reach components would limit its reach to a determined market.

B. SIE Platform.

The platform is composed by (Figure 2) a System-On-a-Chip processor and a FPGA. The SoC is a MIPS processor (Ingenic JZ4725) running at 400MHz with peripherals that allows to control: a 2GB NAND memory for file storage, a 32MB SDRAM, a serial communication channel (UART), micro-SD memories, I2C port, 3-inch color LCD display, 2 input and 1 output audio stereo interfaces and 2 analog inputs. The XC3S500E Xilinx FPGA gives 25 GPIOs and controls an 8 channel analog-to-digital converter. There are two communication channels between the FPGA and the processor, the first one is a JTAG that allows the FPGA configuration and the execution of tests to the implemented circuits. The data, address and control buses are part of the second channel, which is used to exchange information between the processor and the peripherals implemented inside the FPGA.

SIE provides power connection through a USB port, which is also configured as a network interface. This allows the file transfer and the execution of a remote console using the *ssh* protocol. This communication channel can also be used to configure the non-volatile NAND memory, which simplifies the whole configuration of the platform through the USB port.

IV. GENETIC PROGRAM IMPLEMENTATION

The algorithm was implemented in two stages: the first one is about software development and the second one refers to a hardware implementation to speed it up on SIE platform.

¹All this is available at: <http://projects.qi-hardware.com/index.php/p/nn-usb-fpga/>

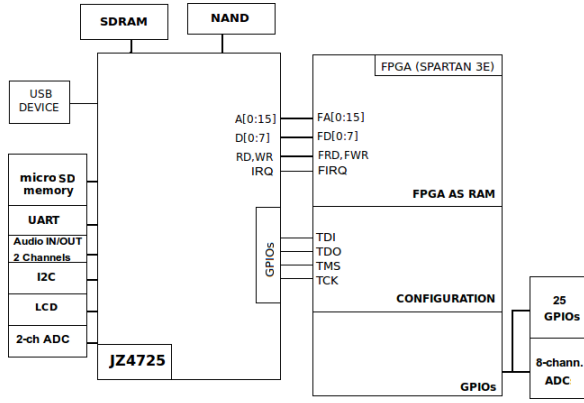


Fig. 2. SIE development platform structure

A. Software design

Natural evolution works with a whole population not with a single individual (except for selection and reproduction) some operations can be done separately, therefore almost all operations in a GP are implicitly parallel. Using the island approach, the population is divided into sub-populations that will evolve in each processor of the cluster or parallel architecture. When the system starts, each processor creates its sub-population and starts the evolution process, made up of fitness evaluation, selection, crossing, mutation and reproduction. Once the system reaches a number of generations, some individuals are selected to be transfer from one processor to another. A master processor is in charge of collecting the in-transfer individuals and moving them to the rest of the nodes (slaves), increasing the probability of convergence of the algorithm. The ratio of data exchange (the number of the best individuals to exchange increases the probability of finding a better solution) and migration frequency are important parameters to improve the performance of the algorithm. To implement communications in SIE, a custom message passing library was created to program the distributed system.

B. Hardware design.

In the second stage a profiling of the software implementation determined that the most consuming parts were the fitness function calculation and the new individual generation (25% and 35% of the execution time, respectively). Therefore, these two steps have been accelerated with a coprocessor in the FPGA. The Fitness Calculation Unit (FCU, see Figure 3) is the hardware element designed in order to accelerate this processes. This coprocessor is connected to the JZ4725 processor through its custom interface.

The chromosome passes from the DRAM memory to the FCU through the custom interface, and each of its cells are converted to a equivalent Look Up Table in ROM based translation. Then the number of wrong minterms is calculated using the information from the objective function and a counter as input. Finally the FCU calculates the final fitness value including the number of gates and the critical path, and will be sent back to the JZ4725 processor. In order to speed up

pseudo random number generation, a Mersenne-Twister-based coprocessor was inserted through the same custom interface.

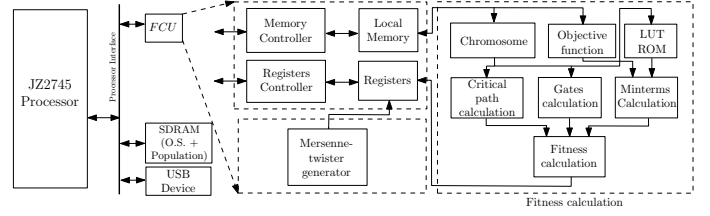


Fig. 3. FCU structure.

V. EVALUATION.

To obtain the performance and prove the scalability of the algorithm in SIE, it was compared to a High Performance cluster called ALTAMIRA. The cluster setup is made up of 18 eServer BladeCenters with 256 JS20 nodes (512 processors), using a Myrinet network with 1 Gbps bandwidth. On the other hand, the SIE configuration is made up of 6 JZ4725-FPGA nodes with the architecture described before. The response times are measured for the parallelized versions of the GP in both ALTAMIRA and SIE. Several scenarios have been checked with different input parameter configurations: 1) number of input variables (4, 8 or 12, corresponding to a comparator problem of 2, 4 and 6 bits); 2) population size (512, 1024 or 2048) and 3) number of nodes running the experiment, ranged from 1 to 6 in SIE, and 2 to 16 or 64 in ALTAMIRA. The first and second parameters determine the size of the problem. The last one gives an idea about the scalability of the system.

A. Response Time.

Figure 4 shows the response time for both platforms with different numbers of nodes and variables with 1024 individuals during 100 generations. These results show the high performance of SIE cluster for the algorithm. This experiment demonstrates that the response time for SIE does not depend on the size of the problem. Contrarily, response time in ALTAMIRA has a high dependency of the size of the problem, because individuals had to be evaluated in software.

Figure 5 shows the response time in both architectures when varying the number of individuals of the population. It is observed that both architectures has a strong dependency of the number of individuals in the algorithm. This is because increasing this number causes an increment of the software computational load for both clusters. Even in this scenario, SIE shows an excellent performance compared to ALTAMIRA.

B. Speedup.

The speedup of the SIE vs ALTAMIRA for different number of variables is presented in figure 6 with the number of variables fixed to 12. The excellent performance of SIE can be explained because individuals have been directly tested in hardware (FPGA), obtaining a combination of their true table on each cycle of the system clock. On the other hand,

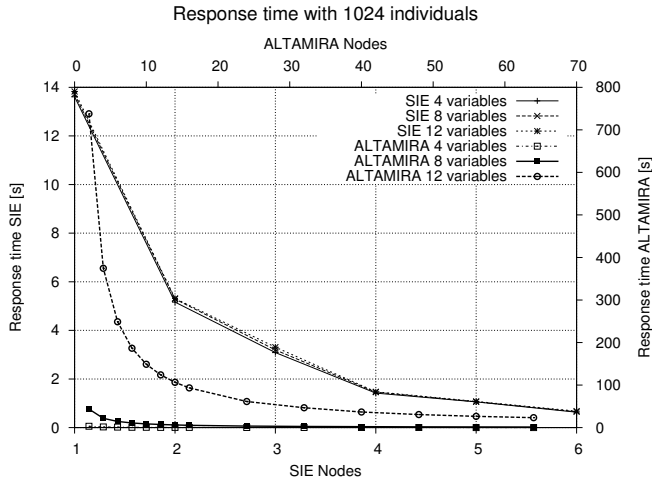


Fig. 4. Response time in SIE and ALTAMIRA for different number of variables.

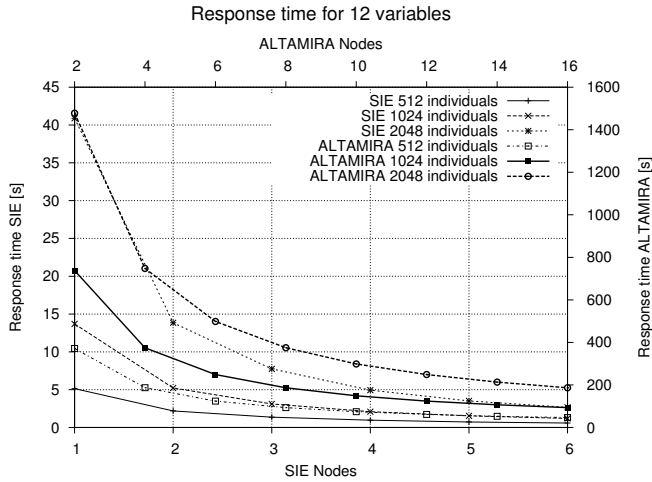


Fig. 5. Response time in SIE and ALTAMIRA for different number of individuals.

individuals evaluated in software by ALTAMIRA require a lot of system clocks, causing response times up to 180 times higher than SIE.

C. Resulting circuits.

Figure 7 shows an example of the resulting circuits for the 2-bit comparator problem. These resulting structures are novel compared to the results obtained with other techniques, such as Karnaugh maps and QuineMcCluskey.

VI. CONCLUSIONS AND FUTURE WORK.

This paper showed a novel way to evaluate individuals in an evolvable algorithm on an open embedded platform called SIE, and results were compared to an HPC called ALTAMIRA. To accelerate the evolution process, a coprocessor was implemented to calculate the fitness function and to generate random numbers, improving the performance for problems with more than 6 bits. Tests proved that the algorithm is more effective for 4-bit and 8-bit problems. 12-bit problems in SIE

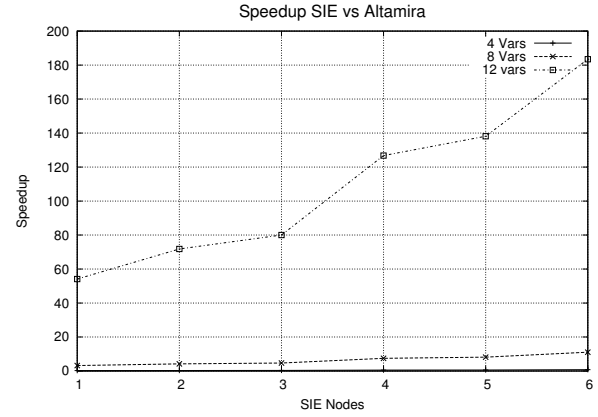


Fig. 6. Speedup of SIE vs Altamira comparing 1 SIE node = 2 Altamira nodes.

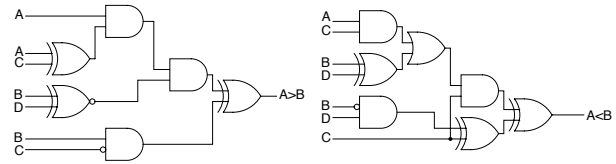


Fig. 7. Example of resulting circuits for the 2-bit comparator problem (A=MSB number1, C=MSB number2).

had excellent performance, but because the search space is too long, converging to a suitable solution was difficult for the algorithm. This problem could be solved as future work with some improvements in terms of multiple FCUs inside an FPGA, more nodes, and other hardware-accelerated genetic operators.

REFERENCES

- [1] D Goldberg and J Holland. Genetic algorithms and machine learning. *Machine Learning*, January 1998.
- [2] F. Rothlauf. *Representations for genetic and evolutionary algorithms*. Springer, January 2006.
- [3] J Koza, F Bennett, D Andre, and M Keane. Genetic programming iii: darwinian invention and problem solving. *Evolutionary Computation*, January 1999.
- [4] T Higuchi, T Niwa, T Tanaka, H Iba, and H de Garis. Evolving hardware with genetic learning: a first step towards building a darwin machine. *Proc. of the second Int. Conf. on from animals to animats*, January 1992.
- [5] J Miller and P Thomson. Aspects of digital evolution: Evolvability and architecture. *Lecture Notes in Computer Science*, January 1998.
- [6] Q. Yu, C. Chen, and C. Pan. Parallel genetic algorithms on programmable graphics hardware. *Lecture notes in computer scienc*, January 2006.
- [7] R Krohling, Y Zhou, and A Tyrrell. Evolving fpga-based robot controllers using an evolutionary algorithm. *Proc. I Int. Conf. on Artificial Immune Syst*, January 2002.
- [8] J Miller and P Thomson. Aspects of digital evolution: Evolvability and architecture. *Lecture Notes in Computer Science*, January 1998.
- [9] C. Camargo, W. Spraul, and A. Wang. Proyecto SAKC. URL: <http://en.qi-hardware.com/wiki/SAKC>.
- [10] Creative Commons. Licencias Creative Commons. URL: <http://creativecommons.org/licenses/>, 2004.
- [11] R. Stallman. Philosophy of the GNU project. URL: <http://www.gnu.org/philosophy/>, 2007.