TRABAJO DE FIN DE MÁSTER

2017

A Novel Approach to the Placement Problem for FPGAs based on Genetic Algorithms.

Máster Universitario en Inteligencia Artificial Avanzada

Department of Artificial Intelligence

Universidad Nacional de Educación a Distancia

by

Dr. Francisco Javier Veredas Ramírez

Advisor: Dr. Enrique J. Carmona Suárez

Abstract

This Master's thesis investigates the critical path optimization in the FPGA's placement. An initial investigation of the FPGA's placement problem shows that the minimization of the traditional cost function used in the simulated annealing's placement not always produce a minimal critical path. Therefore, it is proposed to use the routing algorithm as a cost function to improve the final critical path. The experimental results confirm that this new cost function has better quality results than the traditional cost function, at the expenses of longer execution time. A genetic algorithm using the routing algorithm as a cost function is found to reduce the execution time meanwhile is maintained a minimal critical path. The use of genetic algorithms with the new cost function will be useful in those cases where a minimum critical path is needed. Furthermore, this work investigates the use of genetic algorithm using the traditional cost function. In this case, no better critical path in comparison with a simulated annealing's placement is observed.

Contents

Chapter 1

Introduction

This chapter presents the motivation, and problem statement of this work. The next section is a brief recount of the motivation. In the second section, the scope of this work is described. In the third section the problem's statement is described. It follows a section with the research goals. Finally, in the last chapter, the rest of this work is outlined.

1.1 Motivation

Field Programmable Gate Arrays (FPGAs) are configurable devices that can be used to implement any digital hardware design [1]. FPGAs also offer features such as built-in hardwired processors, substantial amounts of SRAM memory blocks, clock management systems, and support for many of the latest, very fast device-to-device board-level signaling technologies. FPGAs are used in a wide variety of applications, ranging from data processing, storage, instrumentation, network communications, and digital signal processing.

To map a hardware design into an FPGA, it is needed a series of automated software tools (CAD tools). These tools should take into account the minimum clock frequency of the hardware design. Note that the minimum clock frequency is related with the optimal critical path of the hardware design. In the state of the art's CAD tools, not always the minimum clock frequency is achieved. Therefore, better CAD's algorithms are needed. This Master's thesis deals with these algorithms.

1.2 Scope

This section briefly describes the design flow of FPGAs and points to the focus of this work.

A typical design flow for FPGAs consists of a concatenated set of CAD tools. The standard design flow for FPGAs consist of three major steps [2]: design entry and synthesis, design implementation, and design verification.

The start of the design flow is a digital circuit in a form of a schematic entry or a high-level description of the hardware, expressed in high-level hardware description languages (HDL) such as VHDL [3] or Verilog [4]. The description is read by a synthesis program, which maps the HDL into a network of Boolean equations, flip-flops (FFs), and pre-defined modules. During the synthesis process, the Boolean equations are optimized with respect to estimated implementation area, and delay. The optimizations performed at this stage are limited to those that can benefit circuit implementations on any medium, not just FPGAs. The Boolean equations are then first mapped into a circuit's netlist of Look-Up-Tables (LUTs) and FFs. During this technology mapping process, the circuit is again optimized with respect to the estimated implementation area, and delay. The optimizations are targeted towards specific implementation technologies. Area is typically optimized by minimizing the number of LUTs or logic blocks that are required to implement the circuit. Delay is often optimized by minimizing the number of LUTs or logic blocks that are on the estimated critical paths of the circuit.

The design implementation steps consist of packing, placement, routing, and bit-stream generation. The packing process groups logic blocks into Basic Logic Elements (BLEs) [1] . The specific location of each netlist Logic Block (LB)[2] on the target FPGA is determined during the placement process. A placement program assigns each LB to a unique location to optimize delay, and minimize wiring demand. Figure 1.1 shows an FPGA architecture with a placed circuit. As the placement does not deal with the routing, the routing resources are not shown in the figure. During the routing process, a routing program is used to connect the LBs by determining the configuration of the programmable routing resources. The main task of all routing programs is to successfully establish all connections in a circuit using the limited amount of physical resources available on a target FPGA. The other task of the routing programs is to minimize delay by allocating fast physical connections for critical paths. The synthesis, and the technology mapping are together commonly called the front end of the FPGA CAD flow. The packing process, placement, and routing steps are commonly called the back end of the FPGA CAD flow. Finally, from the design, and placement and routing information, a bitstream is created for subsequent programming of the FPGA device.

[1]See BLE's definition in Section 2.1.

[2]LB refers to the entities of the mapped netlist. Configurable Logic Block (CLB) is the physical block in a FPGA.

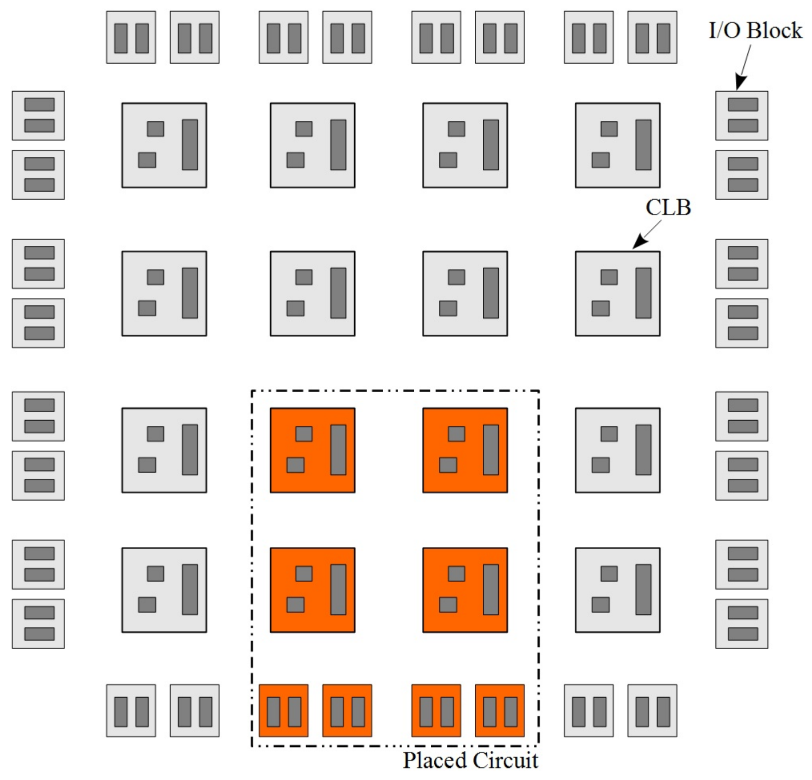


Figure 1.1: FPGA Architecture showing a four LBs and four I/O Blocks placed circuit.

Design verification is the process of testing the function, and performance of a design. The verification can be realized with functional simulation, static timing analysis, and in-circuit verification. Functional simulation determines whether the logic of the designed circuit is correct. Functional simulation can take place in early stages of the design (e.g. in the HDL phase). The static timing analysis verifies that the design meets the timings specifications. In-circuit verification tests the circuit under typical operating conditions.

The focus of this Master's thesis is the optimization of the critical path. In concrete, the optimization of the critical path in the placement's algorithm. The critical path of a digital circuit is here understood as the maximum delay of all the logic paths between two sequential elements (e.g. FFs) of a circuit. The final critical path of a circuit is known at the end of the design flow. This means, that in the case of the placement's algorithm, a model (in form of a cost function) is requested to guess and optimize the critical path. The next section will describe the placement's problem on FPGAs.

1.3 Problem Statement

From the designer point of view, the problems of placement are:

1. Time duration.
2. The placement solution can be no feasible to find a routing.
3. Bad quality of placement metrics: Area, critical path delay, and power consumption.

Time duration. Mapping complex circuits in current FPGAs with more than a million configurable logic gates [5], requires large time (order of hours) to perform placement. This limits significantly the designer productivity.

Solution Feasibility. As explained in the previous chapter, the next step after placement is routing. The routing resources (as number of tracks per channel) in an FPGA is limited. It can be placement solutions that makes routing impossible.

Placement Metrics. As it can be seen from the placement metrics, the placement algorithm tries to optimize the circuit performance (critical path), area (minimum array size), and power consumption of the FPGA.

As one of the most important optimization parameters is the critical path, this work will focus on it. It has been said in the previous section that it is not possible to calculate the final critical path in the placement's step (it is needed the later routing step). To guess the circuit's delays after the placement, a cost function is used. In the next subsection, the common used cost function (called in this work traditional cost function) is presented. The last subsection details the problem of the traditional cost function.

1.3.1 Traditional Cost function

The cost function of the placement algorithm depends of the FPGA architecture and the desired optimization. As one of the objectives of the FPGA placement is to allow the FPGA routing, the wiring congestion (how many tracks are used in one channel) is an important metric. Therefore, the final cost function should have a wire cost term [6]. In most of the cases, it is desired to minimize the circuit delays associated with the placement. These delays can be modeled with a timing cost term [7].

It will follow a description the cost terms most common used for the cost function. To avoid confusion, in this work it is used the term VTR-N to designate specifically the VTR tool with the traditional cost function.

Wiring Linear Congestion Cost

The wiring linear congestion cost term is defined as,

$$wire\_cost = \sum_{i=1}^{N_{nets}} q(i) \cdot \big[bb_x(i) + bb_y(i)\big] \tag{1.1}$$

This cost is applied as a summation to all the nets of the circuit. The smallest geometrical span of one net is represented with a rectangular boundary box. The horizontal value of the boundary box in one net is $bb_x(i)$, and the vertical is $bb_y(i)$. An example of boundary box is depicted in Figure 1.2. Note that the lines represent the pin connections of a net. These lines are only conceptual because the routing is still not done. The $q(i)$ parameter compensates the wire estimation cost and is obtained from [8]. The $q(i)$ parameter values are calculated as follows: (1) M randomly placed pins are set within a boundary box, (2) an optimal Steiner with this placed pins are drawn, (3) the horizontal (or vertical) cut points of this tree is counted, (4) the last steps are repeated for K random tries for each M pins configuration. As example, in the left of Figure 1.3, it is possible to see a net with four randomly pins placed in a boundary box. A Steiner tree is constructed and a vertical cut line is moved from left to right to count the crossing points. In the left of the figure, the vertical line cross one point of the Steiner tree. In [8] is used a configuration of K=10000 random tries for each pins configuration. The pins configuration goes from $M = 1 \sim 3$ to $M = 50$ pins. For each pin configuration and direction (horizontal or vertical) a wiring distribution map (WDM) can be constructed. Figure 1.4 shows the representation of a horizontal WDM with 20 pins. From the figure that the resources demand is lower in near the border of the boundary box. The mean values of the WDM of different pin configurations are used to calculate $q(i)$. Therefore, $q(i)$ represents the expected number of wires crossing a cut line through the bounding box in units of tracks. A table was constructed in [8] with values of 1 for $1 \sim 3$ pins to 2.79 for 50 pins. To extend these values over 50 pins, [7] did a linear regression,

$$q(i) = 2.79 + 0.02616 \cdot (Num\_Net\_Pins - 50) \tag{1.2}$$

where Num_Net_Pins is the number of pins of one net.

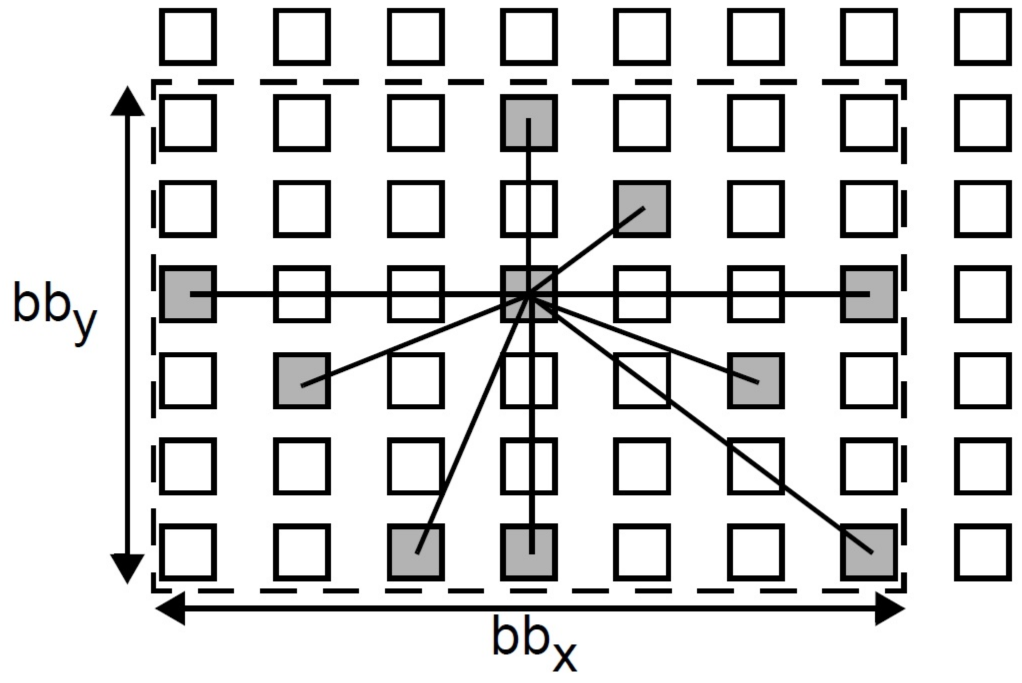


Figure 1.2: Boundary box for a net [7].

Timing Cost

The timing cost term is defined as,

$$timing\_cost = \sum_{\forall i,j \in circuit} Delay(i,j) \cdot Criticality(i,j)^{CE} \tag{1.3}$$

This cost is applied as a summation to all source node i to the other possible (connected) sink node j of the circuit. The delay of a source node i to a sink pin j is expressed with $Delay(i,j)$. The term $Criticality(i,j)$ is defined as,

$$Criticality(i,j) = 1 - \frac{Slack(i,j)}{D_{max}} \tag{1.4}$$

The slack, $Slack(i,j)$ is the delay that can be added to a path without being critical. D_{max} is the critical path delay. The critical exponent, CE, is a constant to weight the paths that are more or less time critical. To find CE, Marquardt et al. [7] ran a set of experiments with a set of configurations a circuits. It is found a best case around $CE = 8$ when the cost is balanced (congestion and timing cost). Greater values of $CE = 8$ does not decrease the critical path, and the algorithm focus to minimize the wiring cost. Another experiment, when the only contribution of the total cost is the timing cost, shows the best

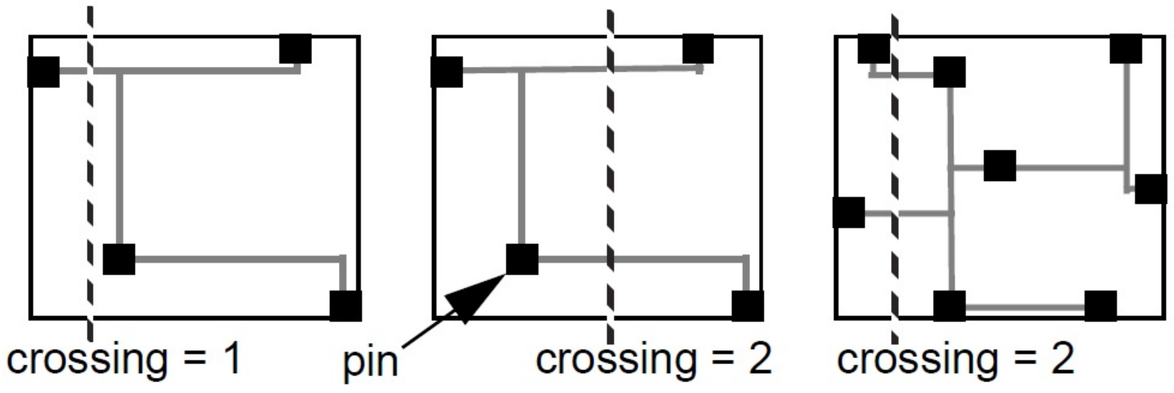


Figure 1.3: Example of crossing points [8].

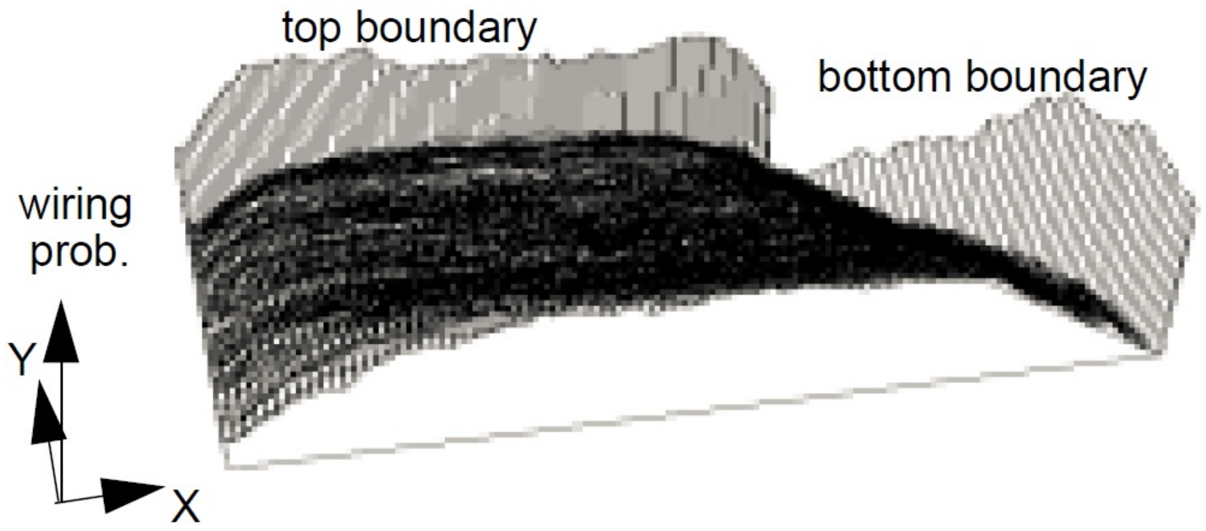


Figure 1.4: Horizontal wiring distribution map of 20 pins [8].

value is $CE = 2$ or $CE = 3$. As a result of these experiments, the default value in the academic VTR-N [9] tool is $CE = 8$.

The traditional cost function to optimize the critical path has the form:

$$Cost_{wire,time} = \lambda \cdot \frac{timing\_cost}{previous\_timing\_cost} + (1-\lambda) \cdot \frac{wire\_cost}{previous\_wire\_cost} \tag{1.5}$$

where λ is a trade-off parameter. To find λ, Marquardt et al. [7] run a set of circuits with $CE = 8$. It is found that in average $\lambda = 0.5$ has the best compromise between wire congestion and critical path cost. A close value $\lambda = 0.6$ was found in another independent experiments [10].

1.3.2 Problems with the Traditional Cost Function.

The extensive use of the traditional cost function by researchers in the last twenty years validates the usability of itself. An open question is if this cost function is accurate enough. An experiment using the VTR-N placement and routing tool [11] was performed. The circuits *planet*, $s1238$ and $mm30a$ from the MCNC benchmark [12] have been tested with different number of iterations and using $\lambda = 0.5$ and $\lambda = 1$. To compare between the different runs, it is used the same normalization. The values of this normalization are chosen with the resulting *wire_cost* and *timing_cost* of the first run. In Figure 1.5 the results of the cost function and the critical path after routing are compared. It can be seen in the figure that, given two cost values, if the former is smaller than the second, the same inequality is not always guaranteed in the respective critical path values. As example, in the circuit $s1238$, a $cost = 0.26$ has in our experiment a critical path of 4.44 ns, and in another test a $cost = 0.33$ has a critical path of 4.36 ns. This indicates that a placement algorithm that optimizes the traditional cost function, no necessarily will optimize the critical path.

1.4 Hypothesis and Research Goals

This work focus in optimizing the critical path of FPGAs using the routing algorithm in the placement step. With this end in mind, the following hypothesis are done:

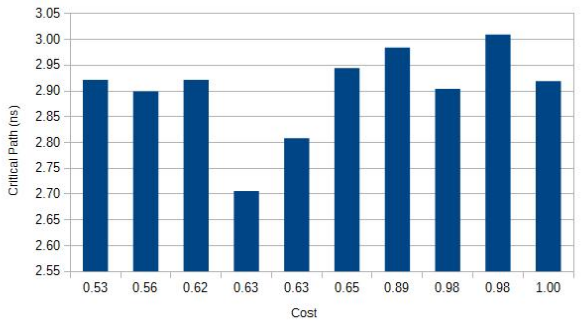
1. A cost function based on the routing algorithm is more accurate than the cost function that is traditionally used to optimize the critical path

2. In order to optimize the critical path, the use of evolutionary algorithms can be competitive compared to classical methods.

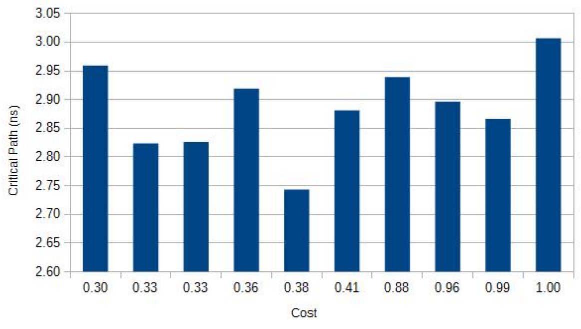
Taking into consideration these hypothesis, the research goals of this Master's thesis are:

1. Analysis and study of a new cost function based on the routing algorithm to optimize the critical path.

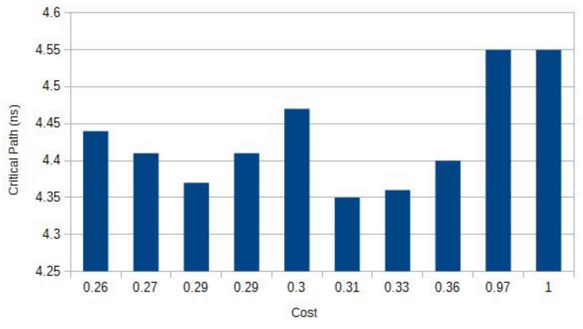
2. Analysis and study of genetic algorithms in the FPGA's placement to optimize the critical path.



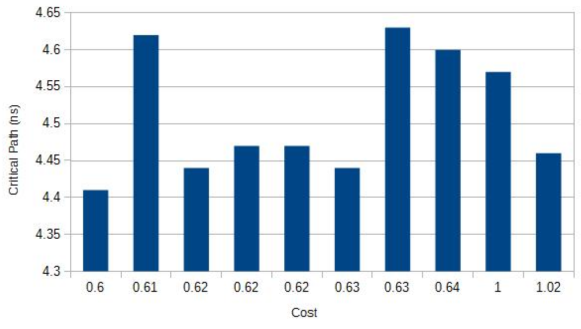
(a) λ=0.5 planet



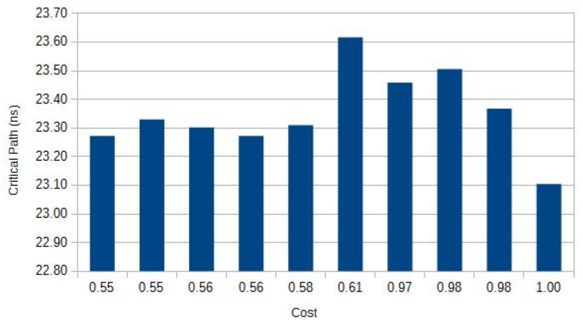
(b) λ=1 planet



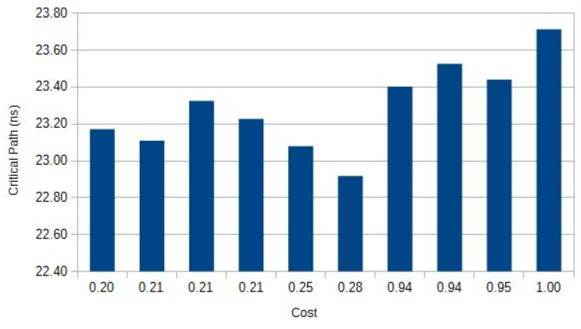
(c) λ=0.5 s1238



(d) λ=1 s1238



(e) λ=0.5 mm30a



(f) λ=1 mm30a

Figure 1.5: Comparison of the traditional cost function and the critical path after routing using the VTR-N's tool.

1.5 Master's Thesis Outline

The rest of this Master's thesis is organized as follows. In the next Chapter 2, the state of the art of the FPGA's placement is presented. The alternatives of the FPGA placement are presented in Chapter 3. Chapter 4 presents and discusses the experimental results. Finally, the last Chapter provides concluding remarks.

Chapter 2

State of the Art

The objective of this chapter is the review of the placement's approaches found in literature. To understand the terminology used in this and further chapters, the next section presents the main concepts of the FPGA's architecture, and the second section briefly describes the algorithms used in FPGA's placement. A comprehensive literature's review is discussed in the last section.

2.1 FPGA Architecture

An FPGA has three key parts:

1. Input and output (I/O) Blocks.

2. Hard macros: SRAM block, multiplier, and digital clock manager (DCM).

3. Logic blocks with the configurable interconnect (tile).

Typically, the I/O blocks are arranged as a ring around the outer side of the FPGA's die. There are two different types of I/O blocks: global I/O blocks, and programmable I/O blocks. The global I/O blocks include dedicated units for configuration, JTAG test[1], clock, and power/ground connection. A programmable I/O block provides individually programmable input, output, or bi-directional (any combination of the input, and output configuration) access to one of the I/O pins on the exterior of the FPGA package.

Hard macros are used to improve the area, and delay efficiency of the FPGA. Common blocks include SRAMs, multipliers, transceivers, processors, and DCMs. A generic FPGA

[1]In some FPGA devices, the JTAG port is also used for configuration.

consists of numerous CLBs. Each of them having the capability to implement a wide range of digital logic functions. A Boolean logic function can be obtained thanks to a LUT-based or multiplexer-based hardware implementation. A CLB has a sub-level of hierarchy called Basic Logic Element (BLE). A BLE has a minimum of one LUT and one FF. Note that a CLB contains one or more BLEs. The configurable interconnect is used to connect the input, and output of the CLBs. This interconnect is designed to support any netlist mapping without major routing congestion. Typically, the array is two-dimensional, although there are some solutions which are based on a one-dimensional architecture (i.e. a row-based architecture [13]). The later however do not take advantage of having short connections to next neighbors in the vertical, and horizontal direction.

In a so-called island style FPGA, the configurable logic blocks are arranged in a two-dimensional array with horizontal routing channels between rows of blocks, and vertical routing channels between columns. Each routing channel comprises a bundle of routing tracks for signal transport (*tracks per channel*). At each crossing point of routing channels, there is a configurable switch-matrix, which allows change of direction or communication with its neighboring logic block. There are several types of switch-matrix depending of the possible configurations (e.g. Wilton [14], subset [15] , hyperuniversal [16]). A so-called tile comprises a CLB, a switch matrix, and routing channels. By pure abutment of tiles, the array size can be varied (i.e. no additional routing resources need to be provided after the tiles are placed.) Figure 2.1 shows an array of four tiles with its routing resources.

The configuration of the FPGA comprises the configuration of the CLBs, IO blocks, and routing resources, and it is done by means of programmable switches. There are different ways of implementing a programmable switch. Programmable switches that are currently in use in commercial FPGAs are SRAM cells, and Flash memory cells. The FPGA market is coped with SRAM cells because this is a cost-effective technology process.

There are two major players in the FPGA market: Xilinx Inc. [17], and Intel Corporation (former Altera Inc.) [18]. Both use architectures which are array-based, and have logic clusters at the lower hierarchy level. They have two families, one for high performance, and high complexity, and another one for low cost, and higher volume. Both companies follow the trend of embedding more, and more optimized macros into the array (e.g. digital signal processors (DSPs), transceivers, ...).

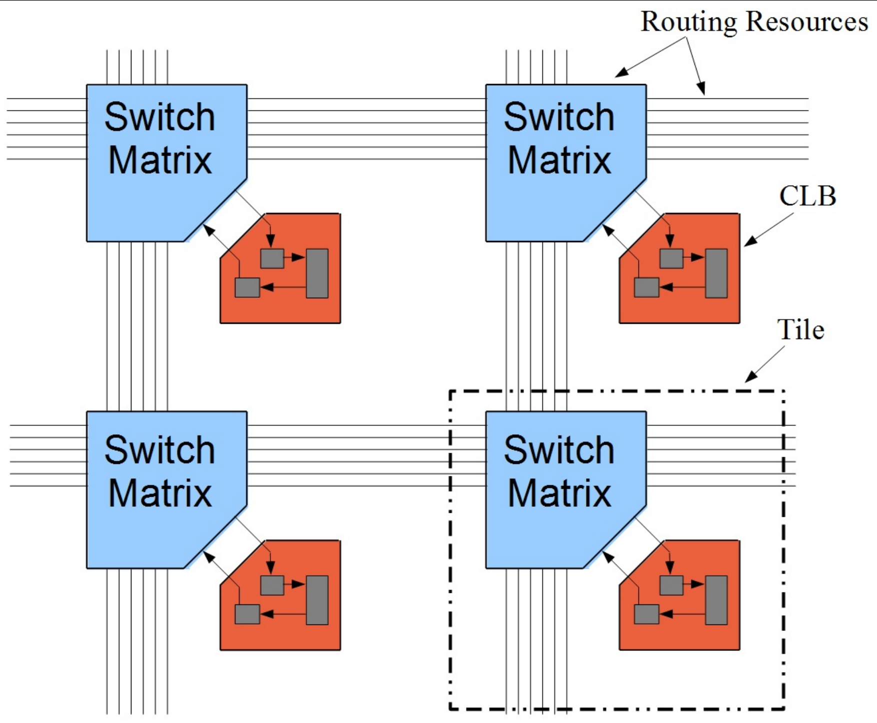


Figure 2.1: FPGA array of four tiles.

2.2 Placement Algorithms

With the invention of FPGAs by Xilinx Inc in 1985 [1], there is the necessity of using automated tools for placement. In all the placement algorithms found in literature, a cost function is involved. The cost function helps in the optimization of a quality parameter (e.g. critical path). Several algorithms can be found to solve the placement problem of FPGAs:

1. Simulated annealing (SA)
2. Genetic algorithm (GA)
3. Analytic method (AM)
4. Other (OT)

Simulated annealing algorithm. The first algorithms used for FPGA were adaptations of standard-cell VLSI's placement algorithms. Specifically, it is used the SA [19]. SA emulates a physical process. In this process, a material is first heat allowing the molecules moving freely, and then the material is cooling down until all molecules takes a fixed position. If the process is slow cooled, the total energy of the material is minimal. Algorithm 1 shows a traditional SA algorithm in FPGAs [20]. Using analogy for an FPGA placement, first the algorithm starts with a random placement of the logic blocks (molecules in the physical model). After the random placement, the algorithm iterates in a loop moving the logic blocks. In each movement of a logic block a cost function (energy in the physical model) is evaluated. It is desired a better cost value than the previous iteration. The acceptance probability of the new placement, depends of a parameter (temperature in the physical model) that varies during all iterations.

Genetic algorithm. GAs emulates the natural evolution of species as they evolve to better adapt to their environment. The GA starts with a set of initial placements (population). An initial placement can be random or not. One placement solution of a population is commonly represented as string of placed logic blocks (chromosome). After the initial placement, the GA iterates to find a feasible solution. In each iteration (generation), a cost function (fitness) is calculated to improve the quality of the placement. At the end of each iteration, it is performed a selection and a combination (e.g. crossover, mutation) of the best placement solutions.

Analytic method. There is a multitude of analytical methods (e.g. cluster growth, quadratic assignment). A popular method in this category is the min-cut algorithm. In the min-cut placement, the designed circuit is split in two sub-circuits that minimize the number of nets connected in both circuits. The two sub-circuits are placed in separates halves of the FPGA. This process of two is recursively applied until a criterion is satisfied. Because the optimization the sub-circuit partitioning becomes difficult and excessive constrained in complex circuits, heuristic algorithms are preferred than analytical methods.

Other. Other types of algorithms are swarm optimization and ant colony optimizations. These two types of algorithms are stochastic.

2.3 Taxonomy

A literature study with the three type of algorithms is shown in Table 2.1. The table shows the characteristic and improvement of each paper. Each paper is briefly discussed in the next subsections.

Simulated annealing algorithm. SA is the most used placement algorithm in FPGAs

```
Input  : circuit netlist, GA parameters, FPGA parameters
Output: placement netlist
                                                        ▷ Init
load circuit and FPGA structure;
random placement;
init temperature;
                                                        ▷ Main Loop
while not termination (exit_criterion) do
    normalization_cost = previous_cost;
    for all mov_lim do
        swap randomly one block;
        calculate new_cost;
        $\Delta cost = \frac{new\_cost - previous\_cost}{normalization\_cost}$;
        if $\Delta cost \leq 0$ then
            accept new placement;
            $previous\_cost = new\_cost$;
        else
            $r = random(0,1)$;
            if $r \leq e^{-\frac{\Delta cost}{temperature}}$ then
                accept new placement;
                $previous\_cost = new\_cost$;
            end
        end
    end
    update temperature;
end
                                                        ▷ End
save best placement;
```

Algorithm 1: Simulationg Annealing Algorithm. $init\_temperature = 20 \cdot std\_dev$ where the standard deviation, std_dev, is calculated with the cost variation of moving blocks randomly (mov_lim times). $mov\_lim = number\_of\_blocks^{1.3333} + 1$. The $exit\_criterion$ is satisfied when $temperature < 0.005 \cdot \frac{cost}{number\_of\_nets}$. The update temperature is done by means of a tabulated parameter α with $T_{new} = \alpha \cdot T_{old}$. The cost of the initial random placement is taken the first time that $normalization\_cost$ is calculated.

and it has been extensively studied by the University of Toronto (Canada) [20]. In 1997, this university made public its own tool (called VTR) [6] together with the source code [9]. Most of the investigations with FPGA's placement found in literature use as reference this tool. In the year 2000, the same university improved its tool adding timing analysis into the function cost [7]. Since then, the University of Toronto has not modified the placement's algorithm of the VPR's tool. Nag et al. in [21] propose an algorithm that performs simulta-

Type	Ref.	Characteristic	Improvement	Year
SA	[6]	Only wire cost	N.A.	1997
SA	[21]	Simultaneous P&R	Better CP (small circuits)	1998
SA	[7]	Wire and timing cost	Better CP	2000
SA	[22]	Modified Swap function	Better exec.time than SA	2007
SA	[23]	Congestion term in Cost	Small CP improvement	2015
GA	[24]	Crossover and mutation	N.A.	1999
GA	[25]	No crossover	Better than SA (small circuits)	2000
GA	[26]	Distance in cost function	Better exec.time than GA	2004
GA	[27]	Neural Network	Better exec.time than GA	2007
GA	[28]	Local search in mutation	Improvement small circuits	2007
GA	[29]	No crossover	Worst than SA	2010
GA	[30]	Crossover and mutation	N.A.	2011
GA	[31]	Crossover operator	Confined-swap operator is better	2012
GA	[32]	Clustering mutation	Better CP than GA	2013
GA	[33]	Parallel execution	Better exec.time than SA	2013
AM	[34]	Quadratic placement	Worst than SA	2005
AM	[35]	Heterogeneous FPGAs	N.A.	2012
OT	[36]	Swarm optimization	Worst than SA	2004
OT	[37]	Ant colony optimization	Worst than SA	2007
OT	[38]	Swarm optimization	Worst than SA	2015

Table 2.1: Literature taxonomy for FPGA's placement algorithms. P&R = Placement and routing, critical path= CP, execution time = exec.time, No Applicable = N.A.

neously the placement and routing. Preliminary tests with trivial circuits (few LBs) showed improvement in the critical path at the expenses of the execution time. In [22] the execution time was reduced using a new way of moving LBs (swap function) during the algorithm. A small gain is shown in [23] improving the routing congestion cost factor.

Genetic algorithm. Most of the GA's placement investigations found in literature or it is not complete (e.g. comparisons with SA), or have experiments with small trivial circuits. The first paper using GA in FPGAs is [24]. This paper gives an overview of the GA used, and shows experimental results without any comparison. The paper [25] presents a GA without recombination, and claims that it is better than SA for small circuits. The GA proposed in [26] has a cost function that only used the distance between blocks. This article focus in the time execution between a GA with and without parallelism. The authors only use small circuits, and results are not compared with SA. In [27], the authors use a neuronal network to control the ratio between the recombination and mutation. The neuronal network speeds up the execution time in comparison a base GA. A placement's local search is presented in [28]. The experiments only use the wire cost's factor, and small improvements are obtained with small circuits. The paper [29] compares the use of SA and GA. It shows not advantage against the SA of using a GA. The same author presents in [30] a crossbar operator, but no comparisons are shown respect SA. The crossover operator inside GA is investigated in [31]. In this paper is found that a so called confined-swap operator was better compared to a partially mapped. In a partially mapped crossover, the combinations with conflicts are avoided blocking the mapping of the parent's gen with conflicts to children. Thus, it can happens that a children has a one-to-one mapping of the parent. In a confined-swap operator, when a conflict is found, it is searched a location without conflict near the conflict's location. The gens are grouped in [32]. Without any comparison with SA, it shows a critical path improvement respect a base GA. Finally, [33] modified the GA to parallelization and improvement of the execution time.

Analytic method. SA outperforms the quadratic method proposed in [34]. A analytical methods to place heterogeneous FPGAs (i.e. FPGAs that contains other blocks than CLBs, such as memory or multiplier blocks) was implemented in [35]. No comparisons were done with SA.

Other. In [36] and [38] used swarm optimization. No improvements respect SA are observed. Ant colony optimization is used in [37]. Here also no improvements respect SA are observed. These algorithms has been exercises to use an algorithm in the placement problem, more than trying to optimize a quality parameter.

Chapter 3

New Solutions to the Placement Problem

This chapter explains the two new placement solutions proposed in this work: (1) the use of genetic algorithms, and (2) the use of a routing algorithm as a cost function. The first section presents the genetic algorithm used within the placement problem. The second section describes the use of a routing algorithm as a cost function. This cost function will be used with the SA and GA algorithms.

3.1 Placement with Genetic Algorithms

In this section the proposed GA for placement is presented. The next subsection explains the chromosome codification, the second subsection explains the adaptation of the traditional SA's cost function into the GA. The last subsection describes the proposed GA.

3.1.1 Codification

The representation of the chromosomes in the placement's GA affects the efficiency of the algorithm and it is needed to be explained. In the implementation of this work, each element of the chromosome is represented by two unsigned integers (one integer for the horizontal axis, and the other for the vertical axis). A gen corresponds to a LB of the circuit. Thus, for a circuit with C LBs, the chromosome is

$$(x'_0, y'_0), (x'_1, y'_1), ..., (x'_C, y'_C) \tag{3.1}$$

The values of the gens to the FPGA location are translated by means of a module operator. In a FPGA with the array size N_x x N_y, a i gen of the chromosome has the FPGA coordinates,

$$\begin{aligned} x_i &= x_i' \ MOD \ N_x \\ y_i &= y_i' \ MOD \ N_y \end{aligned} \tag{3.2}$$

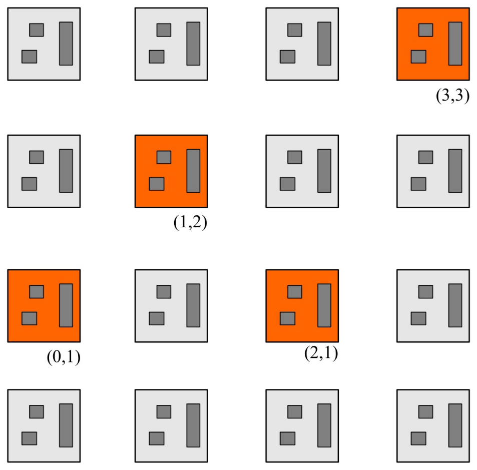
As example, a chromosome of a circuit with four logic blocks can have this codification: $(49, 114)$, $(0, 1)$, $(12626, 17)$, $(35, 414339)$, and an FPGA with an array of 4x4 CLBs, the first gen has a module:

$$\begin{aligned} x &= 49 \ MOD \ 4 \ = 1 \\ y &= \ 114 \ MOD \ 4 \ = 2 \end{aligned} \tag{3.3}$$

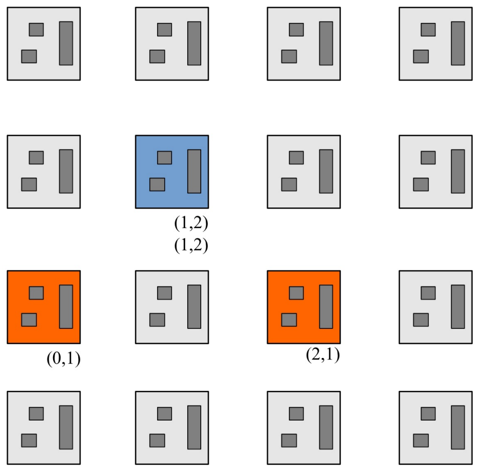
Then, the location of this block will be $(1, 2)$. The final location coordinates will be $(1, 2)$, $(0, 1)$, $(2, 1)$, $(3, 3)$. Figure 3.1 (a) shows the final placed circuit of this example.

It can be that the module operator gives a location that it is used by another LB. As example, a circuit with $(49, 114)$, $(0, 1)$, $(12626, 17)$, $(37, 1234)$, the placement coordinates will be $(1, 2)$, $(0, 1)$, $(2, 1)$, $(1, 2)$. The same coordinate $(1, 2)$ is used by two different LBs (see Figure 3.1 (b)), and this is not possible. To solve these collisions, first the first LB is placed in the found coordinate (Figure 3.1 (c)) and second for the subsequent LBs an algorithm with a spiral search method [39] is used. The spiral search algorithm starts in the lower closer coordinate to look if the surrounding locations of the initial block location are free, if no free location is found, the algorithm searches in the following surrounding locations, it does this until a free place is found (Figure 3.1 (d)). The algorithm search is clockwise. In the example, the algorithm started to look in $(0, 1)$, but this location was already used by another LB. So, the algorithm looked into $(0, 2)$, and it found that it was placed, so the LB is placed there. As the number of LBs is always lower than the number of CLBs, a free location always exist.

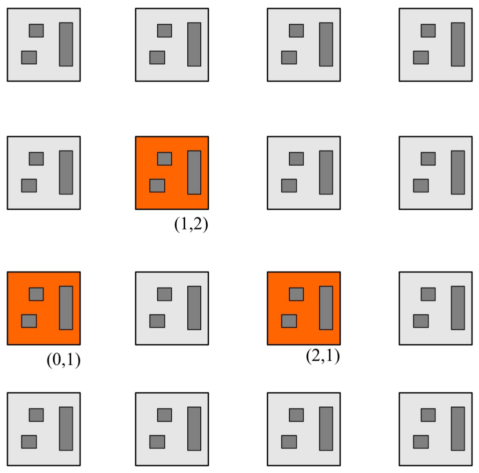
In the existing literature is found a similar representation for the chromosomes [29], [25], [24], [28]. In this literature, as in this work, the codification represents the identification of the LBs of a circuit and its location. The difference with this work, is that in the literature's work the codification is done sequential, i.e. the first CLB block is placed, and after the second CLB is placed, etc. With this technique, the conflicts are avoided.



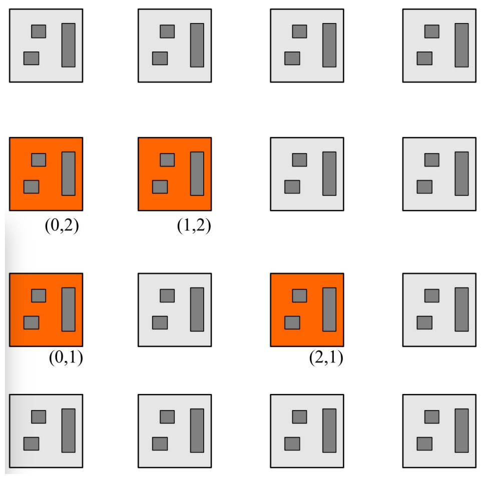
(a) No conflict



(b) Conflict



(c) Resolution conflict



(d) Resolution conflict

Figure 3.1: Placement with and without conflicts.

3.1.2 Fitness Function.

The terms wire and timing cost of the fitness function, is calculated in a similar way as the VTR-N (see Section 1.3.1). But, as the normalization has not the cost updates like the VTR-N, the GA's cost normalization is done with two steps: (1) the initialization is done searching the best individual with a fix normalization from the best wire and time costs of all the population, (2) for the next iterations, the wire and time costs of the previous best individual (*prev_popul_best_wire_cost* and *prev_popul_best_timing_cost* respectively) is used as a normalization factor. So, the total cost is,

$$Cost_{wire,time} = \lambda \cdot \frac{timing\_cost}{prev\_popul\_best\_timing\_cost} + (1-\lambda) \cdot \frac{wire\_cost}{prev\_popul\_best\_wire\_cost} \quad (3.4)$$

where λ is a trade-off parameter, and *timing_cost* and *wire_cost* are defined in Section 1.3.1. Any improvement of the cost in one iteration will be bellow one. Note that a normalization is needed for the calculation of the total cost because the order of the timing cost is quite different as the wiring cost. The GA with this fitness function is called in this work GA-N.

3.1.3 Genetic Algorithm

The proposed algorithm is shown in Algorithm 2. The algorithm starts loading the circuit's netlist and placement's parameters, and after that it initializes the population and performs an initial random placement. The placement of blocks is not restricted and can be done in all the locations of the FPGA array. For the random placement is used the *s_rand* pseudo-random function from the *Microsoft*'s *stdlib* library [40]. After the initial random placement, the cost of each individual is calculated and the best individual is saved.

Inside of the main loop the algorithm performs the parent selection, the recombination, the mutation, elitism, and the selection of survivals.

The recombination is performed for all the population. The parent selection is done by means of tournament with size two, i.e. the first parent is found selecting the best of two random individuals of all the population, and the same it is done for the second parent.

A random numbed between zero and one is generated, and if this number is below of a probability threshold (P_c), parent crossover is applied. The crossover in the recombination is applied has one random selected point. Two types of crossover can be found: (1) Without conflicts, and (2) With conflicts. Crossover without conflicts happens when the LBs of the two parents doesn't share any placement location. An example of this can be seen in Figure 3.2. It is possible to see that the random selected point is in this case, the midpoint of the chromosome. Two children are obtained from the two parents. Crossover with conflicts happens when one LB or more of one parent has the same placement location as the other parent. An example with conflicts is shown in Figure 3.3. One of the children of combining two parents (Figure 3.3 (a)) has two LBs trying to be placed in the same location $(1, 2)$ (Figure 3.3 (a)). The same method as in Section 3.1.1 is used to solve this conflict, i.e. the first LB is placed in $(1, 2)$ (Figure 3.3 (c) left), and for the next LB a spiral search is performed to find the free location place $(1, 1)$ (Figure 3.3 (c) right).

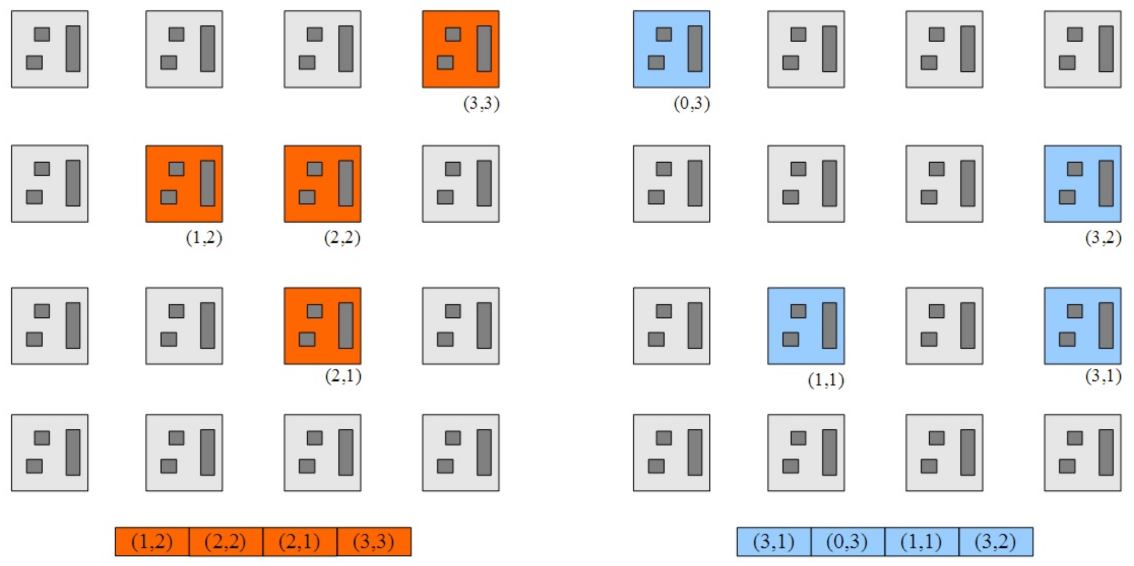
After the new children are found, the mutation step is performed to all the population except the best individual. One mutation is one movement of a circuit logic block from one location of the FPGA array to another allowed (not used) location. The new location is found randomly. The new location can be free and then the block is just moved (see an example in Figure 3.4 (a)), or it can be that another block is already placed. In the case that the new location has another block placed, a swap of the two blocks is performed (see an example in Figure 3.4 (b)). The reason why a swap is performed and not a search of a free place is because the size of the FPGA array is limited and the restriction of movements can be high. As example, with an FPGA array of 5x5, and a circuit with 25 blocks, there is not a free space to move a block, but still the algorithm can do swaps of blocks. Note that with this mutation method, a LB is always placed in a valid location and therefore, it is not needed to solve any conflict. A set of movements (dependent of the probability of mutation (P_m and the numbers of gens)) are done in the mutation. The movements are always accepted. In the simulated annealing algorithm, only the movements with better cost are accepted.

Once that all the new population is processed and the total cost of each individual is updated, the population is ranked by minimum cost and the children's population is updated with the new one (generational model). This genetic algorithm has elitism, i.e. the best individual to the next generation is preserved. The termination of the main *while* loop is done after a number of generations.

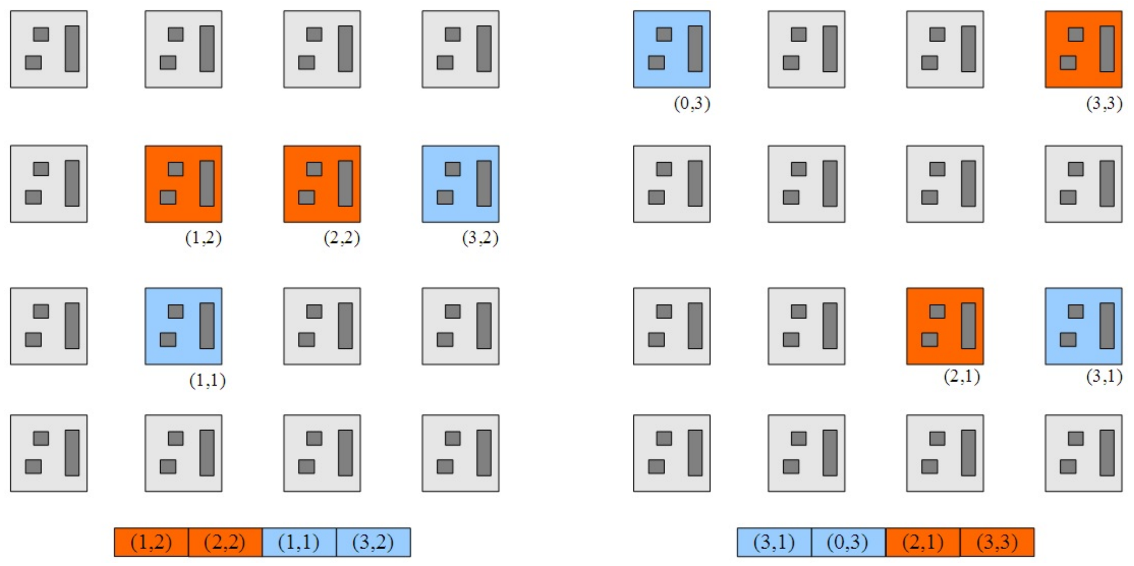
Finally, after the main loop is finished the best placement is stored in a placement netlist file.

```
Input  : circuit netlist, GA parameters, FPGA parameters
Output: placement netlist
                                                        ▷ Init
load circuit and FPGA structure;
random placement;
calculate cost;
save best individual;
                                                        ▷ Main Loop
while not termination (number generations) do
    for all population do
                                                        ▷ Parent Selection
        select two random candidates from all population;
        set parent1 from best of two candidates;
        select two random candidates from all population;
        set parent2 from best of two candidates;
                                                        ▷ Crossover
        if random_probability[0,1) <$P_c$ then
            cross parent1 and parent2;
            save new two children in children population;
        else
            copy parent1 and parent2 in children population;
        end
    end
    resolve location conflicts;
                                                        ▷ Mutation
    for all new population do
        foreach gen in chromosome do
            if random_probability[0,1) <$P_m$ then
                mutate gen;
            end
        end
    end
    calculate total cost;
                                                        ▷ Elitism
    if best_children_population >best_old_population then
        save best_children_population as best individual;
    else
        replace worst in children population with best_old_population;
    end
                                                        ▷ Survival Selection
    replace all population with children population;
end
                                                        ▷ End
save best placement;
```

Algorithm 2: Genetic Algorithm implemented for reference. P_c is a crossover constant, P_m is a mutation constant.

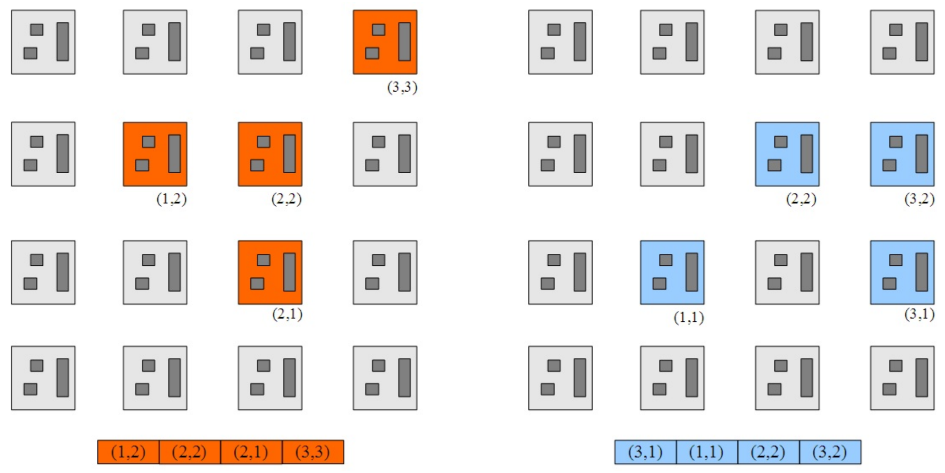


(a) Parents without conflicts.

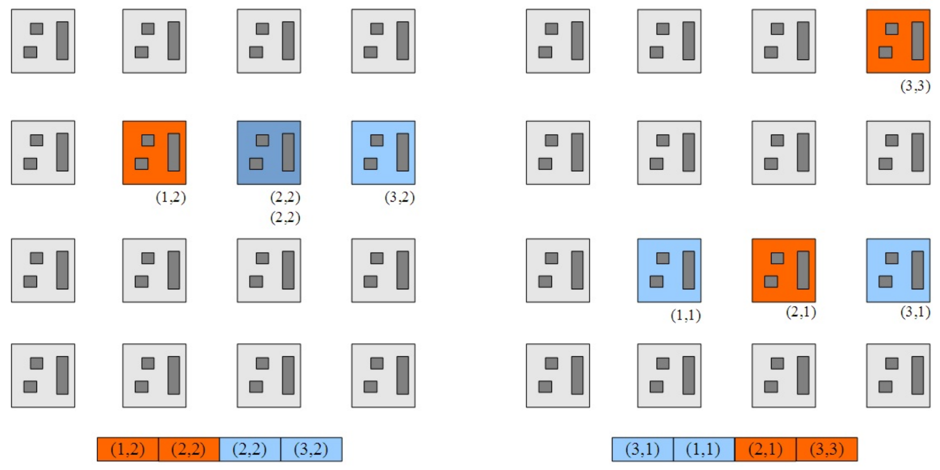


(b) Children without conflicts.

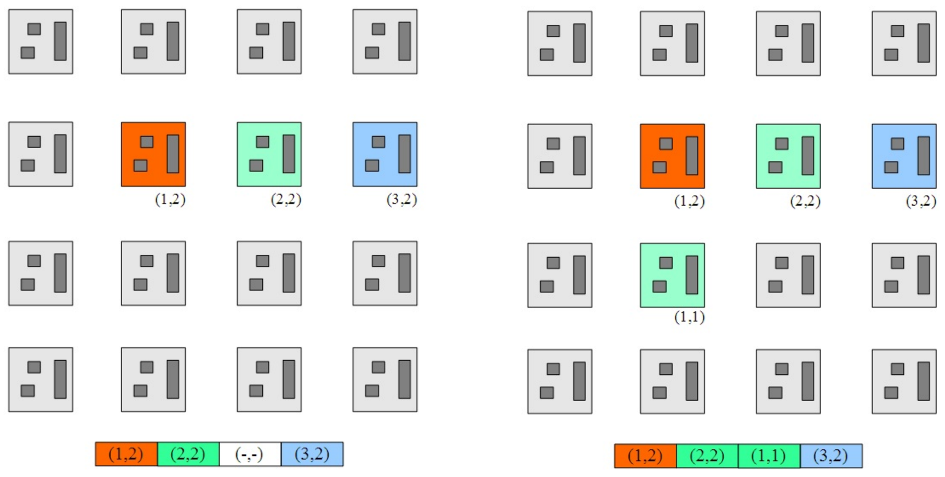
Figure 3.2: Crossover without conflicts.



(a) Parents conflicts.

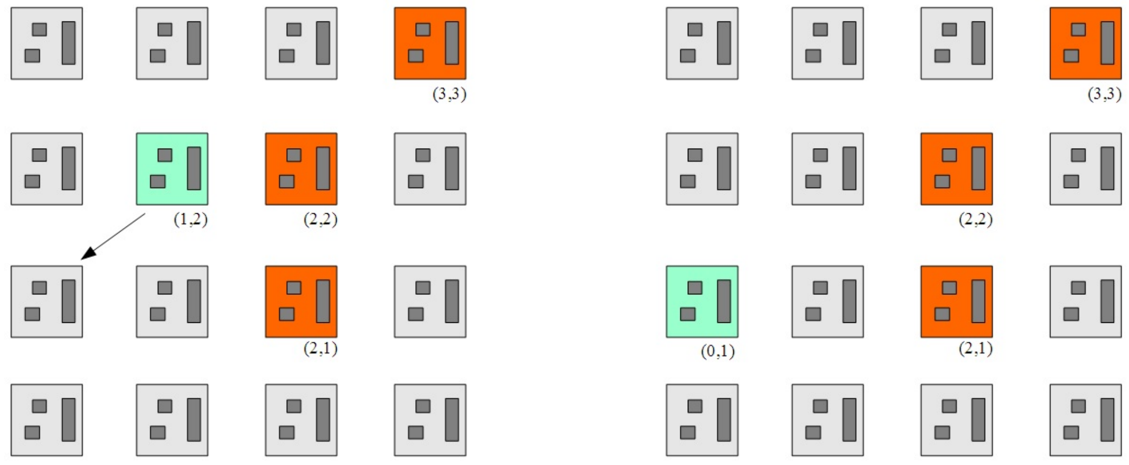


(b) Children. The left child has a conflict in $(2, 2)$.

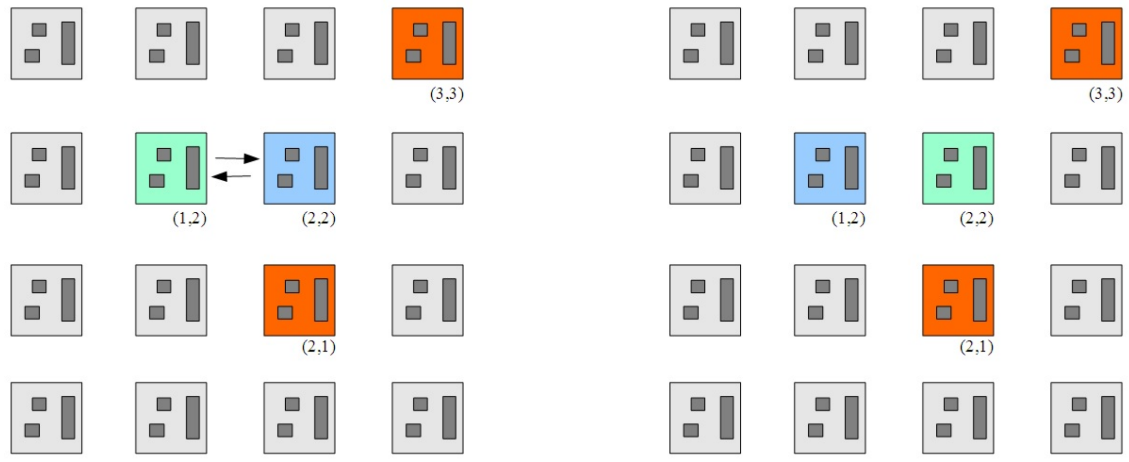


(c) Resolution of conflict in $(2, 2)$.

Figure 3.3: Crossover with conflicts.



(a) To empty place.



(b) Swap.

Figure 3.4: Mutation.

3.1.4 Local Unimodal Sampling Method

To produce good results, tuning the GA's parameters is needed. Two important parameters are P_m and P_c. There are two ways of setting these parameters: (1) sampling span (2) sampling method. In a sampling span a set of experiments are performed within a range of parameters values (e.g. P_m=0,0.2,...1), and the parameter is selected from the best results. The problem here is that these parameters needs a wide fine tuning (e.g. 0.001) and setting a range results in a prohibitive number of experiments. Another way to tune the parameters is to use a sampling method. The simplest case will be to run a set of tests each with a random value of the parameters, and pick the ones with best results. A refined method will be to use an algorithm were a convergence of the parameters to a global or local minimum is found [41].

The method used in this work is Local Unimodal Sampling (LUS) method [42], and its implementation for P_m and P_c is depicted in Algorithm 3. The input of this method are the circuit netlist, the GA's parameters (e.g. P_m) and the FPGA's architectural parameters. The method has two well differentiated parts, the initialization and the main loop. The initialization runs the GA algorithm to find an initial sum of fitness values ($f(\vec{x})$) and also sets the radius, $\vec{d}$, of the search space of the parameters P_m and P_c. The main loop starts calculating the value of the parameters with $\vec{y} = \vec{x} + \vec{a}$ where $\vec{x}$ is the current parameters' value and $\vec{x}$ is a random variation of the parameters within the space $U(-\vec{d}, +\vec{d})$. Using these parameters, the GA is performed in a loop for a set of maximum runs (M) or until the sum of the fitness is worse than the previous one (F). If the final sum of fitness is better than the previous one, the parameters are updated and passed to the next iteration. If the sum of fitness is worst, the parameters' search space is narrowed with $\vec{d} \leftarrow q \cdot \vec{d}$, where q is a gradient parameter with $q = 2^{\frac{-\alpha}{n}}$. A typical value of α is $\alpha = \frac{1}{3}$.

3.2 Routing Algorithm as a New Cost Function

Nowadays, the increase of computation power opens the possibility of using the routing algorithm directly as a cost function. To use the routing algorithm as a new cost function, it is extracted the routing algorithm of the VTR tool and inserted it as a function in the placement algorithm. Every time that this function is called, the placement and its associated structures are loaded. The VPR uses the Pathfinder negotiated congestion routing algorithm [43]. This algorithm routes each net by the shorted path that it can find in the FPGA, an after that the algorithm iterates to resolve constraints (e.g. reuse of the same channel tracks).

```
Input  : circuit netlist, GA parameters, FPGA parameters
Output: Best fitness $f(\vec{x})$ and best GA parameters $\vec{x}$
                                                              ▷ Init
init random parameters $\vec{x} = (P_c, P_m)$ ;
$s \leftarrow 0$;
$j \leftarrow 1$;
while $j \leq M$ do
    $\vec{f_j} \leftarrow GA(\vec{x})$;
    $s \leftarrow s + \vec{f_j}$;
    $j \leftarrow j + 1$;
end
$f(\vec{x}) \leftarrow s$;
Set $\vec{d}$ to cover full space;
$n \leftarrow 1$;
                                                              ▷ Main Loop
while not termination (number iterations) do
    Set random $\vec{a} \sim U(-\vec{d}, +\vec{d})$ ;
    $\vec{y} = \vec{x} + \vec{a}$;
    if $\vec{y} > Max_{range}$ then
        $\vec{y} = Max_{range}$;
    else if $\vec{y} < Min_{range}$ then
        $\vec{y} = Min_{range}$;
    $F = f(\vec{x})$;
    while $j \leq M$ and $s \leq F$ do
        $\vec{f_j} \leftarrow GA(\vec{y})$;
        $s \leftarrow s + \vec{f_j}$;
        $j \leftarrow j + 1$;
    end
    $f(\vec{y}) \leftarrow s$;
    if $f(\vec{y}) < f(\vec{x})$ then
        $\vec{x} \leftarrow \vec{y}$;
        $f(\vec{x}) \leftarrow f(\vec{y})$;
    else
        $\vec{d} \leftarrow q \cdot \vec{d}$;
    end
    $n \leftarrow n + 1$;
end
```

Algorithm 3: Local Unimodal Sampling (LUS) method applied to the Genetic Algorithm. $GA(\vec{x})$ is the placement Genetic Algorithm using the parameters $\vec{x}$, P_c is a crossover constant, P_m is a mutation constant, s is the fitness sum, j is the run-counter, M is the maximum number of runs, $\vec{f_j}$ is the best fitness of the run, $\vec{d}$ is a variable that cover the ranges of the parameters, n is a iteration counter, q is a gradient parameter with $q = 2^{\frac{-\alpha}{n}}$, typical value of α is $\alpha = \frac{1}{3}$.

After the routing is performed, this new cost function returns the critical path to be used as a cost value. This solution is called VTR-R.

On the other hand, as the new cost function is a complex algorithm, it will consume more execution time than the traditional cost function. Therefore, it is investigated another solution using the genetic algorithm of Section 3.1.3 with other function cost (in this work it is called GA-R) in order to try of reducing the number of cost function evaluations needed to achieve the convergence.

It is important to see how many times (evaluations) the cost function is called during the execution of the placement algorithm. The final number of evaluations of the cost function in the SA algorithm is [20]:

$$Eval_{SA} = (num\_gen + 1) \cdot num\_blocks^{1.3333} \tag{3.5}$$

where *num_gen* is the number of generations, and *num_blocks* is the number of blocks of the circuit. The number of blocks includes the LB blocks and the I/O blocks. The number of generations depends on the exit criterion. This exit criterion is satisfied when $T{<}0.005 \cdot \frac{cost}{number\_of\_nets}$, where T is the SA temperature and *number_of_nets* is the number of nets of the circuit. The update temperature is done by means of a tabulated parameter α with $T_{new} = \alpha \cdot T_{old}$. The parameter α depends of the accepted new LB placements per generation divided by the maximum number of blocks movements (refer to [6] for the values of α).

As it can be seen from the equation 3.5, the number of cost function evaluations is not linear in relation to the number of blocks in the circuit. In order to reduce the number of cost function evaluations, the use of the GA will be investigated. In a GA, the number of cost function evaluations ($Eval_{GA}$) depends of the number of generations (num_gen) and population size (pop_size):

$$Eval_{GA} = num\_gen \cdot pop\_size \tag{3.6}$$

The GA is independent of the number of blocks of a circuit. However, a fixed number of generations in the GA must be set in advance by the user.

Chapter 4

Experimental Results and Discussion

This chapter presents and discuss the experimental results with the four algorithms explained in the last chapter. The experiments intend to see the relation between SA and GA, and to investigate the use a routing algorithm as a cost function.

In this chapter, first the used experimental parameters are reported, second the experimental results are shown and discussed.

4.1 Configuration Parameters

There are three types parameters needed for the experiments: FPGA's architecture parameters, genetic algorithm parameters, and LUS method parameters. In the next subsections these parameters are presented.

4.1.1 FPGA's Architecture Parameters

The FPGA's architecture parameters used are shown in Table 4.1 (refer to Section 2.1 for a description of these parameters). In this work, the routing algorithm is not investigated. So, in the experiments of this work, it is used a fixed channel width as it is usual in FPGA placement investigations, e.g. [30]. The other parameters are the default ones of the VTR tool and they emulate the commercial Altera Stratix IV [44]. The Wilton switch block type is used [14].

The characteristics of the circuits used can be seen in Table 4.2. These circuits are provided by the VTR framework in BLIF format [45] and were mapped into LBs using the T-VPack tool [9]. The last column is the minimum array size of the FPGA needed to place the circuit. The circuits are sorted from smaller array size to larger. In general, the array

Parameter	Value used
Tracks per Channel	200
BLEs per CLB	10
Inputs in one BLE	6
Segment distance of a track	4
Ratio of tracks connected to an input	0.15
Ratio of tracks connected to an output	0.1
Switch block type	3 (Wilton)

Table 4.1: FPGA architecture parameters used in the experiments.

Circuit	LUTs	FFs	In	Out	LB	I/O	Nets	Array Size
styr	238	5	10	10	15	20	105	4x4
planet	266	6	8	19	17	27	127	5x5
s1238	292	18	15	14	18	29	148	5x5
vda	253	0	17	39	19	56	176	5x5
daio-rec	311	81	16	46	19	62	230	5x5
mm30a	294	90	34	30	21	64	230	5x5
ecc	291	109	12	14	22	26	178	5x5
ex4p	148	0	84	28	22	112	206	5x5
C2670	214	0	157	64	15	221	305	7x7
rot	242	0	135	107	17	242	293	8x8
x3	255	0	135	99	20	234	281	8x8
i7	103	0	199	67	11	266	266	9x9
frg2	347	0	143	139	26	282	342	9x9

Table 4.2: Circuit characteristics and array size used for the placement.

size determines the time needed to place the circuit. But there are other parameters that affect the complexity of the circuits, such as the number of LBs, I/O or nets.

The placement can be for LB and I/O blocks, or it is possible to fix the placement for the I/O blocks (or LB blocks) and perform the placement only for LB blocks (or I/O blocks). In the experiments, it is fixed the placement of the I/O blocks. The I/O blocks placement file is found performing previously a SA placement.

Parameter	Description	GA-N	GA-R
	Size	70	70
P_c	Crossover Probability	0.12	0.5
P_m	Mutation Probability	0.03	0.04
	Tournament Size	2	2
λ	Time trade-off	0.5	-
	Generations	1000	200[1]

Table 4.3: Parameters used in the Genetic Algorithm. [1] The circuit *styr* has 120 generations because doesn't need more generation to have a good quality result.

4.1.2 Genetic Algorithm Parameters

The parameters used in the GAs are shown in Table 4.3. The population size is fixed to 70. This population's size shows good results in the experiments. GA-R has 200 generations, except in the *styr*. The circuit *styr* is small and a good quality placement can be achieved with 120 generations.

To see the correct critical path time with the algorithms that use the traditional cost function, it is performed the routing step with the placement files coming from the placement.

The initial temperature of the SA is $init\_temperature = 20 \cdot std\_dev$ where the standard deviation, std_dev, is calculated with the cost variation of moving blocks randomly. In the SA with the traditional cost function, it is used $\lambda = 0.5$. To find λ, it was performed a set of tests with λ going from 0 to 1 (see Table 4.4), and it was found that $\lambda = 0.5$ was the value with best final critical path. Similar results can be found in literature (e.g. [7]).

The GA uses a tournament size of two individuals, one-point crossover, and mutation based in LB permutations. In the GA it is needed to set the number of generations, the probability of mutations (P_m), and the probability of crossover (P_c). It was used an exploration method for finding P_m and P_c (this method will be discussed in the next section). The converge of the GA's algorithm was taken into account to set set the number of generations.

A workstation with two Intel E7 Xeon processors with 32GB of RAM has been used for the experiments.

Each circuit was run 30 times. To create a different heuristic in each run, it was changed the deterministic random function of the VTR tool to a semi-random function ($rand\_s$ [39]).

Circuit	**Critical path ± SD (ns) per λ**				
	0	0.2	0.5	0.8	1
styr	3.14 ± 0.07	3.11 ± 0.10	3.08 ± 0.16	3.00 ± 0.10	2.97 ± 0.07
planet	2.89 ± 0.06	2.90 ± 0.05	2.59 ± 0.02	2.85 ± 0.08	2.85 ± 0.07
s1238	4.45 ± 0.11	4.38 ± 0.07	4.45 ± 0.09	4.43 ± 0.06	4.43 ± 0.06
vda	3.21 ± 0.05	3.24 ± 0.06	3.24 ± 0.06	3.27 ± 0.08	3.27 ± 0.10
daio-rec	4.03 ± 0.05	4.00 ± 0.07	3.97 ± 0.08	3.93 ± 0.12	3.87 ± 0.11
mm30a	13.18 ± 0.07	13.17 ± 0.09	13.16 ± 0.13	13.13 ± 0.10	13.08 ± 0.08
ecc	3.01 ± 0.09	3.02 ± 0.09	3.01 ± 0.07	3.01 ± 0.07	2.99 ± 0.08
ex4p	2.79 ± 0.07	2.75 ± 0.03	2.74 ± 0.04	2.76 ± 0.03	2.77 ± 0.04
C2670	3.71 ± 0.01	3.70 ± 0.01	3.74 ± 0.08	3.73 ± 0.07	3.96 ± 0.05
rot	3.82 ± 0.05	3.78 ± 0.03	3.78 ± 0.02	3.77 ± 0.01	3.83 ± 0.09
x3	2.47 ± 0.05	2.56 ± 0	2.56 ± 0	2.56 ± 0	2.56 ± 0
i7	1.93 ± 0.00	1.93 ± 0.00	1.93 ± 0.00	1.93 ± 0.00	1.93 ± 0.00
frg2	3.47 ± 0.07	3.32 ± 0.03	3.31 ± 0.01	3.31 ± 0.01	3.31 ± 0.01

Table 4.4: VTR evolution of λ. Critical path averaged over 30 runs. SD = Standard Deviation.

4.1.3 LUS Method Parameters

The method has two parameters that should be set: M, α and the number of iterations of the main loop. The parameter M was set to have an enough variation and to fit a time-frame of 48 hours for the total run of the LUS method. To adjust α, a program was written to see how it behaves α in a search range $\vec{d} \leftarrow q \cdot \vec{d}$ where $q = 2^{\frac{-\alpha}{n}}$. The intention was to converge the LUS method in around 200 iterations. If the parameter values are kept fix (e.g. $\vec{x} = (0.5, 0.5)$), with $\alpha = \frac{1}{3}$ is obtained the graphic for the range d of Figure 4.1 (a). With this α is possible to plot a parameter value by iterations. The equation for a new parameter's value is $\vec{y} = \vec{x} + \vec{a}$ where $\vec{x}$ are the current parameter's values and $\vec{a}$ is a random value within the radius d (note that it is random). Knowing this, a plot for one parameter is shown in Figure 4.1 (b). It can be seen that there is no convergence in 200 iterations. Figure 4.1 (c) and Figure 4.1 (d) show the same experiment with $\alpha = \frac{1}{5}$. In this case, it is possible to see that the parameter value converges.

In the previous experiments, the parameter value $\vec{x}$ of $\vec{y} = \vec{x} + \vec{a}$ was always the same in each iteration ($\vec{x} = (0.5, 0.5)$). In another experiment it was allowed to have a variation of the $\vec{x}$ and it was used the conditional statement (set randomly per iteration) of the lines 28 to 33 of the Algorithm 3. The results for two tests are shown in Figure 4.1 (e) and (f). The important conclusion from these two graphics is that within 200 iterations parameter value can move in all the search range. Therefore, for the next experiments it will be used $\alpha = \frac{1}{5}$ and 200 iterations of the main LUS method loop.

The LUS method to find P_m and Pc used the circuit $s1238$, and it was set to iterate 200 times with 10 runs of the genetic algorithm (with 200 generations). Table 4.5 shows the final $P_{m,rslt}$ and $P_{c,rslt}$ results together with the parameters used to perform the LUS. As there are two different algorithms (GA-N and GA-R), two different experiments were done. The resulting P_m is similar for both algorithms. The differences with P_c is because the crossover is less critical for the placement algorithm.

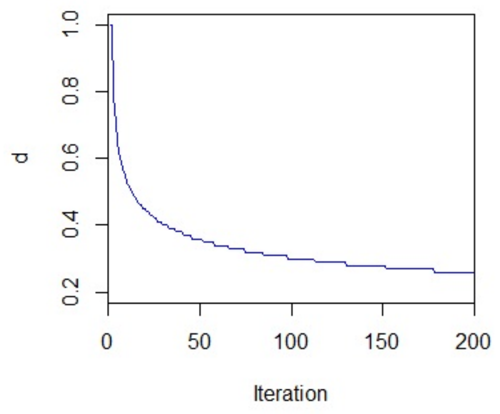
Alg.	λ	$P_{c,init}$	$P_{m,init}$	Gen.	Size	LUS-Runs	α	LUS-Iter.	$P_{c,rslt}$	$P_{m,rslt}$
GA-N	0.5	0.1	0.001	500	70	8	$\frac{1}{5}$	200	0.1153	0.0336
GA-R	-	0.1	0.001	200	20	5	$\frac{1}{5}$	200	0.4997	0.0434

Table 4.5: Crossover probability ($P_{c,rslt}$) and mutation probability ($P_{m,rslt}$) obtained with LUS results with the s1238 circuit. Alg.=Algorithm, Gen.=Generations of one run.

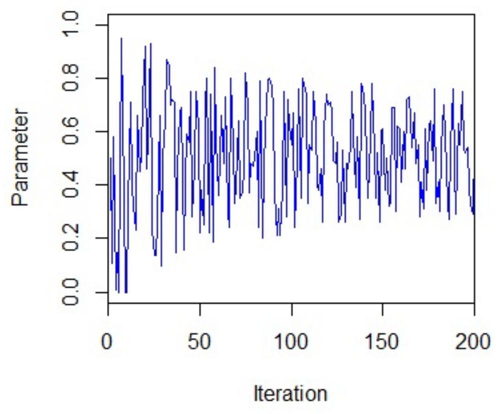
4.2 Experimental Results and Discussion

The critical path results are shown in Table 4.6. As it can be seen in the mentioned table, the use of the new cost function (VTR-R and GA-R) improves the average critical path in comparison with the traditional cost function (VTR-N and GA-N). This comes at the expenses of the required execution time (Table 4.8), that is in the order of 10^5 or 10^6 higher. The critical paths between VTR-R and GA-R are similar. But a noticeable reduction in the execution time can be observed with GA-R against VTR-R. From Table 4.8 is also possible to see that at measure that the array size is augmented the time required for the placement is also higher.

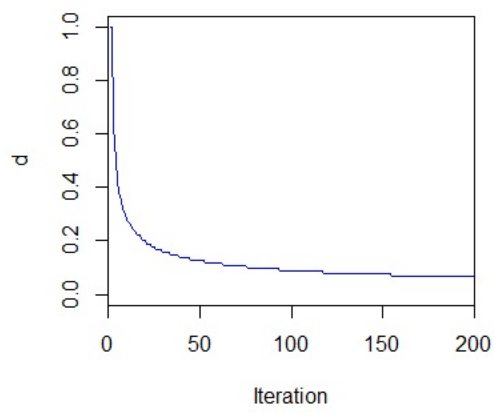
In order to compare the experimental results obtained in Table 4.6 from a statistical point of view, it is performed a nonparametric bootstrap hypothesis test [46]. A nonparametric test was needed because the data associated to each circuit rejected the null hypothesis of normality (Shapiro-Wilk test [47]). The significance level was set to $\alpha = 0.05$. As it can be seen from the p-values obtained in Table 4.7, the null hypothesis (there is no difference between the two population means) is always rejected for all the cases of VTR-R vs VTR,



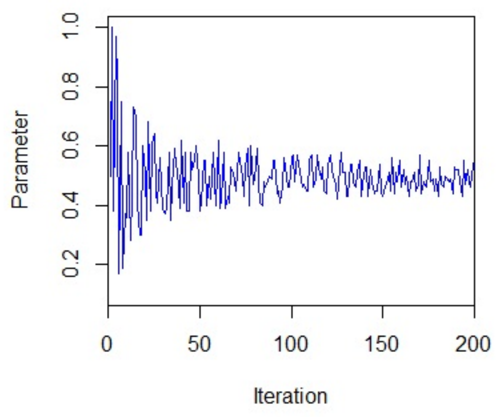
(a) $\alpha = \frac{1}{3}$



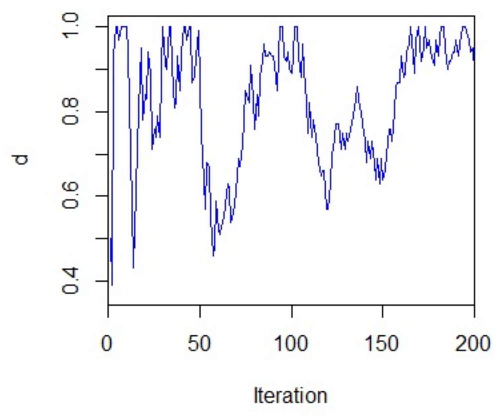
(b) One parameter value with $\alpha = \frac{1}{3}$



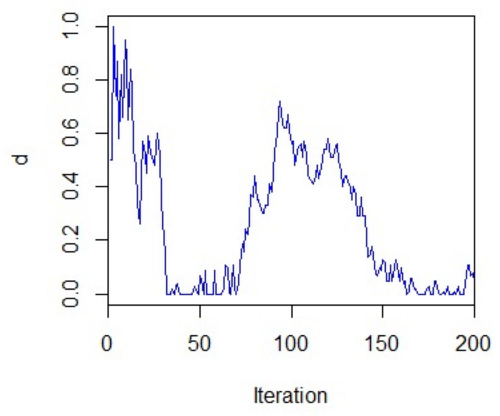
(c) $\alpha = \frac{1}{5}$



(d) One parameter value with $\alpha = \frac{1}{5}$



(e) Test 1



(f) Test 2

Figure 4.1: (a) to (d): Adjustment of LUS metod parameters with different α and keeping the parameter value constant ($\vec{x} = (0.5, 0.5)$) in each iteration. (e) and (f): Variation of one value parameter in two random tests of the LUS method.

GA-R vs VTR, and GA-N vs GA-R. On the other hand, the null hypothesis cannot be rejected in all the cases of GA-R vs VTR-R. Therefore, from the hypothesis test results, it can be said that there exists statistical evidence to affirm that the critical path values obtained for VTR-R and GA-R are better than those obtained for VTR. However, it can be assumed similar critical path results between GA-R and VTR-R. In the case of GA-N vs VTR, there are circuits that null hypothesis is rejected and others not.

Figure 4.2 shows an example of the convergence of three circuits (*planet*, *s*1238 and *ecc*). It is possible to observe the typical random-walk of the SA. This happens because the temperature parameter of the SA forces the acceptance of placement solutions even with worst cost. Looking at these graphics, there is a question: is possible to stop the algorithm VTR-R before it reaches the final iteration? As it can be seen in Figure 4.3, there is the possibility of improving the critical path in the last iterations. In the same figure it is possible to see also one of the problems with a SA algorithm. A better critical path is found around the iteration 45, but because the temperature forces to accept other placement solution, the placement solution deteriorates. To solve this problem, it would be needed to modify the SA algorithm with the possibility of storing the best solution for all the iterations, and at the end of the iteration choose the better solution.

An open question is how the GA-N performs in comparison with VTR-N. The use of GAs with the traditional cost function has been investigated in [29]. In this paper, the author shows that a GA with the traditional cost function is not better than a VTR-N algorithm. The results with GA-N in of this work corroborate the results of [29]. As the number of cost function evaluations in the VTR-N can be higher than the GA-N, there is a question if it is possible to get a better execution time using the GA-N. The tests with GA-N show it is not possible. This is because our GA-N is not optimized for execution time and, additionally, the traditional cost function was designed for the VTR-N. For instance, the traditional cost function has a normalization that it is keep constant for several evaluations during a generation. This is not possible with the GA-N. Another problem, already mentioned in Chapter 2, is that the traditional cost function produces inconsistent values of the critical path (see Figure 1.5). This is not a big problem with the VTR-N where the cost function is evaluated for one single LB movement (an error in one cost function evaluation is compensated with the next evaluation). But, with the GA-N, the cost function is evaluated after several LB movements (with crossover and mutations). So, if the best individual is found with a bad cost function prediction, it will pass to the next generation and produce a wrong offspring. This suggest that if it is not used the routing algorithm as a cost function, like in this work, a better model for the cost function should be found in order to be used by a GA.

The percentage differences of the critical path are shown in Table 4.9. The second and third columns compare VTR-N respect the algorithms using the routing as a cost function. The two last columns compares the same but with GA-N. From the comparisons, we can see that VTR-N is better than GA-N. VTR-R and GA-R are similar and always better than VTR-N and GA-N.

Table 4.10 shows the cost function evaluations. The VTR-R algorithm needs more evaluations than the VTR-N algorithm. This is because the termination criterion of the VTR-N is fulfilled when not better placements are found. Meanwhile the fine granularity of the cost function in VTR-R reaches improvements in new generations. It is also possible to see that the number of evaluations in GA-N and GA-R is a constant defined by the Equation 3.6. Note that the GA-R needs fewer cost function evaluations than GA-N, VTR-N and VTR-R to get a better or similar result.

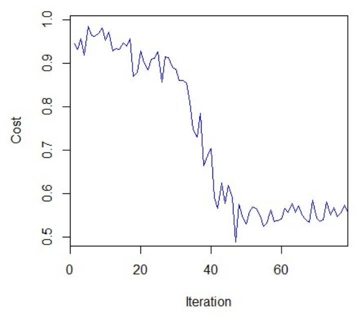
Besides of the constant number of evaluations, another advantage of GAs (though not explored in this work) is that the algorithm can be easily paralleled [48] [49]. The SA has only one thread where a new solution depends on the previous one. On the other hand, in a GA, each individual of a generation can be run independently in a thread. As an example, a population of 100, can be run in parallel each individual in a cluster of 100 cores. In this case, the execution time will depend only of the number of generations (according to eq. 3.6). Note that a circuit with higher number of blocks will require also higher execution time in the VTR-R and the GA-R because the routing is also much complex.

Circuit	Crit.Path(ns)			
	VTR-N ± SD	VTR-R ± SD	GA-N ± SD	GA-R ± SD
styr	3.08 ± 0.16	2.71 ± 0.02	3.07 ± 0.05	2.72 ± 0.02
planet	2.85 ± 0.08	2.59 ± 0.02	2.94 ± 0.05	2.60 ± 0.02
s1238	4.44 ± 0.09	4.11 ± 0.02	4.50 ± 0.09	4.11 ± 0.03
vda	3.24 ± 0.06	2.96 ± 0.03	3.46 ± 0.11	2.97 ± 0.03
daio-rec	3.97 ± 0.08	3.47 ± 0.02	4.15 ± 0.12	3.49 ± 0.03
mm30a	13.16 ± 0.13	12.47 ± 0.03	13.21 ± 0.16	12.50 ± 0.04
ecc	3.01 ± 0.07	2.75 ± 0.03	3.08 ± 0.07	2.75 ± 0.03
ex4p	2.74 ± 0.04	2.60 ± 0.02	2.89 ± 0.09	2.64 ± 0.03
C2670	3.74 ± 0.08	3.41 ± 0.03	4.10 ± 0.10	3.53 ± 0.07
rot	3.78 ± 0.03	3.57 ± 0.04	3.95 ± 0.20	3.59 ± 0.06
x3	2.56 ± 0.00	2.23 ± 0.03	3.01 ± 0.32	2.20 ± 0.02
i7	1.93 ± 0.00	1.67 ± 0.01	1.92 ± 0.05	1.66 ± 0.01
frg2	3.31 ± 0.01	3.00 ± 0.03	4.00 ± 0.20	3.14 ± 0.05

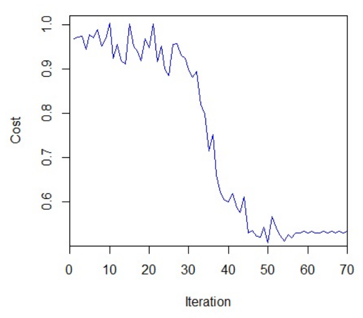
Table 4.6: Experimental results for critical path with the standard deviation (SD) averaged over 30 runs.

Circuit	VTR-R vs VTR-N	GA-R vs VTR-N	GA-R vs VTR-R	GA-N vs VTR-N	GA-N vs GA-R
styr	0.0001	0.0001	0.6991	0.3255	0.0001
planet	0.0001	0.0001	0.1104	0.0001	0.0001
s1238	0.0001	0.0001	1	0.4934	0.0001
vda	0.0001	0.0001	1	0.0001	0.0001
daio-rec	0.0001	0.0001	0.0594	0.0001	0.0001
mm30a	0.0001	0.0001	0.0645	0.08819	0.0001
ecc	0.0001	0.0001	1	0.0006	0.0001
ex4p	0.0001	0.0001	1	0.0001	0.0001
C2670	0.0001	0.0001	1	0.0001	0.0001
rot	0.0001	0.0001	1	0.0001	0.0001
x3	0.0001	0.0001	1	0.0001	0.0001
i7	0.0001	0.0001	1	0.1589	0.0001
frg2	0.0001	0.0001	1	0.0001	0.0001

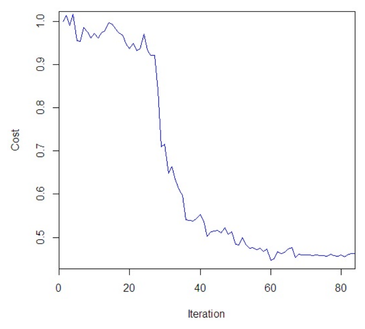
Table 4.7: Results of the non-parametric Bootstrap hypothesis test (p-values).



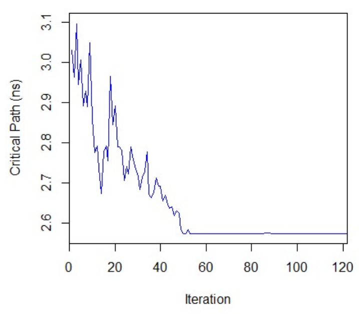
(a) VTR planet



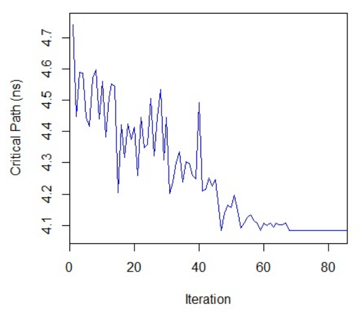
(b) VTR 1238



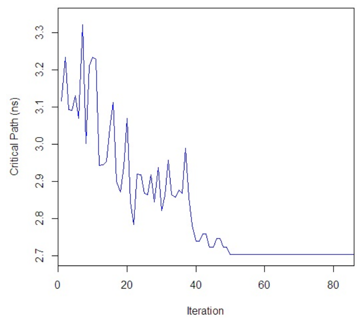
(c) VTR ecc



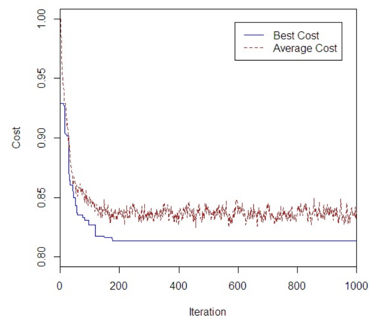
(d) VTR-R planet



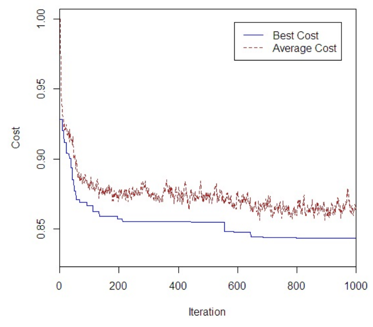
(e) VTR-R s1238



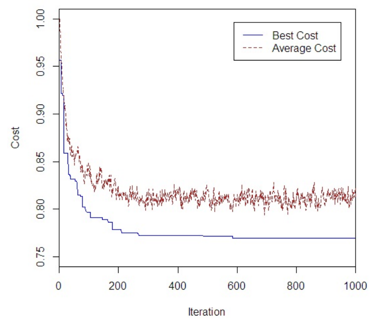
(f) VTR-R ecc



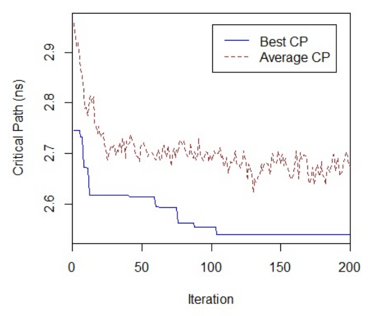
(g) GA-N planet



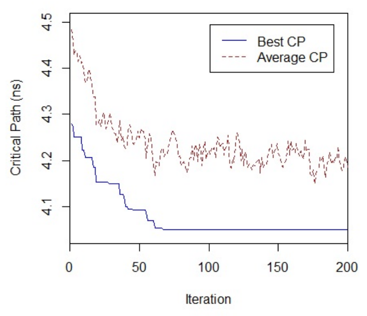
(h) GA-N s1238



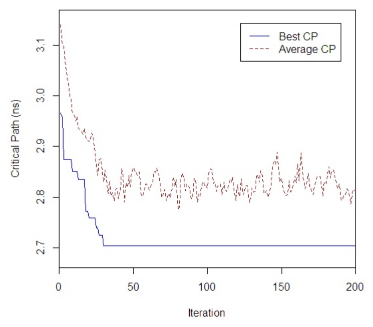
(i) GA-N ecc



(j) GA-R planet



(k) GA-R s1238



(l) GA-R ecc

Figure 4.2: Plots of convergence of the best result of the *planet*, *s*1238 and *ecc* circuits.

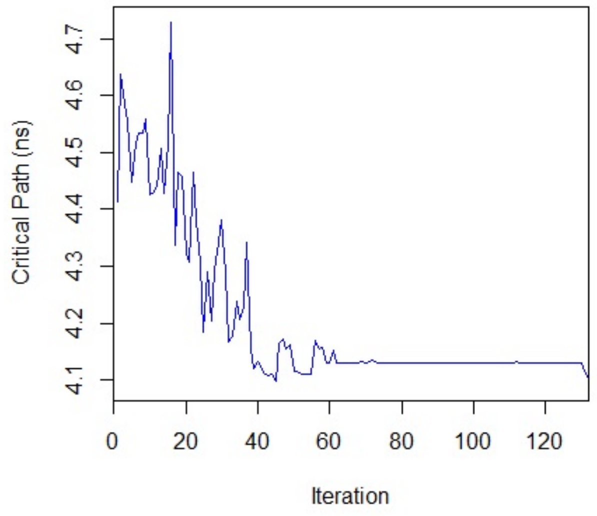


Figure 4.3: Plot of convergence of a $s1238$ circuit with SA with the new cost function (VTR-R).

Circuit	Exec.Time(sec)			
	VTR-N ± SD	VTR-R ± SD	GA-N ± SD	GA-R ± SD
styr	0.37 ± 0.05	1853 ± 377	46 ± 5	1365 ± 110
planet	0.65 ± 0.05	5031 ± 1127	50 ± 1	3592 ± 328
s1238	1.71 ± 0.62	5906 ± 1510	71 ± 24	3969 ± 310
vda	1.85 ± 0.78	8977 ± 1720	195 ± 80	6318 ± 968
daio-rec	0.95 ± 0.04	11960 ± 750	68 ± 2	3161 ± 134
mm30a	0.98 ± 0.05	11802 ± 1390	75 ± 2	3395 ± 121
ecc	0.78 ± 0.03	8617 ± 2051	203 ± 68	5902 ± 897
ex4p	3.35 ± 0.85	25236 ± 7181	220 ± 87	6536 ± 1713
C2670	12.12 ± 8.74	90184 ± 20692	167 ± 79	9199 ± 2173
rot	15.42 ± 10.06	137549 ± 23390	202 ± 51	11470 ± 3230
x3	11.52 ± 3.13	170177 ± 10699	192 ± 73	12311 ± 2171
i7	19.81 ± 4.38	241126 ± 15327	117 ± 39	11180 ± 2942
frg2	14.68 ± 2.17	258359 ± 18465	275 ± 123	12229 ± 4328

Table 4.8: Experimental results for execution time with the standard deviation (SD) averaged over 30 runs.

Circuit	Diff. Crit.Path(%)			
	$\Delta_{VTR-N\ vs\ VTR-R}$	$\Delta_{VTR-N\ vs\ GA-R}$	$\Delta_{GA-N\ vs\ VTR-R}$	$\Delta_{GA-N\ vs\ GA-R}$
styr	-11.68	-11.62	-12.77	-12.70
planet	-9.14	-8.83	-13.27	-12.89
s1238	-7.45	-7.45	-9.32	-9.32
vda	-8.65	-8.56	-16.65	-16.53
daio-rec	-12.62	-12.23	-19.45	-18.91
mm30a	-5.24	-5.02	-5.93	-5.69
ecc	-8.79	-8.81	-12.12	-12.15
ex4p	-5.03	-3.67	-11.08	-9.52
C2670	-8.79	-5.70	-20.29	-16.35
rot	-5.86	-4.77	-11.01	-9.74
x3	-13.01	-14.01	-35.12	-36.71
i7	-13.39	-13.79	-14.96	-15.49
frg2	-9.28	-4.94	-33.50	-27.40

Table 4.9: Differences for critical path.

Circuit	Number of Evaluations.			
	VTR-N ± SD	VTR-R ± SD	GA-N ± SD	GA-R ± SD
styr	7969 ± 434	12510 ± 2408	70000 ± 0	8400 ± 0
planet	11754 ± 447	19207 ± 2737	70000 ± 0	14000 ± 0
s1238	12603 ± 402	21470 ± 4077	70000 ± 0	14000 ± 0
vda	25250 ± 903	23680 ± 898	70000 ± 0	14000 ± 0
daio-rec	28279 ± 1154	52662 ± 815	70000 ± 0	14000 ± 0
mm30a	30024 ± 902	47319 ± 3939	70000 ± 0	14000 ± 0
ecc	14002 ± 582	24466 ± 510	70000 ± 0	14000 ± 0
ex4p	58678 ± 1513	75667 ± 14270	70000 ± 0	14000 ± 0
C2670	127713 ± 2805	172182 ± 35844	70000 ± 0	14000 ± 0
rot	153654 ± 2791	188909 ± 38129	70000 ± 0	14000 ± 0
x3	146850 ± 2571	247564 ± 4556	70000 ± 0	14000 ± 0
i7	170669 ± 4067	276803 ± 25262	70000 ± 0	14000 ± 0
frg2	192546 ± 2660	318533 ± 20008	70000 ± 0	14000 ± 0

Table 4.10: Number of cost function evaluations with the standard deviation (SD) averaged over 30 runs.

Chapter 5

Conclusions and Future Work

In this Master's thesis the critical path optimization in the FPGA's placement has been investigated. The comparisons allow to see the differences with SA and GA in the FPGA's placement problem. It is found that the minimization of the traditional cost function used in the SA not always produce a minimal critical path. To alleviate this problem, it has been proposed to use the routing algorithm as a cost function. From the experiments, it can be seen that VTR-N is a bit better than GA-N. This can be because several reasons. It can be that the code and the adjustment of its parameters is not optimal causing bad quality results. Most likely GA-N produce worst results because traditional cost is optimized for SA. Experimental results show that the quality of the placement is improved using the routing algorithm as a cost function (VTR-R). Observed drawback is the longer execution time required. To reduce the execution time with the new cost function, it has been proposed the use of a GA (GA-R). It is found that the GA-R improves the execution time maintaining a competitive critical path. The new cost function will be useful in those cases where a minimum critical path is needed.

While this Master's thesis has shown the benefits of using GA and routing algorithm as a cost function, many opportunities are still open to improve the results: (1) The inherent parallelism of GAs can be exploited for improving the execution time, (2) in this work a fix number of generations was set, an exit criteria depending of the algorithm's convergence can be investigated, (3) a better routing algorithm as a cost function can be investigated (instead of the default used in this work), (4) to improve the placement quality, the routing parameters (e.g. routing effort, number of tracks, etc) can be investigated.

Bibliography

[1] Stephen Brown, Robert Francis, Jonathan Rose, and Zvonko Vranesic. *Field-Programmable Gate Arrays*, volume 180. Springer Science & Business Media, 1992.

[2] Deming Chen, Jason Cong, Peichen Pan, et al. FPGA design automation: A survey. *Foundations and Trends® in Electronic Design Automation*, 1(3):195–330, 2006.

[3] Zainalabedin Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[4] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[5] Xilinx Inc. 7 series FPGAs overview, may. 27, 2015 (version 1.17).

[6] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[7] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for FPGAs. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 203–213. ACM, 2000.

[8] Chih-Liang Eric Cheng. RISA: accurate and efficient placement routability modeling. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 690–695. IEEE Computer Society Press, 1994.

[9] VTR tool, University of Toronto. `http://https://github.com/verilog-to-routing/vtr-verilog-to-routing/`. Accessed: 2016-05-17.

[10] Mingjie Lin and John Wawrzynek. Improving FPGA placement with dynamically adaptive stochastic tunneling. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(12):1858–1869, 2010.

[11] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6, 2014.

[12] MCNC LGSynth93. Benchmarks. *Obtained from http://www. eecg. toronto. edu/˜ lemieux/sega/ccts_blif. tar.*

[13] FPGA Actel. Data book and design guide. *Actel Corp*, 955, 1995.

[14] Imran Masud and Steven Wilton. A new switch block for segmented FPGAs. In *Field programmable logic and applications*, pages 274–281. Springer, 1999.

[15] Herman Schmit and Vikas Chandra. FPGA switch block layout and evaluation. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 11–18. ACM, 2002.

[16] Hongbing Fan, Jiping Liu, Yu-Liang Wu, and Chak-Chung Cheung. On optimal hyper-universal and rearrangeable switch box designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(12):1637–1649, 2003.

[17] The Xilinx website, xilinx inc. `http://www.xilinx.com/`. Accessed: 2016-04-05.

[18] The Altera website, altera inc. `http://www.altera.com/`. Accessed: 2017-04-05.

[19] Carl Sehen. An improved simulated annealing algorithm for row-based placement. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 478–481, 1987.

[20] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for deep-submicron FPGAs*, volume 497. Springer Science & Business Media, 2012.

[21] Sudip Nag and Rob Rutenbar. Performance-driven simultaneous placement and routing for FPGA's. *IEEE Transactions on Computer-Aided design of integrated circuits and systems*, 17(6):499–518, 1998.

[22] Emilio Vasconcelos de Lima, Antônio Carlos Cavalcanti, and Luis dos Anjos Formiga Cabral. A new approach to VPR tool's FPGA placement. In *Proceedings of the World Congress on Engineering and Computer Science (WCECS)*, 2007.

[23] Ali Asghar and Husain Parvez. An improved diffusion based placement algorithm for reducing interconnect demand in congested regions of FPGAs. *International Journal of Reconfigurable Computing*, 2015:8, 2015.

[24] Zoltan Baruch, Octavian Creţ, and Horia Giurgiu. Genetic algorithm for FPGA placement. In *Proceedings of the 12th International Conference on Control Systems and Computer Science (CSCS12)*, volume 2, pages 121–126, 1999.

[25] Rahman Venkatraman and Lalit Patnaik. An evolutionary approach to timing driven FPGA placement. In *Proceedings of the 10th Great Lakes symposium on VLSI*, pages 81–85. ACM, 2000.

[26] Manuel Rubio del Solar, Juan Antonio Gomez Pulido, Juan Manuel Sanchez Perez, and Miguel Angel Vega Rodriguez. Genetic algorithms for solving the placement and routing problem of an FPGA with area constraints. *Proc. IEEE ISDA*, pages 31–35, 2004.

[27] Siva Nageswara Rao Borra, Annamalai Muthukaruppan, Sivaprakasam Suresh, and Viri Kamakoti. A novel approach to the placement and routing problems for field programmable gate arrays. *Applied Soft Computing*, 7(1):455–470, 2007.

[28] Mei Yang, Any Almaini, and Liu Wang. FPGA placement by using a genetic algorithm. *Engineer IT*, 6(1), 2007.

[29] Peter Jamieson. Revisiting genetic algorithms for the FPGA placement problem. In *GEM*, pages 16–22. Citeseer, 2010.

[30] Peter Jamieson. Exploring inevitable convergence for a genetic algorithm persistent FPGA placer, 2011.

[31] Robert Collier, Christian Fobel, Laura Richards, and Gary Grewal. A formal and empirical analysis of recombination for genetic algorithm-based approaches to the FPGA placement problem. In *Electrical & Computer Engineering (CCECE), 2012 25th IEEE Canadian Conference on*, pages 1–6. IEEE, 2012.

[32] Peter Jamieson, Farnaz Gharibian, and Lesley Shannon. Supergenes in a genetic algorithm for heterogeneous FPGA placement. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 253–260. IEEE, 2013.

[33] Dionisios Diamantopoulos, Kostas Siozios, Sotirios Xydis, and Dimitrios Soudris. A framework for supporting parallel application placement onto reconfigurable platforms. In *Proceedings of the PARMA Workshop, HiPEAC Conference, Berlin, Germany*, 2013.

[34] Yonghong Xu and Mohammed AS Khalid. QPF: efficient quadratic placement for FPGAs. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 555–558. IEEE, 2005.

[35] Marcel Gort and Jason Anderson. Analytical placement for heterogeneous FPGAs. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 143–150. IEEE, 2012.

[36] Venu Gudise and Ganesh Venayagamoorthy. FPGA placement and routing using particle swarm optimization. In *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pages 307–308. IEEE, 2004.

[37] Wenyao Xu, Kejun Xu, and Xinmin Xu. A novel placement algorithm for symmetrical FPGA. In *ASIC, 2007. ASICON'07. 7th International Conference on*, pages 1281–1284. IEEE, 2007.

[38] Haim Akbarpour, Giri Karimi, and Ahmed Sadeghzadeh. Discrete multi objective particle swarm optimization algorithm for FPGA placement. *International Journal of Engineering Transactions C: Aspects*, 28(3):410–418, 2015.

[39] rand_s , MSDN Microsoft Website. `http://stackoverflow.com/q/398299`. Accessed: 2016-05-17.

[40] Looping in a spiral, stack overflow website. `https://msdn.microsoft.com/en-us/library/sxtz2fa8.aspx`. Accessed: 2016-05-17.

[41] Roland White. A survey of random methods for parameter optimization. *Transactions of the Society for Computer Simulation*, 17(5):197–205, 1971.

[42] Magnus Erik Hvass Pedersen and Andrew John Chipperfield. Local unimodal sampling. *HL0801 Hvass Laboratories*, 2008.

[43] Carl Ebeling, Larry McMurchie, Scott Hauck, and Steven Burns. Placement and routing tools for the Triptych FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(4):473–482, 1995.

[44] David Lewis, Elias Ahmed, David Cashman, Tim Vanderhoek, Chris Lane, Andy Lee, and Philip Pan. Architectural enhancements in Stratix-III$^{\text{TM}}$ and Stratix-IV$^{\text{TM}}$. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–42. ACM, 2009.

[45] Berkeley logic interchange format (BLIF). Oct Tools Distribution 2, 1992.

[46] Joseph Romano et al. Bootstrap and randomization tests of some nonparametric hypotheses. *The Annals of Statistics*, 17(1):141–159, 1989.

[47] Samuel Sanford Shapiro and Martin Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[48] Erick Cantú-Paz. A survey of parallel genetic algorithms. In *Calculateurs paralleles*. Citeseer, 1998.

[49] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.