CONSTRUCTIVE/ITERATIVE BASED TECHNIQUE FOR FPGA PLACEMENT

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

XIAOJUN BAO

In partial fulfilment of requirements

for the degree of

Master of Science

August, 2004

©Xiaojun Bao, 2004

ABSTRACT

CONSTRUCTIVE/ITERATIVE BASED TECHNIQUE FOR FPGA PLACEMENT

Xiaojun Bao

University of Guelph, 2004

Advisor:

Professor Shawki Areibi

Professor Dilip Banerji

Today the logic capacity of Field-Programmable Gate Arrays (FPGAs) has increased dramatically (up to 10-million gates) that prohibitively long compile times may adversely affect instant manufacturability of FPGAs and become intolerable to users seeking very high speed compile. This thesis presents several heuristic techniques and investigates the effectiveness and efficiency of heuristics and meta-heuristics for FPGA placement. In constructive based heuristics, Cluster Seed Search (CSS) is developed to improve averagely random initial solutions by 19%; GRASP and a Partitioning based method are also implemented and achieve 25% and 44% improvement respectively. In iterative based heuristics, an enhanced local search technique is implemented in two forms: Simple Local Search (SLS) and Immediate Neighbourhood Local Search (INLS), which both achieve 50% improvement quickly. A Tabu Search (TS) technique and a Genetic Algorithm approach are also implemented to further enhance solution quality. Results obtained indicate that both Tabu Search and Genetic Algorithms can enhance solution for FPGA placement

and produce on average 74% and 20% improvement in reasonable time.

Acknowledgements

My sincere thanks go to my advisor, Prof. Areibi for his great support and advice throughout this research. Without his help, this work would never have been possible.

I would also like to express my appreciation to my co-advisor, Prof. Banerji for advising and revising this thesis with great meticulosity.

I am grateful to Mr. Don Cavanaugh and my friend Sharlene Shuizing for providing excellent proofreading.

To

my wife, Suling Liu

whose love and encouragement helped accomplish this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The development of a typical Very Large Scale Integration (VLSI) circuit includes a design phase, utilizing Computer-Aided Design (CAD) tools, fabrication, verification and testing. Since the complexity of VLSI design has been growing at an exponential rate in the last decade, the efficiency of each step involved becomes essential for a robust design. Various VLSI design styles can be used to implement a specific digital design. Each design style bears its own merits and drawbacks. To create a good design at low cost and in a short time, designers have to select proper tools to carry out their designs effectively.

Several design styles can be considered for chip design. Full custom design, one of the most fundamental VLSI design methods, depends solely on designers who have to calculate the geometry, orientation and placement of every individual transistor without the use of any existing library in a CAD tool. Standard cell and Mask-Programmable Gate Arrays (MPGAs) fall into the class of semi-custom style and can be easily used to implement Application-Specific Integrated

Circuits (ASICs). In standard-cell design [Kang03] all of the commonly used functional modules are developed, optimized and stored in a typical library with fixed height. Mask-Programmable Gate Arrays (MPGAs) [Kang03] are an alternative way for designers to define the interconnections between the transistors which have been fabricated by the manufacturer. Field-Programmable Gate Arrays (FPGAs) [Brow92] provide a means for fast prototyping and also for a cost-effective chip design, especially for low-volume applications [Kang03]. Unlike MPGAs, all programmable interconnects in FPGAs are pre-fabricated such that users can implement the physical interconnects by programming RAM cells or melting fuses.

An FPGA has become a popular means to realize digital systems because of its dramatic reduction of turn-around time and start-up cost compared with traditional ASICs. FPGA placement and routing are two critical phases in FPGA design. FPGA placement determines the location of logic blocks required by circuits in the chip such that the area and speed are optimized. The quality of placement greatly affects the routing phase. Once placement is completed, routing is performed by assigning the actual interconnections between logic blocks. Due to the fact that both placement and routing are NP-hard [Shah91], both phases of design can consume most of the CPU time during compilation. Current CAD tools provide high-quality placement and routing solutions at the expense of CPU time [Mulp01]. The compile time tends to increase tremendously as the size of circuits becomes larger. With the continuous increase in the logic capacity of FGPAs, it is imperative to develop effective and efficient placement and routing algorithms that will provide acceptable solutions in reasonable amounts of CPU time.

1.1 Problem Definition

In order to realize a digital system on an FPGA, an effective and special CAD tool is required. These tools attempt to achieve a successful FPGA design by using heuristics to tackle problems in the form of packing, placement and routing. The focus of this thesis is on developing effective placement techniques that can manage the growing complexity of designs carried out by engineers in industry.

FPGAs have become a popular way to realize digital systems because of their dramatic reduction in turn-around time and NRE cost compared with traditional Application-Specific Integrated Circuits (ASICs) [Kang03]. The logic capacity of FPGAs has increased so rapidly in the last decade (up to 10-million gates), that it has imposed a new challenge on FPGA compile time dominated by placement and routing operations. Although the current CAD algorithms offer high-quality solutions, they require a considerable amount of CPU time. Actually, the compile time will increase exponentially as the complexity of the circuit increases. In addition, there is a necessity today for new fast heuristics that can face the challenge of dynamic placement for reconfigurable computing systems.

1.2 Motivation

The prohibitively long compile times may adversely affect instant manufacturability of FPGAs and become intolerable to users seeking very high speed compile. Today FPGAs, as one of the most popular VLSI design styles, could lose their time-to-market advantage and capacity, since users don't have patience to wait for long periods of time to compile their design. Therefore users are willing to accept

the decrease in quality of final results with less FPGA compiling time. This demand motivates the necessity to explore new meta-heuristics for fast compilation of FPGA design with acceptable quality. Meta-heuristics are a good choice to obtain a good solution to many combinatorial optimization problems, e.g. FPGA placement known as NP-hard [Trim94]. In addition to single-solution search algorithms such as Local Search and Tabu Search, population-based meta-heuristics – Genetic Algorithms or multi-start meta-heuristics – GRASP can be used to deal with FPGA placement problem. In this thesis we investigate the performance of these meta-heuristics on FPGA placement, and attempt to obtain acceptable solutions in a short period of time.

1.3 Proposed Research Approach

Figure 1.1 presents the research approach followed in this thesis to deal with the FPGA placement problem. In most FPGA placement algorithms, initial solutions of placement are constructed randomly. The placement algorithm attempts to improve this random solution by an iterative process which may result in longer execution times. Therefore, a good initial solution is very important to achieve a high-quality solution in a short time. Cluster Seed Search (CSS), a constructive based method, is developed and can be easily implemented in trivial time. In the common VLSI cell placement, constructive placement algorithms are generally based on primitive connectivity [Shah91]. However, in FPGA placement, CSS uses the fanout number criteria to select the best block and create an improved initial and legal placement solution. Greedy Random Adaptive Search Procedure [Feo89]

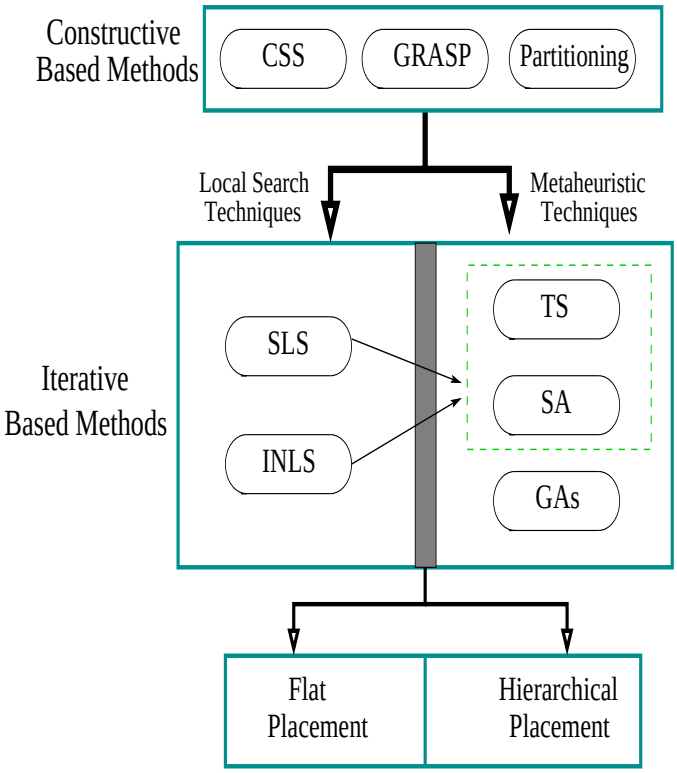


Figure 1.1: Approaches for FPGA placement

(GRASP) is yet another constructive heuristic based approach, which combines the power of a greedy heuristic, randomization and local search procedures. Since it is easy to implement the GRASP heuristic and a few parameters need to be set and tuned, it is appealing to many researchers. Initially, all the logic blocks required by the circuit are placed into one logic block. Next, the best logic block is selected and removed from the initially overlapped blocks and placed into the nearest location to previously set blocks. This procedure is repeated until a feasible initial solution is generated. The Partitioning based approach is another constructive based technique which applies a recursively dividing strategy to reduce the complexity of

FPGA placement, such that improved feasible solutions are obtained quickly.

In addition, two iterative heuristics are proposed to further improve solution quality. The first is an enhancement of local search and is implemented in two different ways. Simple Local Search (SLS) attempts to achieve reduction in wire-length cost by swapping blocks in a window which limits the swapping region. Initially the window is large, and as the heuristic progresses the window shrinks in size. Limiting the scope of swap within the region of the original block position gives superior results compared to unrestricted moves [Lam88]. Local search is also implemented as an Immediate Neighbourhood Local Search (INLS). This technique can achieve suboptimal solutions in a very short period of time by only swapping the adjacent blocks around the selected blocks. Meta-heuristics are also considered in this thesis to further improve solution quality. Tabu Search(TS) guides local search to continue exploration without getting stuck in a local optimum by an absence of improving moves [Arei93]. In order to further explore the solution space, two different search techniques are integrated into the TS search heuristic, i.e SLS and INLS.

Eventually, we propose a population-based meta-heuristic—Genetic Algorithm (GA) for FPGA placement. In GA, each chromosome of the population is encoded to represent a solution to the placement problem. During each generation, the fitness of each chromosome is evaluated, and the best chromosomes of the population are carried over to next generation by an elitism selection mechanism. By using crossover and mutation to explore the problem space, GA can converge toward an improved placement solution.

1.4 Contributions

The main contributions of this thesis can be summarized as following:

- Development and implementation of constructive based techniques(CSS, GRASP, Partitioning based method) as good starting points for other iterative based techniques; Development and implementation of iterative based techniques(SLS, INLS, SA, TS, GA) to efficiently and effectively explore the solution space in a reasonable amount of CPU time.
- A thorough investigation of the performance of several search techniques for FPGA placement on flat and hierarchical levels [Du03].
- A paper was submitted and accepted in CCECE conference proceedings [Bao04].

1.5 Thesis Organization

This thesis is organized as follows: Chapter 2 discusses previous work done on FPGA placement, as well as prior work done on hierarchical clustering. Chapter 3 describes our proposed constructive placement algorithms, and presents experimental results based on the approaches developed. Chapter 4 focuses on iterative placement algorithms and analyzes solution quality obtained from several meta-heuristics (i.e trajectory placement algorithms and population-based placement algorithm). Chapter 5 attempts to present some key conclusions of the work accomplished and proposes possible directions for future work.

Chapter 2

Background and Literature Review

Field Programmable Gate Arrays(FPGAs) are programmable logic devices(first introduced in 1983[Cart86]) that can be used to implement digital systems and are fully reprogrammable. Their appearance is ascribed to a successful combination of the programmability of the configurable logic blocks and the interconnection structure. By eliminating customization during manufacturing, FPGAs eliminate each design's custom mask-making, test pattern generation, wafer fabrication, packaging and testing [Trim94]. Compared to other design styles, FPGAs cut down the design cycle time by providing quick and cheap modifications to the design. Rapid design turnaround results in short time-to-market. With the capability of re-programmability, FPGAs reduce the fabrication cost of implementing a digital design in ASICs.

2.1 FPGA Architecture

In general, a typical FPGA chip is composed of IO blocks, an array of uncommitted configurable logic blocks(CLBs), that can realize a variety of logic functions, and programmable interconnect structure implementing any interconnection topology. Figure 2.1 shows a diagram of a typical FPGA defined in [Trim94].

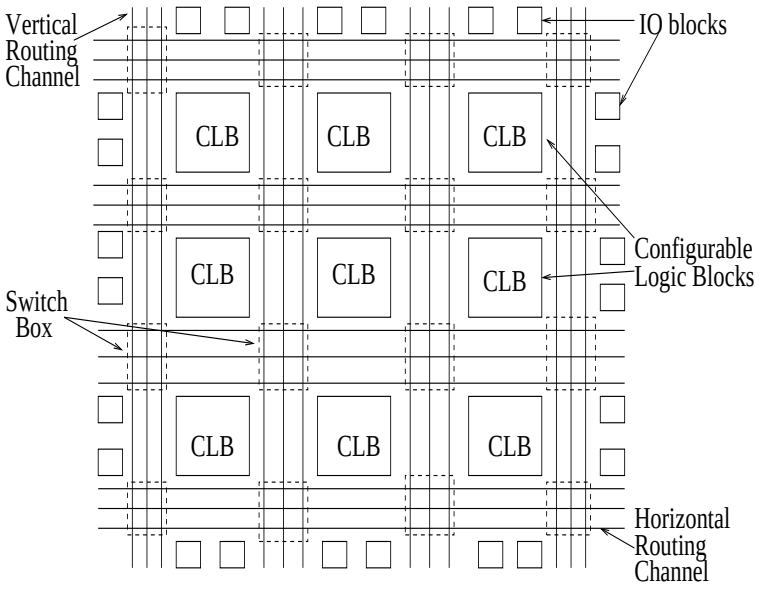


Figure 2.1: 3x3 island-based array FPGA

The functionality of the logic blocks depends on the complexity and architecture of the design. A logic block is a collection of look-up tables (LUTs) and registers [Betz99]. These LUTs and registers are clustered into a logic block capable of implementing either combinational and/or sequential logic functions. The higher the number of LUTs and registers combined into a logic block, the more functions the logic block can fulfill. Based on previous research, the logic block with 4-input look-up table achieves the best performance for most architectures [Brow92].

According to FPGAs routing resource layout, they can fall into one of the following four classifications [Brow92]. The structures of these FPGAs are illustrated in Figure 2.2 .

Symmetrical Island-Based: In this architecture, logic blocks are surrounded by the routing resources which are placed in horizontal and vertical channels. Logic blocks, referred to as Configurable Logic Blocks(CLBs), are connected to routing resources by programmable switches.

Sea-of-Gate: Logic blocks are laid out as a symmetrical array, and routing resources are overlaid on top of the logic blocks. This structure is similar to the architecture used in MPGAs.

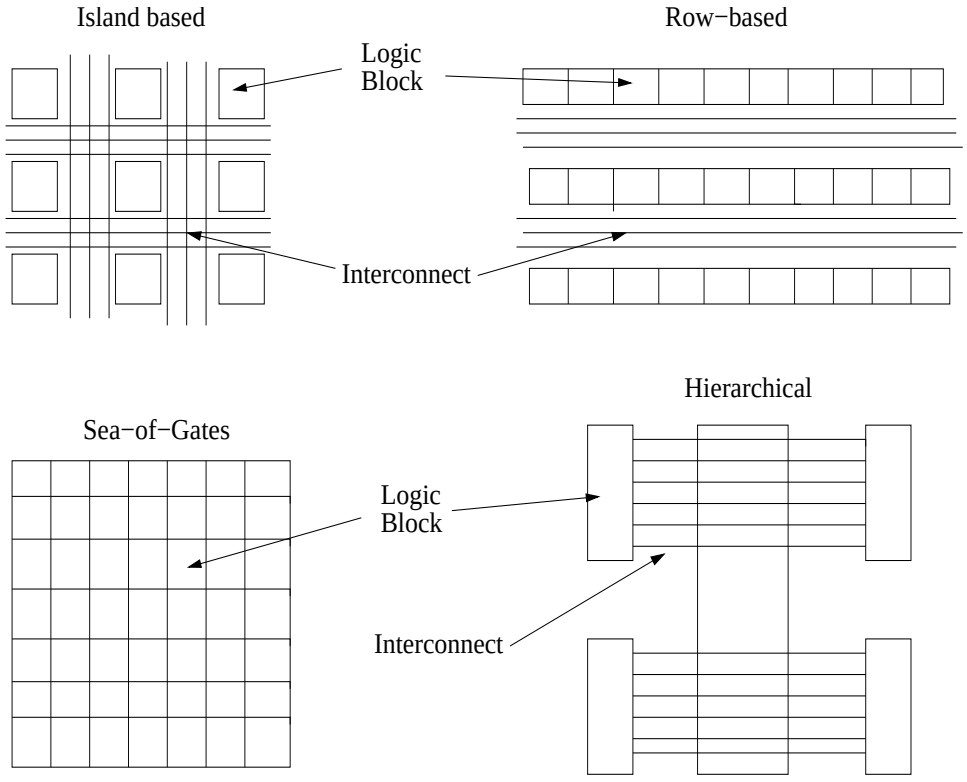


Figure 2.2: The classification of FPGAs

Row Based: In this style of design, the basic architecture consists of rows of logic blocks with horizontal routing resources between the rows. In ASIC design the structure of the standard cell resembles that of Row based FPGA style.

Hierarchical: This architecture is based on an EPROM programming technology. Logic blocks and routing resources are arranged into a hierarchical mode or macrocell mode.

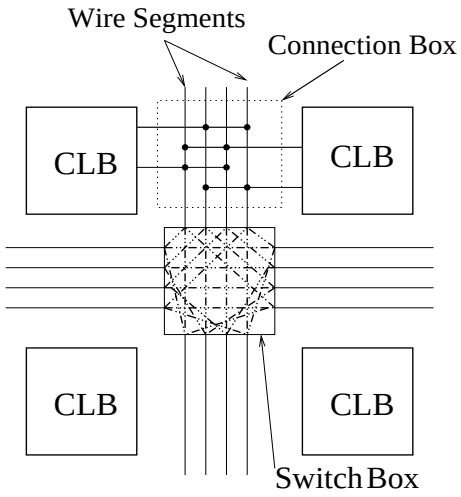


Figure 2.3: The routing architecture of island-based FPGAs

As shown in Figure 2.3, the routing resources include three basic parts: connection block, switch box and wire segments. The input or output pins of CLBs are connected to routing channels by programming the connection block. The switch box lying in the center of four CLBs is a user programmable switch box which can join the wire segments together in the different routing channels. The connectivity F_c of the wire segments in the routing channels depends on the topology [Chan96]. Each wire segment could be connected to some or all of the wiring segments on the neighbourhood channels through the switch box. This connectivity also relates

to the flexibility F_s of the switch box. When $F_s = 0.7W$(W denotes the number of wires in a channel) and $F_c = 3$ or 4, the most area-efficient architecture can be achieved in this type of FPGA [Rose91, Tsen92].

2.2 CAD for FPGAs

The tradeoffs are the essence to the design of FPGAs. A designer who wants to make the best use of FPGAs must balance the cost and performance of the design. The design of FPGAs involves complex steps that cannot be carried out manually. Therefore, an effective and specialized CAD system is required to support the design procedures of FPGAs. A typical CAD system for designing FPGAs is illustrated in Figure 2.4. The detailed description of each step involved is given below.

- Initial Design Entry:

  At the top phase of the CAD system, the description of the digital design to be implemented can be created by using schematic capture, hardware description language such as VHDL or Verilog. The output of this phase is a boolean expression format that is further processed by synthesis and logic optimization.

- Synthesis and Logic Optimization:

  If the description of the circuit is specified in a hardware descriptive language or schematic form, it is necessary to transform the description into logic level representation. The restricted physical structure of FPGAs make it critical to optimize the logic design in order to achieve the speed, area and timing constraints.

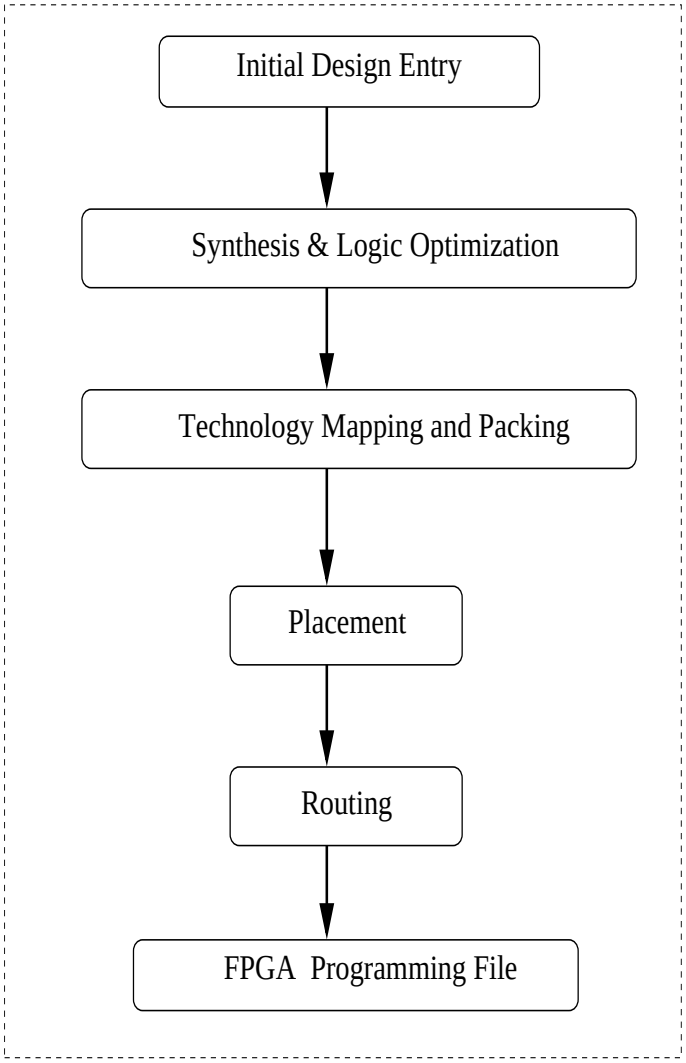


Figure 2.4: The flow of CAD system for FPGA

- Technology Mapping and Packing:

  The optimized logic description of the circuit should then be mapped into look-up tables (LUTs) to implement specified logic functionalities by the technology mapping programs [Fran91b, Fran91a, Chen92, Cong94]. Since a logic block is a cluster of many LUTs and registers, the goal of packing is to minimize the number of logic blocks for the circuit by trying to combine as many LUTs and registers in a logic block as possible. This step attempts to reduce the number of signal connections between CLBs and increase the feasibility of routing [Betz97, Betz99].

- Placement:

  Following the mapping and packing phase, the placement phase is performed to determine the exact physical location of logic blocks on the target device. Obviously, placement algorithms are employed to minimize the total length of interconnect and critical path delay. The quality of placement can affect the final performance of the circuit. Current placement algorithms attempt to achieve high-quality solutions at the expense of high CPU runtime [Betz99].

- Routing:

  Routing is a critical step within the CAD tool to verify the correctness of the digital circuit. In this phase, the routing resources of an FPGA (like wire segments and programmable switches) are assigned to construct the actual connections between logic blocks occupied by the circuit. Routability is an important measure for the success of placement phase.

- FPGA programming file:

  This is the final step of the CAD flow. Following the simulation of the design, a bitstream file, created by the CAD tool, is used to implement the final designed circuit on the target FPGA.

2.2.1 Placement Process in FPGA Design

FPGAs provide a paradigm for cost-effective design and implementation. As the logic capacity of FPGAs increases quickly, efficient CAD tools for FPGAs become essential. FPGA placement is a very important step in CAD flow of a digital system and its quality has a great influence on the routing process which decides if the digital system can successfully be mapped onto the FPGA chip. As a result, FPGA placement problem has been widely studied [Betz99].

In the physical design automation, the placement phase decides the physical location of each logic block, preceding the routing phase. The input to the placement stage is generally a set of a technology-mapped netlist of logic blocks, input and output pads and their interconnections. The result of placement is an assignment of the blocks and pads to specific physical locations of the FPGA that minimizes a specific cost function [Brow92]. As a fundamental element, a logic block is used to perform a specified logic function. Figure 2.5 shows the structure of CLB and the representation in the netlist. IO blocks are the physical interface connecting the implemented system to the outside world.

Since the FPGA placement problem is known to be NP-hard [Brow92], meta-heuristics have been proposed to solve such a problem. By minimizing the total wirelength based on the specific cost function, CAD tools use these meta-heuristics

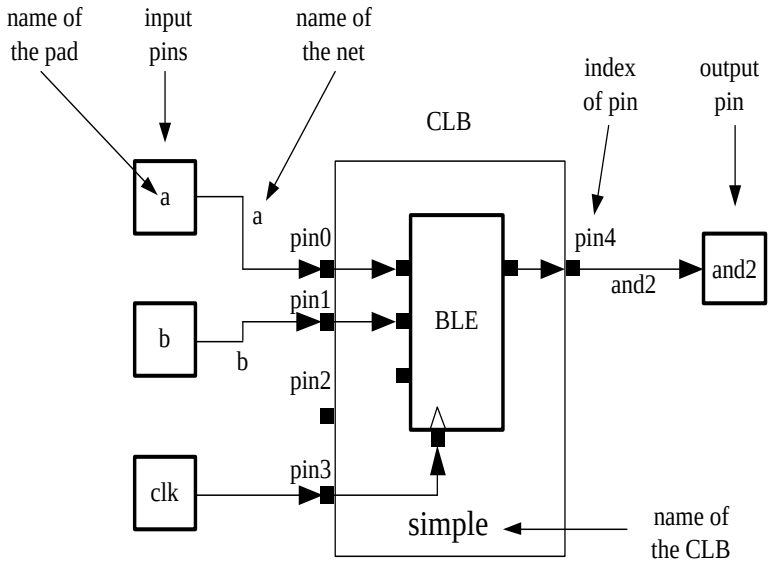


Figure 2.5: A CLB with four pins

to efficiently map blocks and pads in the circuit netlist on the physical layout of FPGA chip. In order to understand the efficiency of CAD tools for FPGAs, in this chapter, the design procedure of FPGA and the details of the placement and routing problems are discussed.

2.2.1.1 Wirelength Estimation for Placement

The total wirelength refers to the length of routing wires required to map the circuit onto an FPGA chip. Area efficiency is one of the issues drawing the most concern in digital system design. Designers attempt to minimize overall interconnection length of the circuit to achieve the minimum chip area. In the placement stage, it is very difficult to determine the exact wire area for the physical interconnection between CLBs and pads. Wirelength estimation is a key factor to solve the placement

problem and can significantly impact the performance of placement algorithms. Various models to estimate wirelength have been proposed, such as Steiner tree [H04], bounding box [Brow92] and spanning tree [Shah91].

Steiner Tree Wirelength Model

A Steiner Tree [H04] is used to construct the shortest route between a set of terminals. Single-Trunk Steiner Tree (STST) is the most appropriate steiner tree estimation technique for gate array or standard cell design styles [Prea88]. In STST, the calculation of the wirelength estimator for a net is illustrated in Figure 2.6. A single trunk can lie at the mean position of all the terminals in a net on a x-y coordinate. Horizontal or vertical wire segments are drawn from each terminal to the horizontal or vertical trunk respectively.

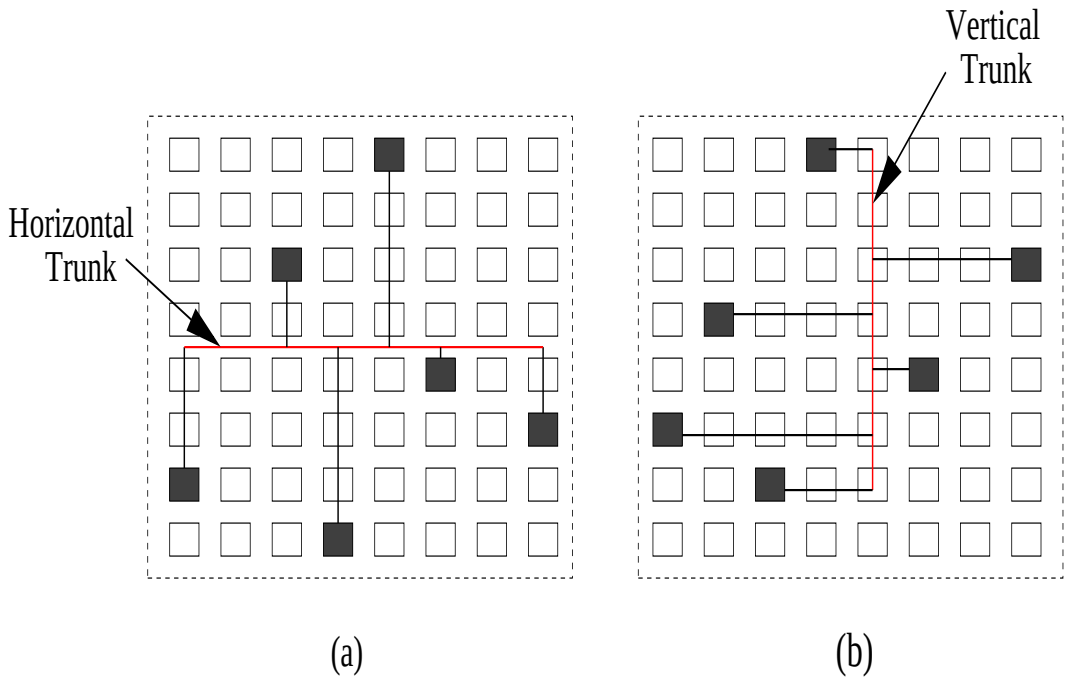


Figure 2.6: Single-trunk Steiner Tree wirelength model

In a horizontal trunk case shown in Figure 2.6(a), the wirelength of net N_i can be calculated as follows. Initially, a single horizontal line is drawn at the mean position ($\overline{x}N_i$) of all terminals' y-coordinate of net N_i. Next, vertical wire segments are drawn from each terminal of Net N_i to the horizontal trunk. The wirelength $L^V_{N_i}$ of the net N_i on x-coordinate is therefore obtained by:

$$L^V_{N_i} = x_{maxN_i} - x_{minN_i} + \sum_{t_j \in N_i} \left(y_{t_j} - \overline{x}N_i \right) \tag{2.1}$$

x_{maxN_i} and x_{minN_i} are defined as following:

$$x^{N_i}_{max} = max\{x_{t_j}, t_j \in N_i\} \tag{2.2}$$

$$x^{N_i}_{min} = min\{x_{t_j}, t_j \in N_i\} \tag{2.3}$$

In order to obtain the accurate wirelength of net N_i, we have to calculate the wirelength of net N_i on the y-coordinate by using a vertical trunk drawn at the mean y position ($\overline{y}N_i$). Figure 2.6(b) shows a vertical trunk with x-coordinate wire segments from the terminals that can be calculated using the following formulas:

$$L^V_{N_i} = y_{maxN_i} - y_{minN_i} + \sum_{t_j \in N_i} \left(x_{t_j} - \overline{y}N_i \right) \tag{2.4}$$

where

$$y^{max}_{N_i} = max\{y_{t_j}, t_j \in N_i\} \tag{2.5}$$

$$y_{N_i}^{min} = min\{y_{t_j}, t_j \in N_i\} \tag{2.6}$$

Combining the above two formulas, the final single-trunk Steiner tree wirelength of net N_i is calculated as:

$$L_{N_i} = \frac{1}{2}\left(L_{N_i}^{V} + L_{N_i}^{H}\right) \tag{2.7}$$

Bounding Box Wirelength Model

Although the STST model provides an accurate estimation of wirelength, the complexity of STST model makes it impractical. The bounding box wirelength model is a widely used approximation technique to estimate the wirelength of a net [Shah91]. In general, it greatly shortens the computation time required by estimation, compared to the STST model. The approximate wirelength of a net is computed by half the perimeter of the smallest bounding box (rectangle) that encloses all terminals of the net, as shown in Figure 2.7.

For a net N_i with n terminals, the maximum (x, y) coordinate and the minimum (x,y) coordinate from the Manhattan net structure are first determined. The wirelength estimation of net N_i is calculated by equation (2.8):

$$L_{N_i} = (x_{max}^{N_i} - x_{min}^{N_i} + 1) + (y_{max}^{N_i} - y_{min}^{N_i} + 1) \tag{2.8}$$

The wirelength determined by this method is equivalent to that calculated by the STST model when nets contain two or three terminals. For nets with more than three terminals, this method usually underestimates the actual wirelength necessary to connect all terminals. Therefore a $q(i)$ factor [Chen93] is required to

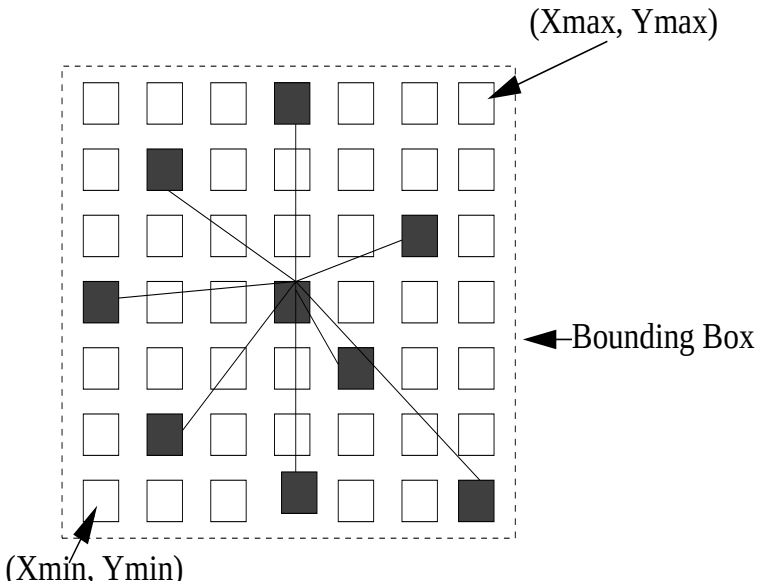


Figure 2.7: Bounding box wirelength model

compensate for the underestimation. The value of q(i) relates to the fanout number of net "i". The factor q(i) is 1 for nets with 3 or fewer terminals and gradually increases to 2.79 for nets with 50 terminals [Betz99]. The calculation of $q(i)$'s value for fanout nets is given by the following equation [Betz00]:

$$q(i) = 2.7933 + 0.02616 \times (TerminalNumber - 50) \tag{2.9}$$

Based on Equations 2.8 and 2.9, a linear bounding box cost function for FPGA placement can be expressed with the following formula:

$$Cost_{bbox} = \sum_{i=1}^{N_{nets}} q(i) \times L_{N_i} \tag{2.10}$$

According to experimental results, this cost function provides good results in

reasonable time [Betz99]. Throughout this thesis, Equation 2.10 will be used to estimate wirelength resulting from solving the placement problem.

2.3 Heuristic Search Techniques

Due to the practical importance of combinatorial optimization problems, many search techniques have been developed to tackle them [Shah91]. However, none of these techniques can solve NP-hard combinatorial optimization problems in polynomial times [Gare79]. Practically, search techniques attempt to generate good solutions in reasonable time at the cost of sacrificing solution quality.

Local Search is a basic search technique that iteratively attempts to achieve improvements by only accepting better solutions in an appropriately defined neighbourhood of the current solution. Instead of exhaustively searching the whole space of possible solutions, Local Search limits the exploration within the local neighbourhood. Accordingly, Local Search is capable to converge to an improved suboptimal solution very quickly. However, it can easily get trapped in a local optimal solution, which is usually far away from the global optima.

Meta-heuristics were developed and introduced [Osma96] to efficiently and effectively explore the search space, by combining basic heuristic methods within higher level frameworks. A meta-heuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space. Learning strategies are used to structure information in order to find efficiently near-optimal solutions [Osma96]. These subordinate heuristics can be either simple local search heuristics

or constructive based procedures.

As high-level strategies, meta-heuristics attempt to converge to high-quality solutions in reasonable time, by combining underlying more problem specific knowledge. Unlike local search techniques, meta-heuristic techniques either allow deteriorating moves, or generate good starting solutions for the local search rather than random initial solutions. These mechanisms enable them to escape from local optima and hopefully reach near-optimal solutions. Due to these advanced features, meta-heuristic techniques are considered excellent candidates to solve the FPGA placement problem.

2.3.1 Genetic Algorithms

Genetic Algorithms were proposed by [Holl75] and have become a common solver for several combinatorial optimization problems. Today, Genetic Algorithms are applied to many scientific and engineering fields such as artificial intelligence [Osma96], strategy planning [Blum03], genetic synthesis [Osma96] and VLSI design [Mazu99], based on the mechanisms of evolution and natural selection. The evolutionary mechanisms of Genetic Algorithms always reserve the best offspring obtained from the parent solutions for a next generation of mating, such that the fittest individuals are encouraged to survive and reproduce through the evolutionary process.

Genetic Algorithms encode parameter sets for the representation of the actual problems, which can be in the form of binary strings. Each parameter set is considered an individual which represents a candidate solution. Rather than working with only a single solution, Genetic Algorithms explore a large population of solutions by using crossover and mutation as the search mechanisms. In addition, selection

and replacement methods are also important genetic operators to create new and improved individuals by making copies of the better individual and removing the worse individual in the previous population. While the cycle of genetic operations is iterated for a number of generations, the overall fitness of individuals in the population tend to improve. Although the transformation of the population between the generations is stochastic, it is a well-structured random search that makes use of all the information obtained during the search and directs the algorithms to converge to an optimal solution. Figure 2.8 illustrates how a population of binary strings evolve in the Simple Genetic Algorithm.

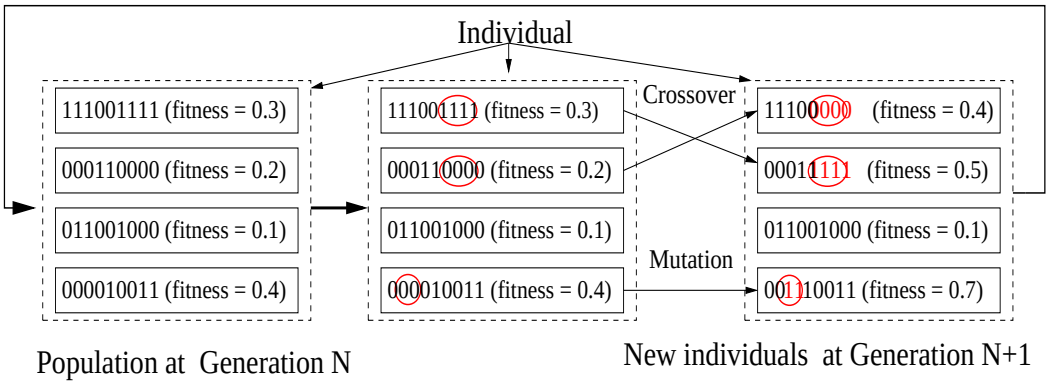


Figure 2.8: Overview of simple Genetic Algorithms

The power of GAs comes from the fact that the technique is robust, and can deal successfully with a wide range of problem areas, including those which are difficult for other methods to solve. However the major drawback of genetic techniques is that they require high computation time. In most cases, many generations or iterations may have to be run on a large population to achieve good results. Another problem of Genetic Algorithms is related to parameter tuning (i.e crossover rate,

mutation rate...). Furthermore, Genetic Algorithms are not guaranteed to obtain the global optimal solutions, although global convergence can be achieved if they assume infinite computation time [Mazu99].

2.3.2 Tabu Search

Tabu search was proposed in its current form by Fred Glover [Glov86] as an important optimization technique. Successful applications in many areas have made Tabu Search one of the standard meta-heuristics used in practice due to its flexibility and efficiency. Undoubtedly, iterative techniques play a key role in solving optimization problems since no technique is able to converge directly to an optimal solution for most optimization problems without consuming many iterations. Tabu search can be considered as a neighbourhood search approach which searches for better solution in the defined neighborhood solution space.

Basic ideas of Tabu search lie in the allowability mechanism that involves historical information to define if a move is forbidden or Tabu and guide the move from the current solution to the next solution in the neighborhood. In order to improve the efficiency of the exploration procedure, the systematic use of memory is an essential feature of the Tabu Search technique. The memory is used to confine the acceptance of the moves such that it is helpful to forbid the moves that may lead to recently visited solutions. Aspiration, an important exploration feature within Tabu Search, is used to temporarily release a solution from the move's tabu status. The role of aspiration is to increase the flexibility of the algorithms while allowing the algorithm to escape local optima, and avoid cyclic behavior. Figure 2.9 shows the components included in Tabu Search technique. Similar to the behaviour of

Simulated Annealing [Kirk83a], Tabu Search accepts non-improving moves which help it escape from a local optimum.

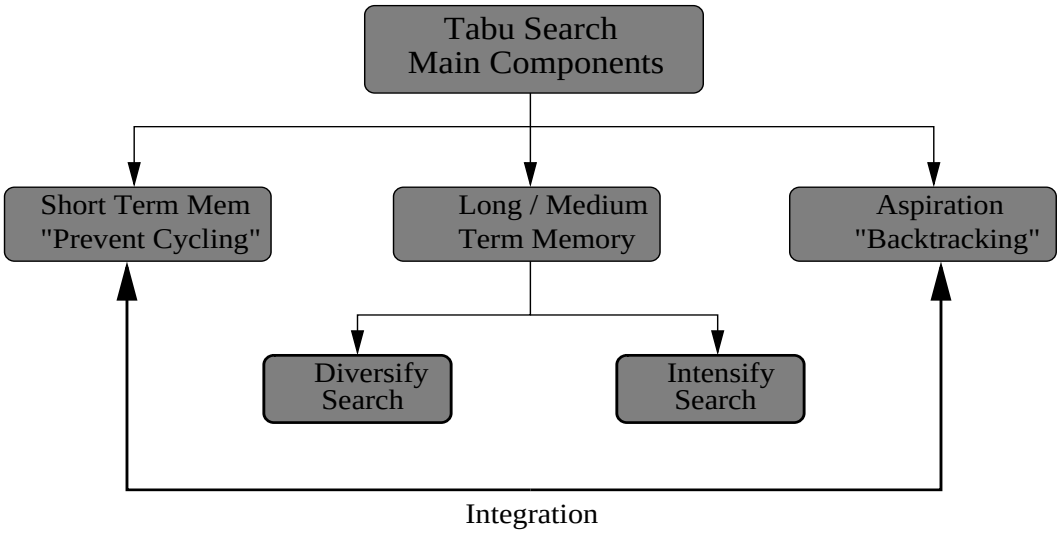


Figure 2.9: Tabu search components

Tabu Search can be used as an iterative technique for FPGA placement by transforming a feasible and initial placement into an improved and final placement. The basic exploration of Tabu Search for FPGA placement is to move from the current solution to the next solution which is the best one in the neighborhood predefined. Even if this move is non-improving, it will still be accepted. If a move is accepted for an arbitrarily defined number of previous iterations, it is deemed not allowable or Tabu, since moving it again may put the routine into cycling. The pseudo code given in Figure 2.10 illustrates the concept of Tabu Search.

```
1.  S = Create an initial solution;
2.  Initialize the tabu memory;
3.  Num_iteration = 1;
    /*set the number of iterations*/)
4.  While(stopping condition is not met)
5.  {   Generate a candidate set in current neighborhood;
          /*Search the whole current neighborhood, put all*/
          /*possible moves into a candidate set*/
6.        S_new = SearchCandidateSet();
          /*find the best solution in candidate set;
7.        if ( f(S_new) < f(S))
8.            S = S_new;
9.        Update tabu memory;
10.       Num_iteration = Num_iteration + 1;
11. } /*end of while*/
12. return S;
```

Figure 2.10: Pseudo-code for basic Tabu Search

2.4 Heuristic Algorithms for FPGA Placement

FPGA placement is a well-known hard combinatorial optimization problem which is NP complete [Shah91]. In the past few decades, several heuristic algorithms have been developed to deal with the FPGA placement. The developed heuristics attempt to achieve good solutions in acceptable time. The fundamental objective of these algorithms is to minimize the wiring area occupied by a digital circuit and speed up the circuit to meet timing requirements. The placement algorithms can be divided into three major classes: constructive technique [Mulp01], min-cut (partitioning-based) technique [Liu98, Maid03], and iterative technique [Nag95, Betz97, Sank99, Tess02, Chan03]. The taxonomy of the heuristic algo-

rithms previously implemented is shown in Figure 2.11.

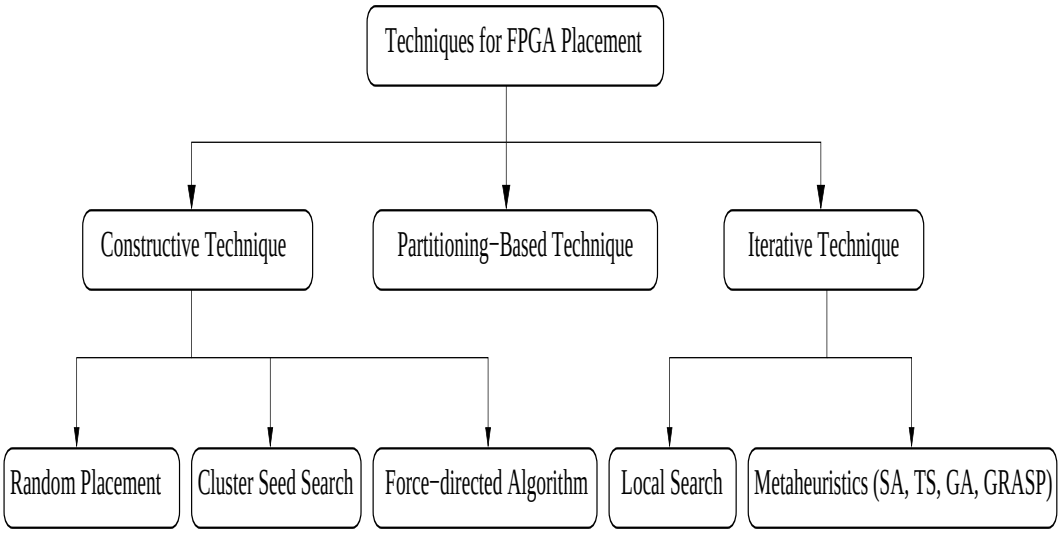


Figure 2.11: Taxonomy for FPGA Placement

Random Placement

Random placement techniques [Mulp01] are the easiest way to place a circuit onto an FPGA chip by randomly scattering the logic block across the chip area. A random placer algorithm can achieve a legal placement in the least amount of time by simply randomly assigning the physical position of the logic blocks on the target chip.

However, since random placement doesn't optimize the location for the appropriate block to achieve the minimum wirelength, the final placement solutions are unacceptable. Therefore, a random placement algorithm is often used to construct a legal initial placement solution for other heuristic algorithms such as Simulated Annealing, Tabu Search and Genetic Algorithms.

Force-Directed Algorithms

A force-directed placement algorithm was proposed for circuit layout in the 1960s [Fisk71]. Force-directed placement algorithms compute the location where the blocks should be placed in order to achieve its ideal placement. Force-directed placement algorithms tend to move blocks in a direction of the total force until the force between the blocks within the same net reaches a certain balance.

At first, a legal initial placement is generated randomly. Based on this initial solution, all the logic blocks are moved to their own appropriate position. Only one block at a time is selected according to the attractive force between any two blocks in the net to which it connects. The magnitude of the force is determined by the distance between the blocks. A block is then selected and moved to its "best location" which is directly related to closest available location to the centroid of all the other logic blocks to which it is connected [Mulp01]. If this "best location" has been occupied by another logic block, the location of these two logic blocks are swapped. Following the swap, the previously selected block is fixed, and the location of this block is forbidden be used by any other block. The swap continues until all the blocks are fixed. All the blocks are then set free and the process is repeated until a stopping criteria is reached.

The main advantage of Force-Directed Algorithms is the fast convergence. With fine tuned parameters and other strategies combined, Force-Directed Algorithm can yield fairly good results [Shah91]. However, it can't always guarantee to find an optimal solution.

Partitioning-Based (Min-cut)Algorithm

As a major class of placement algorithms, Partitioning or Min-cut is based on the Kernighan-Lin [Kern70a] and Fiduccia-Mattheyses algorithms [Fidu82]. Partitioning-

based or min-cut placement algorithms [Liu98, Maid03] use a divide-and-conquer strategy to reduce the problem space by repeatedly partitioning the problem into subproblems. In such a recursive process, while the target FPGA is continuously divided, the algorithm attempts to determine the proper location for the logic blocks occupied by the digital circuit in either part, such that the minimum number of nets cut by the partition can be achieved. The partition procedure terminates when all logic blocks required by the digital circuit are placed onto the target FPGA with least placement cost.

Undoubtedly Min-cut or partitioning-based placement algorithms can run in a comparatively short time compared with iterative improvement algorithms [Huan97]. However these techniques do not directly optimize the placement problem in terms of wirelength. They may achieve a sub-optimal wirelength cost which is an important issue in FPGA placement. Therefore, they can be combined with other heuristic techniques to improve the quality of the final FPGA placement.

Simulated Annealing (SA)

Simulated Annealing Algorithm was proposed in 1983 by Kirkpatrick [Kirk83a]. It mimics the annealing process used to gradually cool molten metal to produce high-quality metal objects [Kirk83b]. It is the most well-developed technique for FPGA placement today. A lot of work [Nag95, Betz97, Sank99, Tess02, Chan03] has been done on Simulated Annealing algorithm for FPGA placement. In this work, VPR (Versatile Placement and Route) [Betz97] is considered to be state-of-art-tool for FPGA placement.

The basic process in Simulated Annealing is to accept all random moves with a controlled probability, which could increase or decrease the cost. A new solution

always generated from the neighborhood of the current solution. The acceptance probability of a move is controlled by a specific parameter "T" that resembles the temperature in metal crystallization. The parameter "T" is used to control the acceptance probability of the cost increasing moves. In most of the implementations of this algorithm, the acceptance probability is given by $e^{\frac{-\Delta C}{T}}$, where ΔC is the cost increase. Initially, "T" is set high so that all moves can be accepted. "T" gradually drops, as the process progresses, so that the acceptance probability for bad moves decreases proportionally. At the final phase, "T" becomes so small that only improving solutions are allowed to be accepted. Although SA is used to obtain very high quality solution, it requires a larger number of iteration to converge to a near optimal solution.

2.4.1 VPR – Versatile Place and Route

VPR (Versatile Place and Route) [Betz97], based on Simulated Annealing, is a well-known package for producing high-quality placement and routing results for FPGAs. In order to achieve the best performance, VPR not only inherits good features from work done by [Huan86, Lam88, Swar90], but also creates its own temperature update scheme and terminating criteria. A novel incremental net bounding box update method is implemented to speed the evaluation process of swapping two blocks. An initial layout is constructed by assigning CLBs and Pads required by the circuit on an FPGA chip. The unoccupied blocks on the FPGA are marked as empty. The pseudo code for Simulated Annealing algorithm is shown in Figure 2.12.

According to [Huan86], the initial temperature is set to 20 times the standard

```
1. P = InitRandomPlacement();
2. T = SetInitTemperature();
3. R_limit = SetInitSearchRange();
   /*R_limit is set to the whole chip initially*/
4. while(ExitCriterion() == false) {
   /*Outloop not done yet*/
5.       while(InnerLoopCriterion()== false) {
         /*innerloop not done yet*/
6.       P_candidate = GenerateMove(P_current, R_limit);
         /*create a new solution from previous one by random pariwise*/
         /*swap with current search range*/
7.       ΔC = Cost(P_candidate) - Cost(P_current);
         /* evaluate the pairwise swap*/
8.       r = GetRandomNumber(0,1);
         obtain a random number between 0 and 1 */
9.       if ( r < e^(-ΔC/T) )
10.            P_current = P_candidate; /*accept solution */
               /* probability for good swap (ΔC < 0) is 1, so the */
               /*improved solution is always accepted. probability for*/
               /* bad swap (ΔC > 0) is e^(-ΔC/T). while */
               /*T is very high, all bad swaps are accepted likely. T */
               /*drops gradually and finally no more bad swaps are accepted*/
11.      }  /*end of inner loop*/
12.      UpdateTemperature(T);
13.      UpdateSearhRang(R_limit);
14.  }   /*end of outer loop*/
15.  return the solution
   /*obtain the final placement P*/
```

Figure 2.12: Pseudo-code for Simulated Annealing

deviation in cost after a set of N_{blocks} swap is evaluated, where N_{blocks} is the total number of CLBs and input and output pads required by the circuit. The following equation is used to calculate the number of new configurations attempted at each temperature:

$$SwapsPer = innerNum \times (N_{blocks})^{\frac{4}{3}} \quad \text{[Swar90]} \tag{2.11}$$

where the default value of the scaling factor *innerNum* is 10. The temperature "T" is set so high that it ensures almost all initial swaps are accepted. The decreasing rate of temperature relates to the frequency of change in cost. The placement quality depends on the number of inner iterations which also affects the execution time as the process continues.

Since the VPR placer targets a wide variety of circuits, the temperature "T" must be updated according to the size of the circuit. The use of the temperature update factor α automatically adjusts the annealing schedule of the VPR placer. The new temperature "T" is obtained by $T_{old} \times \alpha$. Table 2.1 shows the value of the temperature update factor α according to the acceptance rate of swaps at the final temperature.

Acceptance Rate (M)	Temperature Update Factor (α)
$M > 0.96$	0.5
$0.8 < M \leq 0.96$	0.9
$0.15 < M \leq 0.8$	0.95
$M \leq 0.15$	0.8

Table 2.1: VPR temperature update schedule

The current range limiter R_{cur_limit} is controlled by the previous range limiter

R_{pre_limit} and the previous acceptance rate M:

$$R_{cur\_limit} = R_{pre\_limit} \times (1 - 0.44 + M) \tag{2.12}$$

where $R_{pre\_limit} \in [1, max\_dimension]$. All block movements are restricted in this search window whose length is defined by R_{limit}. This search range gradually shrinks, as the annealing process progresses. This range limiting mechanism enables the VPR placer to effectively explore the solution space.

Finally, VPR placer uses Equation 2.13 to terminate the annealing process (the parameter *bbcost* is calculated by Equation 2.10). The final temperature $T_{terminate}$ is related to the average placement cost. If the temperature drops below a certain fraction of the average cost per net, the VPR placer terminates.

$$T_{terminate} < \frac{0.005 \times bbcost}{NumberOfNets} \tag{2.13}$$

VPR employs a fast incremental bounding box update technique to shorten CPU runtime. With these fine tuned parameters, VPR currently outperforms other placement and routing tools in terms of solution quality.

2.4.2 Simultaneous Place and Route Strategy

This simultaneous placement and routing technique has been developed by Nag and Rutenbar [Nag95] to target the Xilinx XC 4000 series FPGAs which contain different wire segments. Unlike sequential place and route algorithms, the basic idea of a simultaneous place and route strategy is to integrate full placement and

full detailed routing to solve the FPGA layout problem. Congestion estimation for conventional place and route approaches is very difficult since each switch block has only finite routing resources. It is possible for the simultaneous place and route approach to obtain extremely dense and routable FPGA layouts by embedding a full maze router into the inner loop of an annealing-based placer.

The placement algorithm is a Simulated Annealing algorithm with refined cost function. The weighted cost function is given by:

$$Cost = W_r \times R + W_t \times T \tag{2.14}$$

Where R refers to the number of nets that lack a complete routing, "T" is the worst-case delay on the slowest path in the current placement. W_r and W_t are the weights of quantities "R" and "T" respectively. In order to achieve the maximal performance optimization, this technique treats the individual LUTs and FF [1] as the placeable cells shown in Figure 2.13. Therefore, the move-set occurring during the placement comprises the swapping of two single LUTs, two groups of LUTs or two CLBs. Since some moves of LUTs or groups of LUTs may cause an infeasible placement solution, the verification of each move's validity is carried out before a disturbance occurs.

An incremental routing phase is performed during the placement process. Every single placement swap results in a small set of affected nets being ripped up. These nets are put into a global queue which is used to record the unrouted nets as the annealer generates. At any intermediate stage, only previously unroutable nets

[1] LUTs and FF refer to look-up tables and flip-flop respectively.

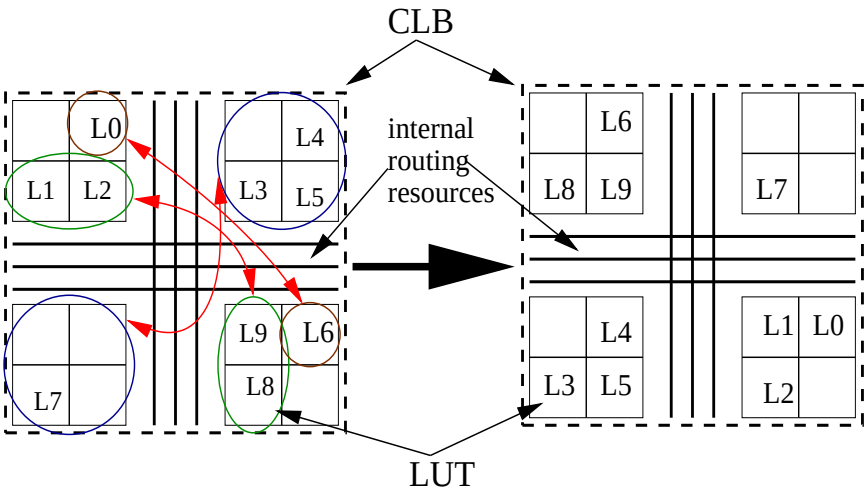


Figure 2.13: LUT swap during placement

and the current nets affected by the placement disturbance are rerouted by a fast, cost-based maze router.

2.4.3 Hierarchical Approaches for FPGA Placement

Hierarchical techniques have been applied successfully on VLSI standard cell placement [Mall89, Sun95, Arei01]. As the capacity of FPGAs tends to grow, FPGA users can't wait a long time for their design to compile, and therefore, efficient placement tools are needed. In order to increase the speedup of the FPGA placement process with high quality, hierarchical approaches have become appealing for the design of large digital circuits on FPGAs [Sank99].

FPGA hierarchical placement includes two steps: (i) proceeding bottom-up clustering (ii) top down improvement. In the first stage, based on module connectivity, hierarchical placement algorithms cluster modules into subclusters gradually as seen in Figure 2.14. Once all the required clustering is done, placement will be performed

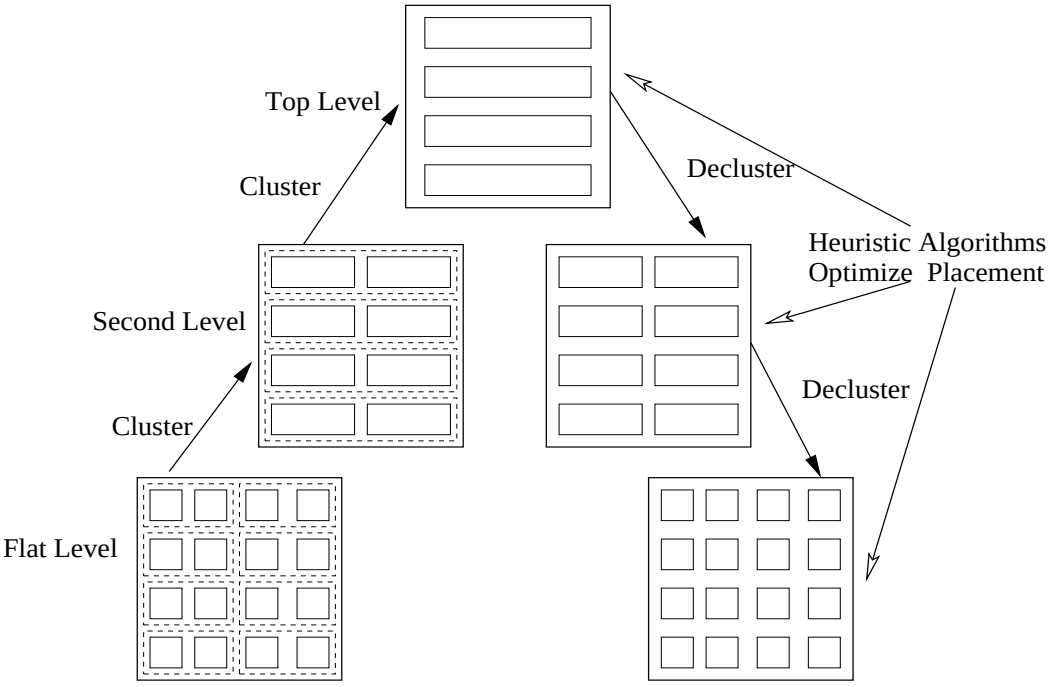


Figure 2.14: Hierarchical Placement

at each level of the hierarchy to yield improvement. The placement heuristics for all the clustered levels (levels other than the bottom level) behave in a top down manner by placing blocks in the general regions where they belong [Du03]. In the second stage, the de-clustering or flattening process is used to decomposes clusters into logic blocks or clusters of lower levels, when the top down optimization at each level is done. A high-quality final solution is obtained at the bottom (flat) clustering level by only moving blocks in small regions. Since the complexity of the placement at the highest level is reduced (with less number of modules), an improved quality of the placement can be obtained in a short time. Figure 2.14 illustrates the clustering/declustering process in hierarchical placement [Du03], which is used in this thesis.

Circuit name	FPGA matrix	Number of CLBs	Number of I/O Pads	Number of Nets	Average Fanout
e64	17x17	274	130	290	3.94
tseng	33x33	1047	174	1099	4.28
ex5p	33x33	1064	71	1072	4.73
alu4	40x40	1522	22	1536	4.52
seq	42x42	1750	76	1791	4.46
frisc	60x60	3556	136	3576	4.48
spla	61x61	3690	62	3706	4.73
ex1010	68x68	4598	20	4608	4.49
s38584.1	81x81	6447	342	6485	4.18
clma	92x92	8383	144	8445	4.61

Table 2.2: MCNC Benchmark circuit suite used as test cases

2.5 MCNC benchmark Circuits

In this thesis, all developed and implemented algorithms target an island-based FPGA model with each CLB including a 4-input lookup table and a D flip-flop. Ten MCNC [Yang91] benchmark circuits, shown in Table 2.2, are divided into three categories: small, medium and large. The number of CLBs in these circuits ranges from a few hundred to almost ten thousand. The scalability of benchmark circuits is described in Section 4.5.1. These benchmark circuits have been broadly used in many FPGA physical design publications [Betz97, Betz99, Sank99, Mulp01, Part01].

2.6 Summary

Several issues related to FPGA placement have been described in this chapter. Basic concepts about the FPGA placement problem and wirelength models that play

a key role on the performance of CAD tools were described. A review of some prior work devoted to FPGA placement was also introduced. The focus was on two well-known FPGA placement and routing packages including *Versatile Place and Route* and *Simultaneous Placement and Routing*. Finally, some insight on hierarchical approaches for FPGA placement was introduced. All heuristic techniques proposed in this thesis will be tested on flat and hierarchical level structures.

Chapter 3

Constructive Based Methods

Most CAD tools for FPGA development place a lot of attention on the placement process to achieve an efficient circuit mapping, which is critical to the routing phase. Placement algorithms can be classified into two main categories: constructive placement and iterative placement. Iterative heuristic algorithms for FPGAs placement can obtain good results by repeatedly improving the placement solution from a random initial starting point. Usually this random initial starting point forces iterative techniques to take a large amount of CPU time to converge to a suboptimal solution. The quality of initial solution therefore has a great effect on the convergence of iterative improvement placement algorithms and may even improve performance for hierarchical flows [Arei93].

Constructive placement techniques are excellent candidates to generate good starting points in negligible amounts of time. These techniques have been successfully applied in ASIC design [Kang83, Yang02]. In constructive placement, a seed cell is chosen and placed within the layout of the chip. Next, a cell is picked up from

a pool of unused cells (according to their connectivity to the previously placed cell) and placed in an empty position near the initial seed cell. This process is repeated until a legal placement solution is obtained. However, the quality of placement solutions produced by constructive techniques is considered poor compared to final solutions produced by iterative techniques.

In this chapter, several constructive heuristic algorithms are developed to create initial legal placement quickly. The first is a simple cluster seed technique explained in Section 3.1. Two more powerful constructive techniques–GRASP and Partitioning-based approach are implemented and described in Section 3.2 and Section 3.3 respectively.

3.1 Cluster Seed Search

Cluster Seed Search (CSS) is a constructive (cluster growth) based approach used to build up an initial and legal placement. Cluster growth placement techniques have been successfully applied to standard cell design [Yang02]. They are considered as bottom-up methods that operate by choosing cells and placing them into a partial placement [Karg86]. There are two main functions used by cluster growth: (i) selection function and (ii) placement function. The selection function is responsible for selecting the best candidate based on a connectivity metric. The placement function decides the best location for the cells according to the availability of vacant space in the area [Karg86].

3.1.1 Implementation

In traditional ASIC standard cell placement [Kang83, Yang02], constructive algorithms are generally based on primitive connectivity rules. Cell arrangement is based on the degree of connectivity to the previously placed cells (most densely connected first) [Shah91]. However, since the number of input and output pins of each logic block is fixed, connectivity rules for traditional ASIC design do not perform as well for FPGA design. Therefore in the FPGA placement, CSS uses the fanout number as a criterion to select the best block and create an improved initial and legal placement solution. A logic block with high fanout indicates that this block belongs to a net which has more terminals. Moving these logic blocks with high fanout together tends to shrink the bounding box of the net containing more terminals with higher probability.

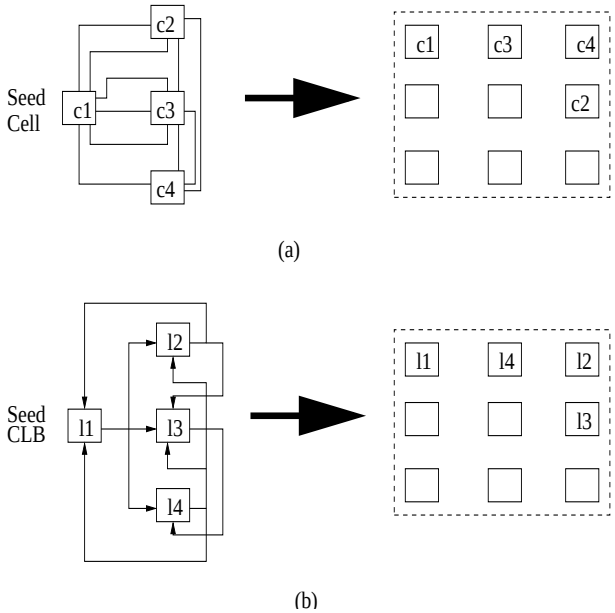


Figure 3.1: CSS in (a) standard cell design and in (b) FPGA design

Figure 3.1 illustrates the difference of CSS in standard cell design and FPGA design. In Figure 3.1 (a), c_1 has higher physical connectivity to c_3 than c_2 and c_4, and c_3 is placed to the location closest to the seed cell c_1. However in Figure 3.1 (b), l_4 has a higher fanout than l_2 and l_3, and l4 is placed to the location closest to the seed CLB l_1.

```
1. Seed = RandomSelectSeed();
   /*randomly pick a block as a seed*/
2. SetLocationOfSeed();
   /*place random seed at the first position of FPGA*/
3. While(Initial Solution Not Complete)
4. {   CreateListOfFanoutNumber(Seed);
       /* create the list of fanout number of blocks connected to the seed*/
5.     Seed = SelectBestBlock();
       /*select the block with the highest fanout number as the next seed*/
6.     SetLocationOf Seed();
   /*place current seed at the location close to the previously placed seed*/
7. } /* end of loop */
8. Return the solution
```

Figure 3.2: Pseudo-code for CSS

The pseudo-code for CSS is shown in Figure 3.2. Typically, a seed block is selected randomly and placed on the FPGA fabric. The next block is then chosen from the remaining unplaced blocks which are connected to the previously placed seed block based on their fanout. The latter is placed at a vacant location closest to the seed block, such that the wirelength is minimized. This current placed block becomes the next seed for next selection. The process is repeated until an improved initial and legal solution is constructed.

3.1.2 Experimental Results

The Cluster Seed Search (CSS) algorithm is run 10 times for all ten MCNC benchmark circuits presented in section 2.5. CSS was compared with randomly generated initial solutions. As shown in Table 3.1, CSS achieves 6% improvement over small-size circuits, 12% improvement over medium-size circuits and 20% improvement over large-size circuits. On average 19% improvement is achieved over all randomly placed circuits in a short period of time. The larger the circuit, the more reduction in terms of wirelength that can be achieved. However, CSS obviously results in poor quality of solutions compared to placements obtained by iterative improvement methods.

Circuit name	**Random cost**	Avg. cost	Max cost	Min cost	**Avg. impr.**	Cost STDEV	Avg. CPU t.	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**7464**	7010	7078	6992	**6%**	38	0.01	0.02	0.005	0
tseng	**40947**	35117	35754	34808	**15%**	211	0.06	0.07	0.05	0
ex5p	**41876**	37532	37802	37381	**11%**	388	0.05	0.06	0.04	0
alu4	**61041**	54028	54902	53175	**13%**	629	0.08	0.09	0.08	0
seq	**78292**	69715	70226	69390	**11%**	340	0.11	0.13	0.11	0
M.avg	**55539**	49060	49671	40349	**11%**	392	0.07	0.08	0.05	0
frisc	**221754**	178101	178902	177393	**20%**	572	0.44	0.45	0.44	0
spla	**233796**	182433	183493	181525	**22%**	703	0.48	0.52	0.47	0.01
ex1010	**329351**	266989	270885	264977	**20%**	2276	0.76	0.83	0.74	0.03
s38584.1	**554873**	480735	482115	478670	**14%**	1415	1.34	1.36	1.33	0.01
clma	**792455**	634721	639724	631368	**20%**	3074	2.23	2.25	2.20	0.02
L.avg	**426446**	348595	351023	346786	**19%**	1608	1.05	1.08	1.03	0.02
Avg	**236185**	194623	196088	193567	**19%**	964	0.55	0.57	0.54	0

Table 3.1: Performance of Cluster Seed Search

3.2 GRASP

Greedy Randomized Adaptive Search Procedure (GRASP) has been successful in solving numerous combinatorial optimization problems [Feo95]. In operations research, GRASP has been applied to solve scheduling problems [Bard89], routing problems [Rese97], partitioning problems [Lagu94], location problems [Rese98] and graph theoretic problems [Lagu98]. GRASP has also been utilized broadly and practically in industrial fields such as transportation [Argu97], telecommunication [Pasi98] and electrical power systems [Bina98].

3.2.1 A Generic GRASP

The GRASP meta-heuristic is a multi-start or iterative process, in which each iteration consists of a construction phase followed by a local search phase [Feo89, Feo95]. Normally, an initial and feasible solution is built up in the constructive phase. A local improving process follows up to explore the neighbourhood of this initial solution and attempts to iteratively improve it. The best solution found in the iteration is stored as the final result.

The pseudo-code of a generic GRASP algorithm is given in Figure 3.3. The algorithm begins with an initial solution construction procedure immediately followed by a local improvement procedure. The GRASP implementation terminates after a number of iterations set by the parameter $Max\_Iteration$. The construction procedure as shown in Figure 3.3 is iterative, greedy and adaptive. The initial solution is constructed by iteratively selecting elements one by one. The choice of the next element to be added to the solution is decided by the order of all elements in

```
MainGRASP(Seed, Max_Iteratrion)
 1.  ReadInputFile();
 2.  for i = 1 to Max_Iteration do {
 3.        S_initial = ConstructionProcedure(Seed);
 4.        S_new = LocalSearchProcedure(S_initial);
 5.        SaveBestSolution(S_new, S_best);
 6.  }; /*end of for*/
 7.  RestoreBestSolution();

ConstructionProcedure(seed)
 1.  S = InitializeSolution();
 2.  EvaluateCostOfSolution(S);
 3.  While ( Initial solution not complete ) {
 4.        GreedyCreateCandidateList(RCL);
 5.        s = RandomSelectionFromRCL();
 6.        S = S ⋃(s);
 7.      ReevaluateCostOfSoluiton(S);
 8.  }; /*end of while*/

LocalSearchProcedure(S_initial)
 1.  ReadInitialSolution(S_initial);
 2.  While( local optimum not obtained ) {
 3.        S_new = LocalImprovement(S);
 4.  }; /*end of while*/
 5.  ReturnSolution();
```

Figure 3.3: Pseudo-code for a generic GRASP

the list of the best candidate elements which is called the restricted candidate list (RCL). Typically, a greedy function is used to guide the addition of each element to the solution. This addition is based on such a criterion that the element to be incorporated into the partial solution is chosen randomly from the RCL. GRASP is adaptive since elements selected at any iteration in the construction phase are a function of those previously selected. The improvement procedure normally is composed of a local search process as shown in Figure 3.3. GRASP can also be used in a hybrid meta-heuristic scheme. Instead of simple local search, Tabu Search or Simulated Annealing may be implemented in GRASP as local search procedures.

3.2.2 GRASP for FPGA placement

As a meta-heuristic technique, GRASP has two main parameters that need tuning. The first is the stopping criteria, and the second is the element selection method in the restricted candidate list. Since only a few parameters need to be tuned, this makes GRASP appealing to researchers.

The pseudo-code in Figure 3.4 illustrates the GRASP algorithm. The parameter *Max_Iteration* is the number of iterations executed and *Seed* is used as the initial point for the construction of the solution in each iteration. In the current implementation, I/O pads are randomly distributed around the FPGA chip and an I/O pad is selected as *Seed* in each iteration.

Initialization and Evaluation of Partial Solution

Unlike a generic GRASP algorithm mentioned in Section 3.2.1, GRASP starts with a partial solution instead of an empty solution in each iteration, as shown in Figure

```
1.  ReadArchtectureFile();
2.  ReadNetlistFile();
    /*read the input files needed for placement*/
3.  for i = 1 to Max_Iteration do {
4.        S = InitializationForConstruction(Seed);
          /*randomly pick up a pad and put all CLBs at the location closest to it*/
5.        EvaluateInitialBBCost(S);
          /*calculate the bounding box cost of this infeasible solution*/
6.        while ( Initial Solution contruction (S) not done ) {
7.              CreateCandidateList(RCL);
                /*greedily create candidate list according to the order of */
                /*bounding box cost resulted in by setting all the unplaced */
                /*CLB at the location closest to previously placed target block*/
8.              TargetBlock = SelectBestBlock(RCL);
                /*select from RCL the best block which will result in the least */
                /*increase in bounding box cost of the current partial solution*/
9.              S = SetBestBlock(TargetBlock);
                /*place selected best block close to the previous target block*/
10.             ReevaluateBBCost(S);
                /*recalculate the bounding box cost of the partial solution*/
11.       }     repeat if a legal initial solution is not created yet
12.       ReadInitialSolution(S);
          /*ready for local search*/
13.       while (local optimum not reached ) {
14.             S = DoLocalImprovement(S);
                /*use local search technique to find the local optima */
16.       }     /*end of local search*/
17.       SaveBestSolution(S);
          /*save the best solution found so far*/
18. }     /*end of for*/
19. RestoreBestSolution(S);
    /*find the best solution*/
```

Figure 3.4: Pseudo-code for GRASP on FPGA placement

3.5. At first, I/O pads are randomly placed around the FPGA chip, and their location are not changed during the construction phase. Next, an I/O pad is chosen randomly as the initial *Seed* for the construction process. Meanwhile, all the CLBs are placed at the closest location to the initial *Seed*. These CLBs are moved, based on the greedy function, until a legal complete placement solution is generated.

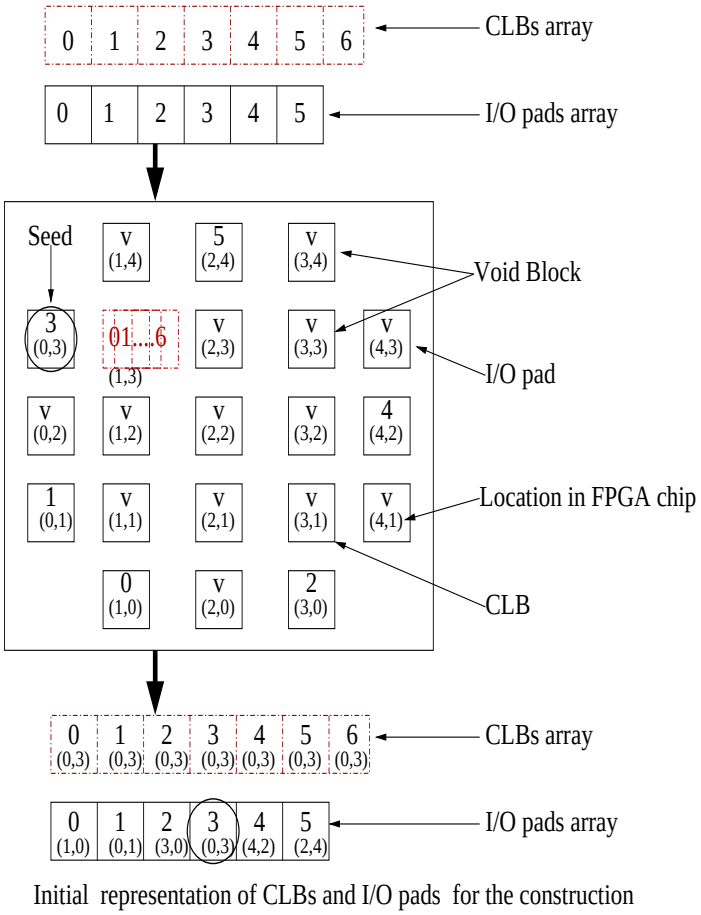


Figure 3.5: Initialization in GRASP construction phase

Construction Phase

At each iteration of the construction phase, a set of candidate CLBs is chosen to be

added to the partial solution. While a CLB is incorporated into the partial solution under construction, the incremental increase in wirelength cost of the new solution is usually represented by the greedy function. The evaluation of all unplaced CLBs by this function leads to the formation of a restricted candidate list (RCL), as shown in Figure 3.6. A CLB that results in the smallest incremental wirelength is selected to be placed to the closest location to the previously placed target CLB. Once the best CLB is added to the partial solution, it is removed from the candidate list. The process is repeated until the construction phase is completed.

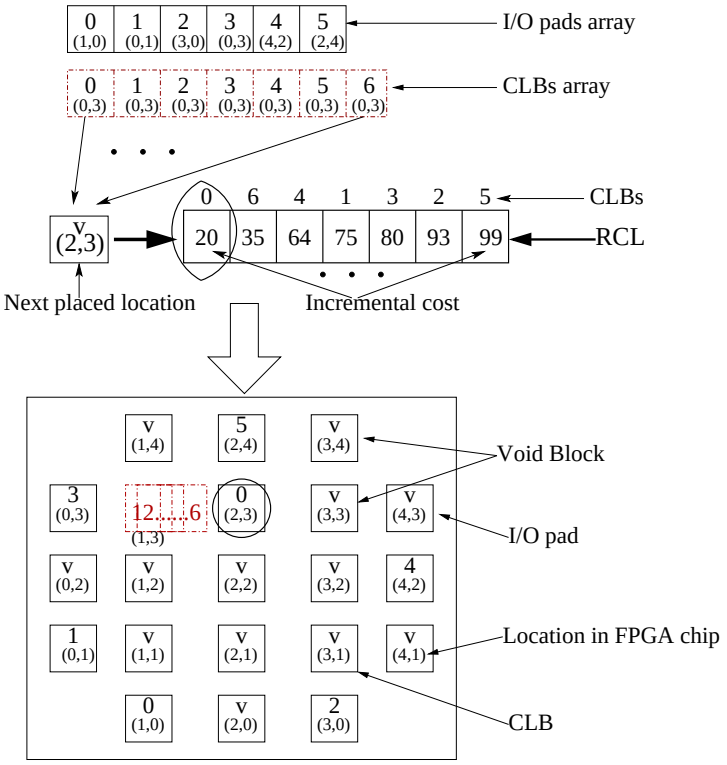


Figure 3.6: Example of RCL construction

Local Improvement Phase

Normally, placement solutions obtained by the construction phase are far from opti-

mum. To further explore the neighbourhoods of these solutions, local improvement techniques usually reach a local optimal solution by iteratively swapping pairs of CLBs resulting in better wirelength cost.

A Simple Local Search (SLS) technique and an Immediate Neighbourhood Local Search (INLS) technique are further implemented as local improvement heuristics for GRASP on the FPGA placement (to be discussed in more detail in Chapter 4). In SLS, the current solution tends to move to a better solution by accepting the first improving swap of CLBs in the neighbourhood of the current solution. On the other hand, in INLS technique, the best-improving strategy is applied where the neighbourhood of current solution is limited to the adjacent area of the target CLB. All immediate neighbours surrounding the target CLB are investigated and the current solution is replaced by the best neighbour. As shown in Figure 3.3, the algorithm terminates after it executes a number of iterations which is determined by the parameter *Max_Iterations*.

3.2.3 Experimental Results

Table 3.2 shows the performance of GRASP based on pure construction phase where no local improvement technique is performed. Two types of local search techniques are embedded into GRASP, the first is Simple Local Search (SLS), and the second is Immediate Neighborhood Local Search (INLS). Both algorithms will be presented in more detail in the next chapter. Tables 3.3 and 3.4 show the performance of GRASP based on SLS and GRASP based INLS respectively, by running GRASP 20 times with $Max\_Iterations = 10$ with different random *Seeds*. In most benchmarks, results obtained indicate that GRASP based on SLS spends more time achieving better

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64	6451	6690	6304	116	0.05	0.05	0.05	0.01
tseng	30579	31847	29945	174	0.7	0.88	0.65	0.25
ex5p	33854	34576	32998	198	0.6	0.8	0.5	1.4
alu4	48711	49271	48005	813	1.4	1.7	1.2	1.7
seq	63513	63976	63082	623	1.8	1.8	1.6	0.6
M.avg	44164	44917.5	43507	452	1.12	1.29	0.99	0.99
frisc	165651	169587	161486	1261	6.7	7.5	5.2	2
spla	161950	163577	160141	2975	7.8	9.4	6.1	3.6
ex1010	245879	249241	241945	3450	10	11.7	8.1	4
s38584.1	415661	428643	400045	8461	22	26	20	4.6
clma	613427	616247	608654	6652	36	44	28	7.5
L.avg	320514	325459	314454	4560	17	20	13	3
Avg	178568	181366	175261	2472	9	10	7	2

Table 3.2: Performance of GRASP based on pure construction phase only

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64	3507	3727	3371	147	0.9	0.9	0.9	0.05
tseng	13646	13896	13501	152	7.5	8	6.5	0.3
ex5p	19535	19920	19174	299	8	9.5	7.5	1.7
alu4	26004	26775	24314	1013	15	17	14	1.7
seq	35199	35860	34082	723	17	18	16	0.7
M.avg	23596	24112	22767	546	12	13	12	1.1
frisc	88254	89594	85426	1641	66	70	64	4
spla	96299	99805	90960	3855	76	120	66	4.3
ex1010	120790	124904	115330	4059	98	106	95	9
s38584.1	147423	160100	123515	9710	255	276	242	26
clma	269330	277197	258250	7092	419	433	366	77
L.avg	144419	150320	134696	5271	182	201	166	31
Avg	81998	85177	76792	2869	96	105	88	16

Table 3.3: Performance of GRASP based on SLS

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64	3797	3952	3685	126	0.4	0.4	0.4	0.01
tseng	16349	17978	15192	165	5.1	6	5	0.08
ex5p	20948	21324	20487	308	5.7	5.8	5.6	0.2
alu4	27213	27529	26845	266	12	13	12	1.1
seq	37229	38805	36100	723	15	18	14	3.3
M.avg	25434	26409	24656	365	9.4	10.7	9	1.3
frisc	96495	99288	93883	1995	60	67	54	26
spla	103765	108000	99488	3532	62	72	58	12
ex1010	135530	141269	130127	4044	88	95	85	9
s38584.1	172593	179760	152512	11402	228	238	219	17
clma	294316	302476	285186	7063	400	410	350	41
L.avg	160539	166158	152229	5607	167	176	153	21
Avg	90823	94038	86345	2962	87	92	80	11

Table 3.4: Performance of GRASP based on INLS

solutions than that based on INLS, by 8% average improvement in wirelength. Figures 3.7 and 3.8 show the comparison of the random based technique, CSS and GRASP used to generate initial solutions. Results obtained indicate that GRASP builds up better initial solutions at the expense of more CPU time.

3.3 Partitioning Based FPGA Placement

Partitioning based methods, also referred to as *min-cut* techniques, have been successfully applied in several areas (i.e VLSI design automation, parallel processing, data mining and efficient storage of large databases on disks). Partitioning Based FPGA Placement presented in this section is based on Peng's Technical Report [Du04]. The algorithm starts by randomly dividing the circuit into two blocks. The partitioning algorithm is then applied to minimize the number of nets cut be-

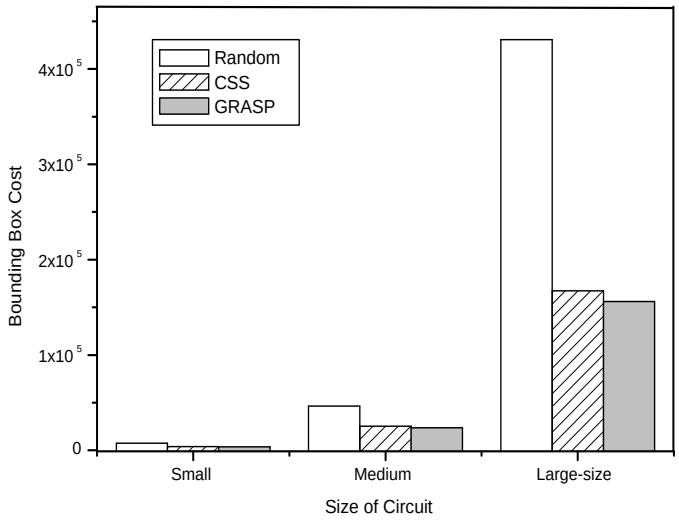


Figure 3.7: Cost of CSS and GRASP construction phase

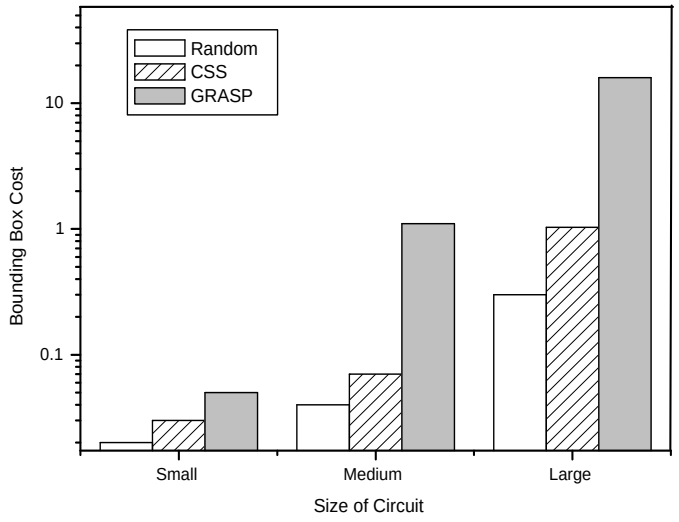


Figure 3.8: Time of CSS and GRASP construction phase

tween the two partitions. This is applied in a recursive manner, as shown in Figure 3.9, until each partition contains a few blocks that are highly-connected.

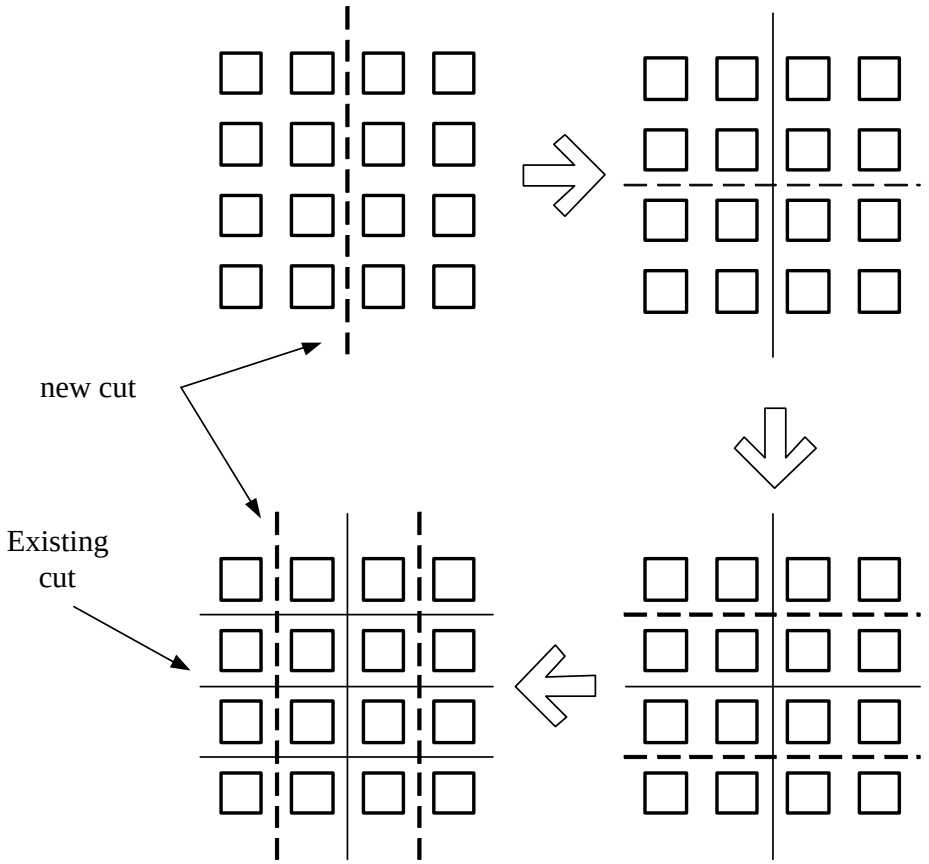


Figure 3.9: Partition based FPGA placement method

Iterative improvement methods used in the past are based on Fiduccia-Mattheyses (FM) algorithm [Fidu82], Kernighan-Lin (KL) algorithm [Kern70b]. Local search can be easily implemented and quickly converge to a local optima. In this thesis an LS and SA are respectively implemented and their performance is investigated and compared in the next section.

3.3.1 Implementation

In the SA based partitioning algorithm implementation, two blocks in **different partitions** are selected and a swap is attempted. If the swap reduces the wirelength cost (cuts), it is accepted; otherwise, the swap is accepted with a probability. The selection of the two blocks is limited to the new partitions, as shown in Figure 3.10.

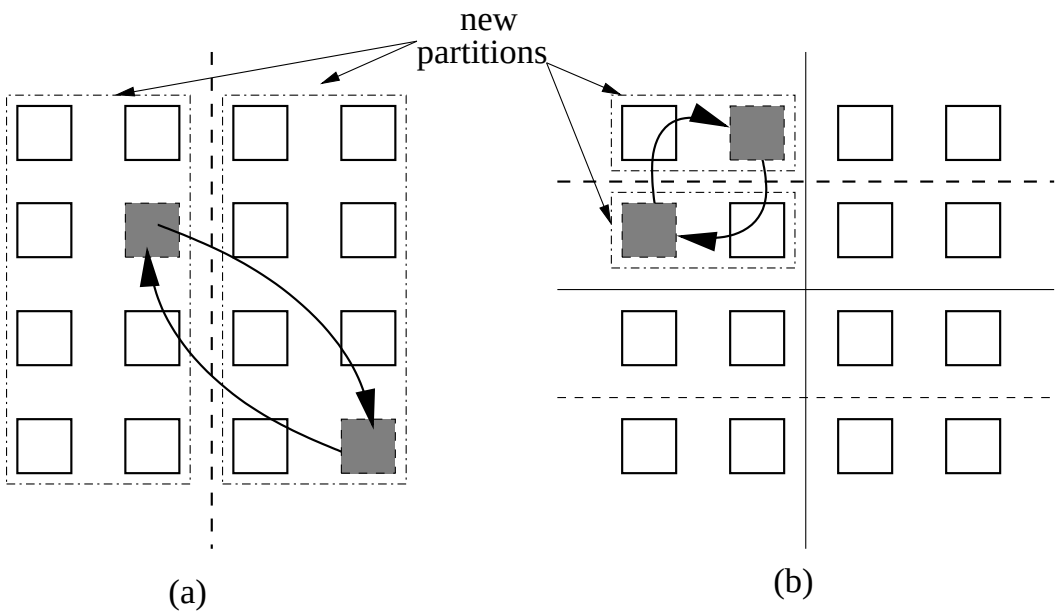


Figure 3.10: Recursive partitioning with SA algorithm

The cost function of SA is based on the cuts between two partitions. The start temperature T_0 is set to

$$T_0 = 10 * Cost_{initial} \tag{3.1}$$

The update of T is calculated by:

$$T_{new} = 0.9 * T_{old} \tag{3.2}$$

Furthermore, the number of swaps evaluated at each temperature is set to

$$MovePerT = (N_{blocks})^{4/3} \tag{3.3}$$

where N_{blocks} is the number of blocks existing in one partition. Finally, the program terminates when no further improvement in cost can be achieved for five temperature updates.

```
1.  read the netfile;
2.  select the partitioning algorithm;
    /*SA or LS*/
3.  while(PartitionSize > 1)
4.  {     perform new partitioning based on previous partitions;
5.        while(allPartitionsAreOptimized == false)
6.           perform SA to minimize the cuts between partitions;
             Or perform LS to minize the cuts between partitions;
7.  }
8.  return final solution;
```

Figure 3.11: Partitioning based FPGA placement algorithm

In order to construct good initial solutions quickly, a local search based partitioning algorithm was also implemented. Similar to SA based partitioning algorithm, local search based partitioning recursively evaluates and swaps blocks in different partitions. To further speed up the algorithm, the blocks are randomly chosen and evaluated for a swap. The algorithm terminates after it performs a

number of iterations. The pseudo code is shown in Figure 3.11.

3.3.2 Experimental Results

Experiments were conducted on ten benchmark circuits based on 10 runs. Table 3.5 shows the performance of the SA partitioning based FPGA placement algorithm. The performance of LS is given in Table 3.6. On average, the solution quality obtained by the SA based partitioning algorithm is better (130% improvement) than that obtained by the LS partitioning scheme. This is achieved at the expense of large CPU time. Figure 3.12 shows a comparison between the random based approach, CSS, GRASP and Partitioning in term of cost and CPU time. It is clear that Partitioning based approach achieves the best solution quality in a reasonable time, compared to the other approaches.

Circuit name	Avg. cos	Max cost	Min cost	Cost STDEV	Avg. CPU t.	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	3478	3529	3204	76	1.4	1.5	1.3	0
tseng	11942	13109	11204	98	5	6	5	0
ex5p	18467	19525	18113	126	5	5	5	0
alu4	23472	24784	22925	236	17	18	16	1
seq	30782	31726	29837	195	19	20	19	1
M.avg	21166	22286	20520	164	11	12	11	0
frisc	68923	70293	67230	536	47	49	44	3
spla	73834	74922	72893	687	52	56	50	5
ex1010	83291	85302	82904	1249	84	90	77	8
s38584.1	103847	108371	100283	2837	179	188	165	23
clma	192843	203894	183283	5023	329	374	308	37
L.avg	104548	108556	101319	2066	138	151	128	16
Avg	61088	63546	59188	1106	74	81	69	8

Table 3.5: Performance of SA partitioning based placement

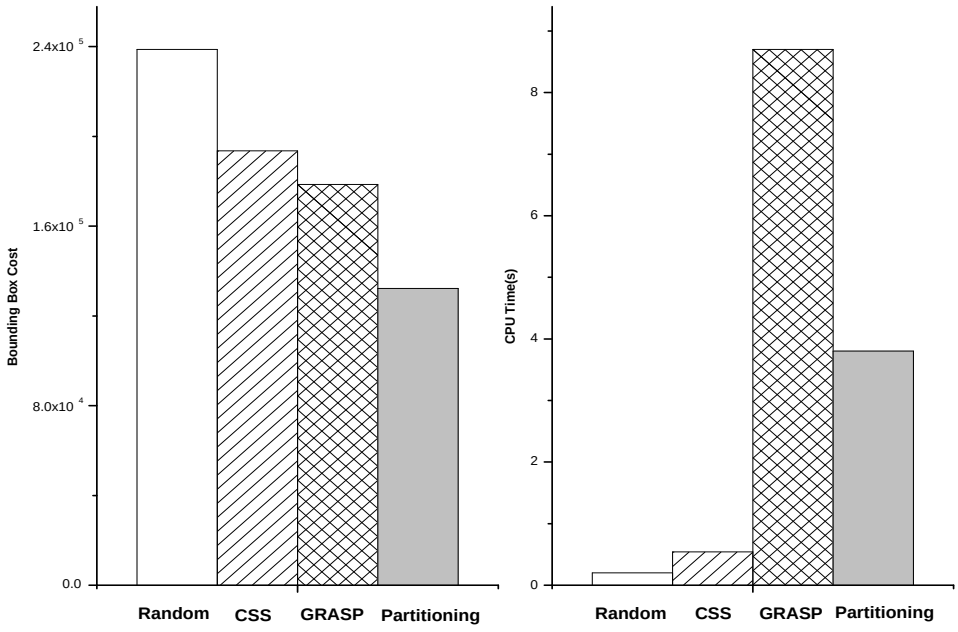


Figure 3.12: Cost/Time of random based technique, CSS, GRASP and Partitioning

Circuit name	Avg. cost	Max cost	Min cost	Cost STDEV	Avg. CPU t.	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	5271	5483	5074	104	0.1	0.1	0.1	0
tseng	21028	23278	20293	194	0.4	0.5	0.4	0
ex5p	24282	25232	22392	274	0.5	0.5	0.5	0
alu4	37922	38283	37291	186	1.8	2.0	1.7	0
seq	52834	53412	52190	382	1.9	2.1	1.8	0
M.avg	34017	35051	33042	259	1.2	1.3	1.1	0
frisc	120129	121932	119272	1009	4.3	4.6	4.0	0
spla	132301	139288	128432	941	4.3	4.5	4.0	0
ex1010	202193	203811	192384	1219	5.8	6.1	5.4	0
s38584.1	292384	299180	286332	3039	7.5	7.7	7.2	0
clma	472992	492382	459302	4283	11.4	11.9	11.0	0
L.avg	244003	251316	237104	1898	6.7	7.0	6.3	0
Avg	136134	140227	132276	1063	3.8	4.0	3.6	0

Table 3.6: Performance of LS partitioning based placement

Partitioning based placement algorithms are good from a "global" perspective, but they do not directly attempt to minimize wirelength. Therefore, solutions obtained are usually sub-optimal in terms of wirelength which can be concluded from Table 3.5 and 3.6 respectively.

3.4 Constructive Techniques: A Comparison

3.4.1 Flat Level Evaluation

The performance of the different constructive based techniques discussed for FPGA placement are compared based on the ten MCNC benchmarks. The comparison is made in terms of solution quality (total wirelength) achieved by each method and the CPU runtime. The Partitioning-based algorithm, CSS and GRASP are run to

create initial solutions. In this case, GRASP was limited to utilize the constructive phase without performing any local improvement. Partitioning-based algorithm is carried out based on simple local search. Table 3.7 shows results obtained by running these algorithms 10 times. Although the results obtained by CSS are inferior, it achieves 19% average improvement over the random based approach in a short time. Partitioning-based algorithm achieves the best results with 44% average improvement compared to random based placement. GRASP yields 25% improvement over randomly generated solutions but suffers from large CPU time overhead.

Circuit name	Avg.random initial cost	**Partition-F**		**GRASP-F**		**CSS-F**	
		Avg. cost	Avg.CPU runtime(s)	Avg. cost	Avg.CPU runtime(s)	Avg cost	Avg.CPU runtime(s)
e64	7542	5074	0.1	6451	0.05	6992	0.01
tseng	41286	20293	0.4	30579	0.7	34808	0.05
ex5p	42301	22392	0.5	33854	0.6	37381	0.04
alu4	61504	37291	1.8	48711	1.4	53175	0.08
seq	79903	52190	1.9	63513	1.8	69390	0.11
M.avg	46489	33042	1.2	44164	1.1	48688	0.07
frisc	229152	119272	4.3	165651	6.7	177393	0.44
spla	236251	128432	4.3	161950	7.8	181525	0.47
ex1010	332664	192384	5.8	245879	10	264977	0.74
s38584.1	559870	286332	7.5	415661	22	478670	1.33
clma	796591	459302	11.4	613427	36	631368	2.20
L.avg	430905	237104	6.7	320513	16	346786	1.03
Avg	238697	132276	3.8	178567	8.7	193567	0.54

Table 3.7: Comparison between Partitioning, CSS and GRASP

Furthermore, hybrid based experiments were conducted by combining the lo-

cal improvement–INLS[1] with these constructive techniques. In this experimental setup, initial solutions obtained by the constructive based techniques are followed by INLS to further enhance the solution quality. Table 3.8 presents the results by running these algorithms 10 times. GRASP based on INLS achieves on average 11% improvement, compared to Random + INLS. CSS combined with INLS still consumes less CPU time and yields average 5% improvement. Partitioning-based algorithm combined with INLS on the other hand achieves on average 12% improvement.

Circuit name	**Random-INLS-F**		**GRASP-INLS-F**		**Partition-INLS-F**		**CSS-INLS-F**	
	Avg. cost	Avg. time(s)	Avg. cost	Avg. time(s)	Avg. cost	Avg. time(s)	Avg cost	Avg. time(s)
e64	4004	0.04	3589	0.05	4001	0.12	3905	0.02
tseng	15803	0.23	13933	0.53	14473	0.44	15729	0.18
ex5p	21352	0.24	19858	0.6	20061	0.51	20445	0.22
alu4	28635	0.35	25160	1.22	26097	1.9	27517	0.27
seq	39096	0.50	35610	1.45	36671	2.1	37174	0.36
M.avg	26221	0.33	23640	0.9	24325	1.3	25216	0.26
frisc	102901	1.28	92581	5.5	93088	4.6	94450	1.04
spla	110372	1.38	98455	5.95	95641	4.9	101296	1.65
ex1010	138479	3.0	108885	8.8	111177	6.4	128115	3.72
s38584.1	204574	3.62	183201	20.1	183297	9.1	194362	3.94
clma	330038	6.07	297464	35.6	301505	13.9	318885	6.14
L.avg	177272	3.07	156117	15.1	156941	7.8	167421	3.31
Avg	99575	1.61	87873	7.9	88602	4.4	94180	1.76

Table 3.8: A comparison between Partition/CSS/GRASP with INLS on flat level

[1]INLS refers to immediate neighbourhood local search which is described in section 4.1.2

3.4.2 Hierarchical performance

To investigate the performance of constructive based techniques on hierarchical designs, the technique proposed by Peng [Du04] is used to perform the clustering/declustering based placement. In the hierarchical experimental setup, the level used is $\mathbf{L = 2}$ and the clustering size at each level is set to $\mathbf{S = 4}$.

Circuit name	Avg.random initial cost	**Partition-h**		**CSS-h**		**GRASP-h**	
		Avg. cost	Avg.CPU runtime(s)	Avg. cost	Avg.CPU runtime(s)	Avg cost	Avg.CPU runtime(s)
e64	7542	4380	0.3	6032	0.01	6132	0.02
tseng	41286	13351	1.1	29059	0.1	26741	0.3
ex5p	42301	18484	1.45	33987	0.1	32948	0.4
alu4	61504	31739	5.2	47133	0.2	44765	0.8
seq	79903	40938	5.9	62598	0.3	61024	0.9
M.avg	46489	26128	3.5	43194	0.17	41368	0.6
frisc	229152	103359	12.5	165348	1.2	152365	3.9
spla	236251	113409	13	176709	1.4	170323	4.1
ex1010	332664	159489	17.4	255865	2.2	244079	6.1
s38584.1	559870	235380	24	419650	4.3	384880	15
clma	796591	409349	36	613284	7.6	593996	22
L.avg	430905	204197	20.1	326171	3.34	309128	10.2
Avg	238697	112988	11.7	180966	1.7	171724	5.4

Table 3.9: A comparison between Partition/CSS/GRASP on hierarchical placement

In this experimental setup, constructive based techniques are applied to the top level of the hierarchy. The final solutions are obtained by simply declustering to the flat level. Table 3.9 shows results obtained based on this hierarchical scheme. Partitioning based algorithm achieves the best results with 53% on average improvement. CSS consumes the shortest time to achieve 24% improvement. GRASP on the other hand achieves 28% improvement. Table 3.10 shows the comparison

Circuit name	**Partition-h**		**CSS-h**		**GRASP-h**	
	Avg.cost Impro.%	Avg.CPU Impro.%	Avg.cost Impro.%	Avg.CPU Impro.%	Avg.cost Impro.%	Avg.CPU Impro.%
e64	+15.1	-200	+13.7	0	+5	+60
tseng	+34.2	-175	+16.5	-100	+12.6	+57
ex5p	+17.4	-190	+9.1	-150	+3	+33.3
alu4	+14.9	-189	+11.4	-150	+8.1	+42.8
seq	+21.5	-211	+9.7	-173	+4	+50
M.avg	+20.9	-191	+11.3	-142	+6.3	+45.4
frisc	+13.4	-190	+6.8	-173	+8	+41.8
spla	+12	-202	+2.7	-198	-5.2	+47.4
ex1010	+17.1	-200	+3.4	-197	+1	+39
s38584.1	+18.8	-220	+12.3	-223	+7	+31.8
clma	+10.8	-215	+3	-245	+3.2	+38.9
L.avg	+14	-200	+6	-224	+3.6	+37
Avg	+14.5	-207	+6.5	-214	+3.8	+37.9

Table 3.10: Comparison between hierarchical and flat constructive techniques

with their flat counterparts. GRASP achieves on average 4% improvement in wirelength cost with 38% improvement in CPU runtime. CSS and partitioning-based algorithm on the other hand consume more CPU time to yield respectively 7% and 14% improvement in wirelength cost.

INLS is also used in a hybrid experimental mode for the hierarchical placement. In this experimental setup, initial solutions are obtained via the constructive based techniques at the top level. INLS is then applied to improve the quality of solution at this level. Gradual declustering is then performed and INLS is utilized to further fine tune the search. Table 3.11 shows the performance of Partition-Based/CSS/GRASP with INLS on the hierarchical placement. Table 3.12 shows the comparison with their flat counterparts. It is clear from Table 3.12 that CSS and partitioning-based algorithm achieve on average 15% and 19% improvement.

Circuit name	Avg.random initial cost	**Par-INLS-M**		**CSS-INLS-M**		**GRASP-INLS-M**	
		Avg. cost	Avg.CPU runtime(s)	Avg. cost	Avg.CPU runtime(s)	Avg cost	Avg.CPU runtime(s)
e64	7542	3380	0.38	3620	0.09	3573	0.1
tseng	41286	12983	1.6	14170	0.5	13782	0.7
ex5p	42301	17830	1.87	19943	0.57	19012	0.9
alu4	61504	23091	6.2	26309	0.8	25530	1.4
seq	79903	32932	7.1	35861	1.05	34606	1.75
M.avg	46489	21709	4.2	24070	0.73	23232	1.2
frisc	229152	83275	13.6	88033	3.15	80147	5.8
spla	236251	88633	17.2	98708	3.7	92198	6.68
ex1010	332664	91096	23.6	104627	5.99	95594	10.5
s38584.1	559870	136205	30.1	147381	9.27	138395	19.6
clma	796591	228753	43.8	266101	16.7	247412	33.2
L.avg	430905	125592	25.6	140970	7.8	130749	15.2
Avg	238697	71818	14.5	80475	4.2	75104	8.1

Table 3.11: Comparison between Partition-Based/CSS/GRASP with INLS

Circuit name	**Partition-INLS-M**		**CSS-INLS-M**		**GRASP-INLS-M**	
	Avg.cost Impro.%	Avg.CPU Impro.%	Avg.cost Impro.%	Avg.CPU Impro.%	Avg.cost Impro.%	Avg.CPU Impro.%
e64	+15.5	-217	+7.3	-350	+0.5	-100
tseng	+10.3	-264	+9.9	-178	+1.1	-33
ex5p	+11.1	-267	+2.5	-159	+4.3	-50
alu4	+11.5	-226	+4.4	-196	-1.4	-15
seq	+9.8	-238	+3.5	-192	+2.8	-21
M.avg	+10.7	-223	+4.6	-181	+1.7	-27.6
frisc	+10.8	-196	+6.8	-202	+13.4	-3.6
spla	+7.3	-251	+2.6	-124	+6.4	-10
ex1010	+18	-268	+18.3	-61	+12.2	-15.9
s38584.1	+25.7	-230	+24.2	-135	+24.5	+2.5
clma	+24.1	-215	+16.6	-172	+16.8	+6.8
L.avg	+19.9	-228	+15.8	-136	+16.2	0
Avg	+18.9	-229	+14.6	-138	+14.5	-2.5

Table 3.12: Comparison between hierarchical and flat constructive techniques with INLS

GRASP based on INLS takes 3% more CPU runtime to obtain a 14% improvement in wirelength cost.

3.5 Summary

In this chapter, a new technique, cluster seed approach for FPGA placement was proposed and implemented. In addition two meta-heuristics: GRASP and Partitioning-based Algorithm were implemented and Compared. Although these algorithms belong to the constructive based techniques, GRASP and Partitioning are effective search technique that guides integrated local search to explore the solution space. Based on fanout criteria, CSS generates a good starting point in a trivial amount of time. In GRASP, a greedy construction method iteratively provides multi-starting solutions for the local search. The construction phase plays a very important role for searching the landscape effectively. Partitioning achieves the best results at the expense of CPU time, compared to CSS and GRASP.

In the next chapter, we focus on iterative based techniques for the FPGA placement. An enhanced local search is developed and implemented in two different ways. Immediate Neighbourhood Local Search limits exploration to adjacent areas of the target block, while Simple Local Search limits exploration using an effective window mechanism. Furthermore a Tabu Search technique and Genetic Algorithms are implemented to enhance solution quality.

Chapter 4

Iterative Based Techniques

Iterative improvement based algorithms usually converge to suboptimal solutions in reasonable amounts of time [Shah91]. Iterative placement approaches start with an initial legal solution which can be generated either randomly or by constructive based techniques. Meta-heuristics are iterative processes that efficiently produce high-quality solutions in reasonable amounts of time by guiding local search to explore and exploit the solution space effectively. An improving neighbouring solution with less cost is always accepted in an iterative heuristic. However for non-improving moves (worse moves), the accepting criteria may vary by different algorithms. Local search techniques in general tend to discard non-improving moves. On the other hand the acceptance of such a non-improving solution in meta-heuristic based techniques is accepted with certain criteria. The improvement evaluation function returns an accepted solution from the neighbourhood of the current solution, which need not be better than the current solution. The probability of the acceptance is usually determined by an additional parameter such

as T (temperature) in Simulated annealing, and H (the history of the search) in Tabu Search. The appropriate randomness may guide a heuristic to converge on a high-quality local optimal. On the other hand, excessive randomness may lead to a large amount of time in examining poor solutions or revisiting solutions checked before.

In this chapter, we investigate several iterative heuristic techniques for FPGA placement. Two local search iterative improvement techniques are implemented to obtain solutions fast for both flat/hierarchical designs. The first is Simple Local Search (SLS) which uses a general iterative improvement strategy. SLS attempts to achieve reduction in wire-length cost by swapping blocks in a window which limits the swapping region. Initially the window is large, and as the heuristic progresses the window shrinks in size. Local search is also implemented as an Immediate Neighbourhood Local Search (INLS). This technique can achieve suboptimal solutions in a very short period of time by swapping adjacent blocks surrounding the selected blocks. Furthermore, several meta-heuristics are implemented to further improve solution quality by effectively exploring the solution space. Tabu Search (TS) attempts to efficiently explore the solution space without getting stuck at a local optimal. A simple Simulated Annealing (SA) algorithm is also implemented and its performance is compared with TS. Finally, we investigate Genetic Algorithms for the FPGA placement. Genetic Algorithms effectively attempt to explore the solution space by coordinately using crossover, mutation, selection and replacement operators on a pool of initial solutions.

4.1 Local Search Techniques

As one of the most basic iterative heuristic methods, local search algorithms can find approximate solutions to large-scale combinatorial optimization problems [Arts03]. The fundamental principle underlying a local search algorithm is that it always moves from the current solution to the next improving solution within the neighborhood in a greedy manner.

Local search algorithms attempt to improve the solution quality either stochastically or deterministically. The stochastic strategy always accepts the first improving solution found during random evaluation. The deterministic strategy on the other hand scans and evaluates the whole neighborhood and accepts the best available solution. Typically, local search techniques terminate either by getting stuck in a local minimum or after a predefined number of iterations have passed.

4.1.1 Simple Local Search

Simple Local Search (SLS) uses a simple iterative improvement strategy that swaps blocks in a window which limits the region for swapping as shown in Figure 4.1. Initially the window is set to some large value, usually spanning the whole FPGA chip to enable the exploration of the solution space. As the algorithm progresses, the window shrinks in size to finely tune the search (i.e. enable fine-tuned search). In the current search window, two blocks are randomly selected and evaluated; the swap of these blocks is accepted if the wirelength cost is reduced. The number of

iterations is defined by the following equation:

$$\mathbf{N}_{iterations} = 10 \times (N_{blocks})^{1.33} \tag{4.1}$$

where N_{blocks} is the total number of CLBs and I/O pads. By starting from an initial legal solution, searching the whole neighborhood consumes considerable time which is prohibitively large for NP-hard problems, while attempting to achieve improvements.

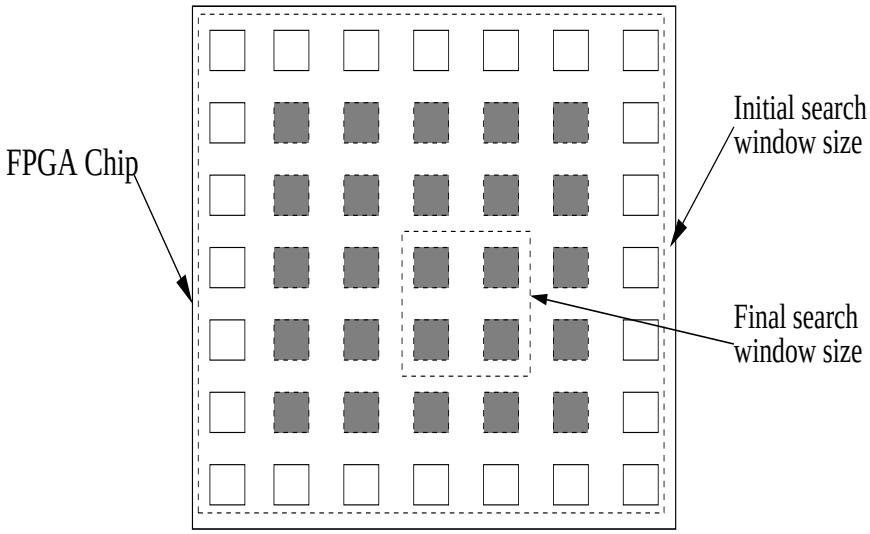


Figure 4.1: The search window of SLS

The pseudo-code for this strategy is shown in Figure 4.2. The complexity of the algorithm is of order $O(n)$ where n is the number of iterations. Since SLS uses a non-deterministic strategy to move from the current solution to a neighbouring solution, it attempts to accept the first improving solution. Although this stochastic strategy makes local search algorithms run fast, it prevents the latter from find-

ing aggressively the best solution in the neighborhood. This may easily trap the heuristic in sub-optimal solutions that are far away from the global optimal.

```
1.  SetExitCriteria();
    /*set iteration number*/
2.  S = InitialPlacement();
    /*create the initial solution*/
3.  window = SetToWholeChip;
4.  set N_iteraion;
5.  while(N_iteraion != 0)   /*start of loop*/
6.  {   Block1 = RandomSelectBlock(window);
        /*randomly pick up the first block*/
7.      Block2 = RandomSelectBlock(window);
        /*random pick up the second block*/
8.      C = Cost(Block1) - Cost(Block2);
       /*calculate the change of cost if swapping these two blocks*/
9.      if(ΔC < 0)   /*only accept the improving swaps*/
10.           S = SwapPosition(Block1,Block2);
11.     N_iteraion = N_iteraion -1;
12.     window = UpdateWindow(N_iteretions);
       /*update the size of the search window*/
13. } /* end of loop */
14. Return the final solution;  /* get final placement solution S */
```

Figure 4.2: Pseudo-code for SLS

Searching within some local neighourhood of the current solution boosts the exploration and exploitation ability of SLS. Figures 4.3 and 4.4 illustrate the effect of window size on the efficiency of SLS. The size of the neighourhood has an impact on possibilities of revisiting previous solutions. When the size of the search window is too large, the efficiency of the search of SLS is mitigated.

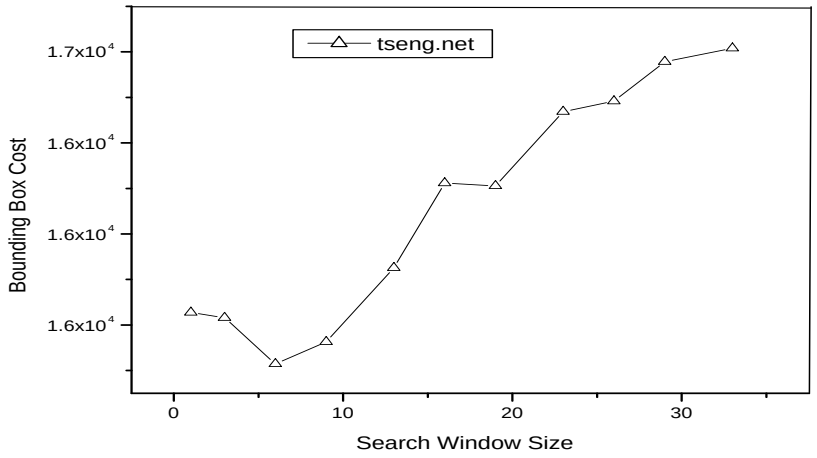


Figure 4.3: Effect of the search window on medium-size circuits

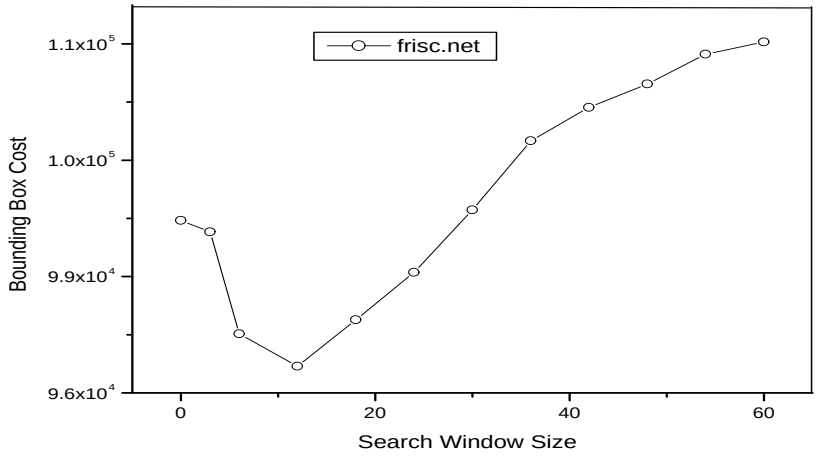


Figure 4.4: Effect of the search window on large-size circuits

4.1.2 Immediate Neighborhood Local search

An alternative local search method is developed to achieve adequate placement solution quality in a short time. Limiting the scope of swaps within the region of the original block position has been shown to give superior results compared to unrestricted moves when a good initial placement exists [Lam88]. Therefore this method doesn't randomly pick up any pair of blocks in the whole neighborhood region but checks the vicinity of target block and swaps the nearby blocks around the target block as shown by Figure 4.5. The next seed block is chosen from the immediate neighbors of the previously selected block, where the selection can be based on either a deterministic or random criteria. As shown in Figures 4.6 and 4.7 respectively, INLS with deterministic seed selection achieves slightly better performance than that achieved by random based selection.

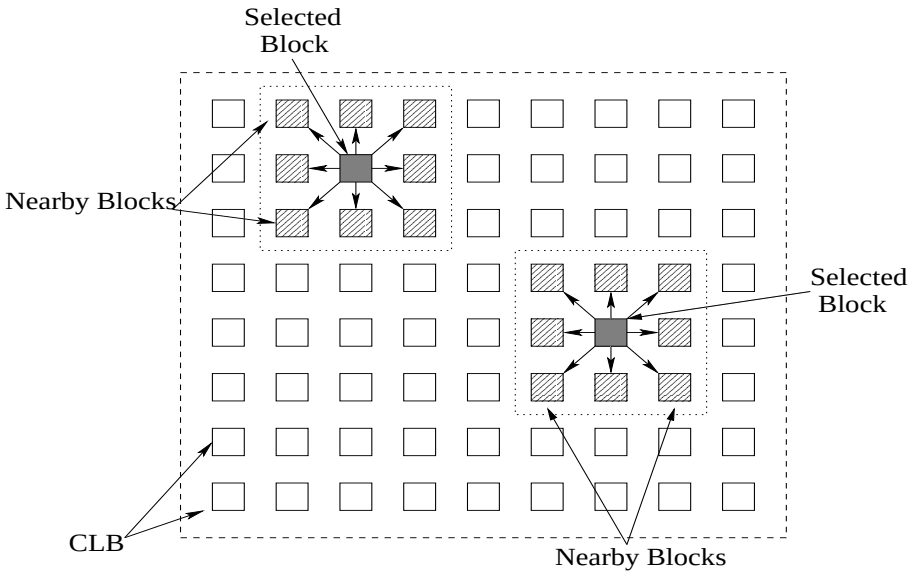


Figure 4.5: Searching Region of INLS

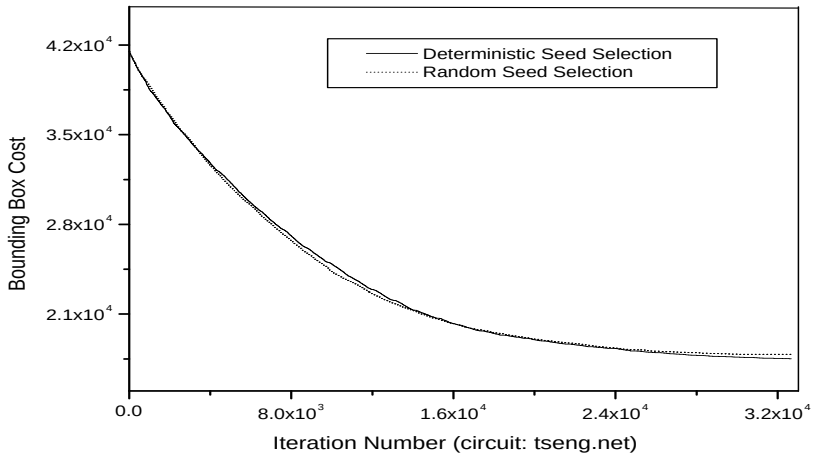


Figure 4.6: Block seed selection criteria on a medium-size circuit

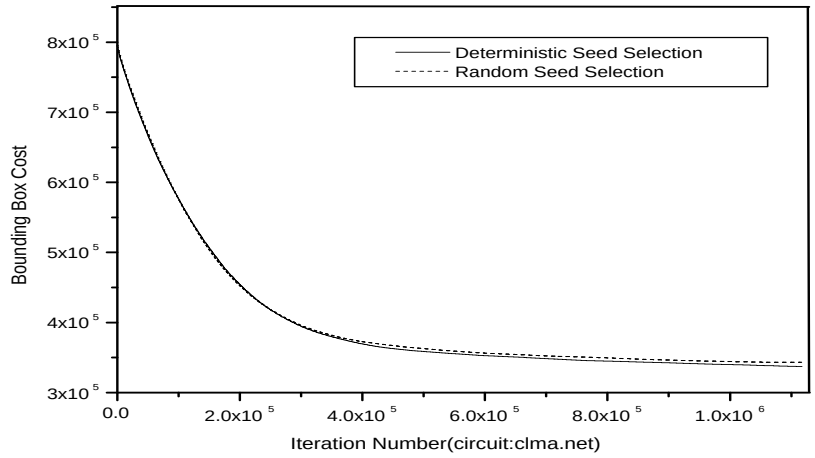


Figure 4.7: Block seed selection criteria on a large-size circuit

```
1.  S = InitialPlacement();
2.  while(ExitCriteria() = false);   /*start of loop*/
3.  {    for( i=0; i < NumberOfTotalBlocks; i++)   /*start of inner loop*/
4.       {    CreateListOfNearby(SeedBlock(i));
              /*create the list of blocks around the selected block*/
5.            for( j=0; j < NumberOfNearbyBlocks; j++)
                /*search the adjacent neighbors*/
6.            {      Candidate = SelectNearbyBlocks();
7.                   ΔC = Cost(Candidate) – Cost(SeedBlock(i));
                     /*calculate the change in the cost*/
8.                   if( ΔC < 0 )   /*only accept the improving swaps*/
9.                   {      S = SwapPosition(SeedBlock(i),Candidate);
10.                         Break;
11.                  } /*avoid doing greedy search*/
12.           }
13.           if( no swap happen)   /*avoid the early convergence*/
14.           {     Candidate = randomSelectBlock();
15.                 ΔC = Cost(Candidate) – Cost(SeedBlock)
16.                 if( ΔC < 0 )
17.                       S = SwapPosition(SeedBlock(i),Candidate);
18.           }
19.      }  /*end of inner loop*/
20. }  /*end of loop*/
21. Return the final solution; /*get the improved initial solution*/
```

Figure 4.8: Pseudo-code for INLS

The pseudo-code for INLS is shown in Figure 4.8 and the complexity is order $O(m)$ where m is the circuit size. In INLS algorithm, the neighborhood is limited to a very small region – adjacent to the target block as shown in Figure 4.5. The algorithm begins its search from any location of the FPGA layout. The first seed block is chosen randomly and the next seed block is selected from the neighbors of the previous seed block. In each iteration all the blocks' adjacent are checked. To further reduce the runtime, INLS swaps the current seed block with one of its nearby blocks which could result in cost improvement. If no improvement can be achieved, an alternative random block is chosen. This methodology is helpful to expand the exploration space efficiently, as illustrated in Figure 4.9. The INLS algorithm uses the greedy strategy to scan and evaluate the placement space and

accepts the best solution until no further improvement is obtained. This greedy method can guarantee the best possible move in the solution space and eventually reaches the local optima.

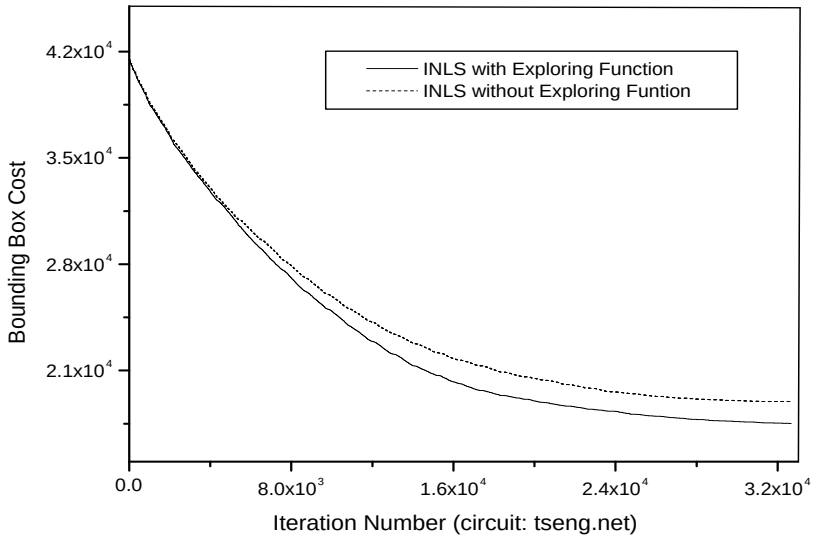


Figure 4.9: Effect of the exploring function

4.1.3 Peformance of SLS and INLS

To evaluate the performance of SLS and INLS, both methods were implemented and their performance was compared. Both methods start with either random initial solutions or improved solutions constructed by CSS. The results in Tables 4.1 and 4.2 are obtained by running SLS and INLS 10 times for the ten MCNC benchmark circuits. The results in Table 4.1 are based on random initial solutions. Table 4.2 is similar except that initial solutions are based on the CSS constructive

method. Tables 4.1 and 4.2 confirm that both SLS and INLS are fast heuristics. INLS achieves better improvement than SLS with less CPU time over medium circuits. However for large circuits, INLS takes less CPU time to obtain the same improvement as SLS.

Circuit name	Init. Cost	**SLS-R**			**INLS-R**		
		Ave. cost	Ave. Impr.	Ave. t(s)	Ave. cost	Ave. Impr.	Ave t(s)
e64	7542	4006	47%	0.06	4004	47%	0.04
tseng	41286	16478	60%	0.32	15803	62%	0.23
ex5p	42301	21670	49%	0.33	21352	50%	0.24
alu4	61504	28797	53%	0.46	28635	53%	0.35
seq	79903	39080	51%	0.62	39096	51%	0.50
M.avg	46489	26506	43%	0.43	26221	44%	0.33
frisc	229152	102676	55%	1.67	102901	55%	1.28
spla	236251	111485	53%	1.74	110372	54%	1.38
ex1010	332664	138229	58%	2.38	138479	58%	3.0
s38584.1	559870	205301	63%	3.97	204574	64%	3.62
clma	796591	332142	58%	6.82	330038	59%	6.07
L.avg	430905	177967	59%	3.31	177272	60%	3.07
Avg	238697	99986	58%	1.88	99575	59%	1.61

Table 4.1: Performance of SLS and INLS

Experimental evaluation was also conducted of a hybrid technique that combines the three methods together(CSS-SLS-INLS). In this experimental setup, initial solutions are obtained via CSS. This is followed by SLS to explore (diversify the search) the solution space. Finally, INLS was used to fine tune the search (intensify the search). Table 4.3 compares CSS-SLS-INLS with solutions obtained by CSS-SLS and CSS-INLS. Results obtained clearly indicate that this hybrid approach achieves on average 10% improvement over the other individual heuristic approaches.

Circuit name	Init sol CSS	CSS-SLS			CSS-INLS		
		Ave. cost	Ave. Impr.	Ave. t(s)	Ave. cost	Ave. Impr.	Ave. t(s)
e64	6992	3904	44%	0.1	3905	44%	0.08
tseng	34808	16036	54%	0.41	15729	55%	0.34
ex5p	37381	21002	44%	0.42	20445	45%	0.35
alu4	53175	27956	47%	0.56	27517	48%	0.47
seq	69390	38062	45%	0.81	37174	45%	0.75
M.avg	40349	25766	36%	0.55	25216	38%	0.47
frisc	177393	94479	47%	2.09	94450	47%	1.81
spla	181525	101300	44%	2.13	101296	45%	1.95
ex1010	264977	128431	52%	2.82	128115	52%	2.72
s38584.1	478670	194515	59%	5.27	194362	60%	4.94
clma	631368	319227	49%	9.01	318885	50%	8.04
L.avg	346786	167594	52%	4.26	167421	53%	3.8
Avg	193657	94493	51%	2.34	94180	52%	2.07

Table 4.2: Comparison of SLS and INLS (CSS initial)

Circuit name	CSS-SLS		CSS-INLS		CSS-SLS-INLS		
	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. Impr.	Ave. t(s)
e64	3904	0.1	3905	0.08	3647	7%	0.11
tseng	16036	0.41	15729	0.34	14059	11%	0.7
ex5p	21002	0.42	20445	0.35	20076	2%	0.75
alu4	27956	0.56	27517	0.47	25927	6%	1.07
seq	38062	0.81	37174	0.75	35997	3%	1.5
M.avg	25766	0.55	25216	0.47	24014	5%	1.01
frisc	94479	2.09	94450	1.81	92098	2%	4.12
spla	101300	2.13	101296	1.95	100592	1%	5.2
ex1010	128431	2.82	128115	2.72	110097	14%	8.01
s38584.1	194515	5.27	194362	4.94	173668	11%	13.9
clma	319227	9.01	318885	8.04	272831	14%	20.21
L.avg	167594	4.26	167421	3.8	149857	10%	10.28
Avg	94493	2.34	94180	2.07	84898	10%	5.56

Table 4.3: Performance of SLS, INLS and Hybrid

4.2 Simulated Annealing

In this section, a simple Simulated Annealing(SA) algorithm is implemented for the FPGA placement. The pseudo code for FPGA placement is given by Figure 4.10. As a common meta-heuristic technique, the SA algorithm attempts to escape from a local optima by accepting moves that deteriorates the object function with a controlled probability. A heuristic function is used to determine the probability of accepting a move, expressed as $f = e^{\frac{-\Delta C}{T}}$ where "T" is a control temperature. Initially, all moves are accepted while "T" is very high. As the search progresses and "T" decreases accordingly, the chance for accepting a bad move mitigates. This acceptance function also implies that a move causing small increase in wirelength is more likely to be accepted than one causing a large increase. To intensify the exploration of the problem space, SA employs a double loop. The outer loop is related to parameter "T", and the inner loop determines the number of neighbourhood moves visited at each temperature.

4.2.1 Annealing Schedule

The Annealing schedule simply includes the initial temperature, the temperature dropping rate, the number of inner iterations and the stopping criteria. Theoretically, the SA algorithm is able to converge with probability 1 to an optimal solution if a certain annealing schedule is utilized [Kirk83a]. In practical implementations, the SA algorithm uses a heuristic based annealing schedule, which no longer guarantees global convergence.

Initial Temperature

```
1.  P_current = InitialPlacement();
2.  Evaluate(P_current);
3.  T = SetInitTemperature();
4.  while( T ≥ T_final) {    /*Outloop not done yet*/
5.        while(InnerLoopExit()== false) {    /*neighbourhood moves*/
6.        P_candidate = GenerateNewSolution(P_current)
          /*create a new solution from previoud one by ranodm pariwise*/
          /*swap within the neighborhood*/
7.        ΔC = Cost(P_candidate) - Cost(P_current);
          /* evaluate the pairwise swap*/
8.        r = GetRandomNumber(0,1);
          /*obtain a random number between 0 and 1 */
9.        if ((ΔC ≤ 0) OR ( e^(-ΔC/T) > r ))
10.             P_current = P_candidate;
                /* accept new solution */
11.       }  /*end of inner loop*/
12.       UpdateTemperature(T);
13.   }   /*end of outer loop*/
14.   return the solution
    /*obtain the final placement P*/
```

Figure 4.10: Simple SA pseudo-code for FPGA placement

The initial value of "T" is determined by the following equation:

$$InitialTemperature = 10 * \frac{InitialNetsCost}{NetsNumber}[\text{Du04}] \tag{4.2}$$

where *InitialNetsCost* is the total wirelength of the initial placement solution and *NetsNumber* is the number of nets in the circuit.

If the acceptance ratio X, defined as the number of accepted transitions divided by the number of proposed iterations, is less than a given value X_0, double the

current value of "T_0". This process is continued until the acceptance ratio exceeds X_0.

Inner Iteration

In our SA implementation, the number of solutions attempted at each temperature is related to the circuit size. Therefore the number of pairwise moves is calculated by the following equation:

$$InnerIteration = innerNum \times (N_{blocks}^{1.33})[\text{Du04}] \tag{4.3}$$

where N_{block} is the total number of CLBs and I/O pads in the circuit. The parameter *innerNum* can be controlled (users defined) to balance the CPU time and placement quality.

Temperature Dropping Rate

Temperature dropping rate α is arbitrarily set to a constant value close to 0.99.

Stopping Criteria

A stopping criterion determines the final value of temperature "T" which is calculated as follows:

$$FinalTemperature < 0.005 * \frac{CurrentNetsCost}{NetsNumber} \tag{4.4}$$

where *CurrentNetsCost* is the total wirelength cost of the current placement solution and *NetsNumber* is the number of the nets in the specific circuit.

4.2.2 Experimental Results

SA is executed on 10 random initial solutions as shown in Table 4.4. SA is also conducted about 10 times with the same parameter set by using good initial solutions constructed by CSS, as shown in Table 4.5. It is clear that SA not only achieves about 5% improvement in wirelength cost but also runs faster by starting from good initial solutions, compared to runs based on random initial solutions. The parameter settings used are summarized as following.

Simple Simulated Annealing Parameter Settings

Initial Temperature	T_{ini}	Temp doubled until $X \leq 0.8$
Final Temperature	T_{fin}	calculated by the equation 4.2.1
InnerIteration	I_{inn}	calculated by the equation 4.2.1
Temperature Decrease	α	0.96

4.3 Tabu Search Technique

Tabu Search (TS) is considered a promising meta-heuristic to solve combinatorial optimization problems [Blum03] with its ability to escape local minima and memory to avoid cycling. In this section, a Tabu Search technique for the FPGA placement is implemented and described. Tabu Search technique is used to guide the local search to avoid getting trapped in local minima. Two local search techniques previously described in section 4.1 are embedded into Tabu Search technique to solve the FPGA placement problem.

The Tabu Search implementation for FPGA placement is illustrated by the

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64-4lut	3051	3100	2973	27	3.9	4	3.9	0.02
tseng	11258	11468	11038	157	19	19.5	19.1	0.14
ex5p	17645	17877	17474	47	19.4	19.6	19.3	0.14
alu4	21500	21616	21391	135	32	32.7	32.2	0.20
seq	27914	28487	27676	336	36.6	37.2	36.2	0.38
M.avg	19579	19862	19395	169	27	27.4	26.8	0.22
frisc	65615	67075	64316	792	97.7	99.6	96.1	0.98
spla	72489	73549	71498	687	104	106	103	0.88
ex1010	78397	78960	77838	593	156	161	154	2.83
s38584.1	96831	97189	96323	451	228	250	216	19.7
clma	182936	185089	179416	1374	362	394	319	38.6
L.avg	99254	100372	97878	779	189	202	178	12.4
Avg	57764	58445	56994	460	105	112	100	62.9

Table 4.4: SA with random initial solutions

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64-4lut	3040	3091	2977	36	3.9	4.1	3.9	0.04
tseng	11165	11358	10793	185	19.2	19.4	18.9	0.16
ex5p	17642	17885	17386	162	19.3	19.5	19.1	0.14
alu4	21480	21696	21295	136	32.1	32.4	31.7	0.23
seq	27796	27975	27558	153	36.7	36.8	36.4	0.15
M.avg	19520	19728	19265	159	26.8	27	26.5	0.13
frisc	65359	66988	64080	835	96.4	97.9	94.4	1.11
spla	71472	72389	70587	615	103	104	102	0.70
ex1010	78282	78813	77720	438	152	158	149	3.86
s38584.1	96732	97104	96243	442	227	251	213	21.1
clma	182134	184230	179001	4435	352	390	304	44.0
L.avg	98795	99904	97526	1353	186	200	172	14.1
Avg	57510	58152	56767	743	104	111	97	7.2

Table 4.5: SA with CSS initial solutions

```
1.  S = InitialSolution(construct_method);
   /*using different methods to create the initial solutions for iterative*/
   /*tabu search technique, i.e cluster seed, random method*/
2.  set local improvement methods;
    set tabu move criteria and tabu_list = ∅;
   /*two local search methods: INLS and SLS */
   /*two tabu move criteria are following:*/
   /*if anyone of a pair of blocks is in current tabu list, these blocks are tabu*/
   /*if both of a pair of blocks are in current tabu list, these blocks are tabu*/
3.  set max_iterations and the size of tabu list ;
    /*set the maximum number of iterations*/
4.  inner_iteration = 0; best_solution = S;
    best_cost = Cost(S); best_moves = 0;
5.  while ( inner_iteration - best_moves < max_iterations ) {
6.        inner_iteration = inner_iteration + 1;
7.        target_block = RandomSelect();  /*randomly pick up a block*/
8.        candidate_list = Neighborhood(target_block);
          /*according to local search method, create the candidate list based*/
          /*on neighborhood of target block */
9.        target_move = selectBestMove(candidate_list);
          /*select a pair of blocks from candidate list leading to the best cost*/
10.       if ( target_move ∉ tabu_list ){
11.            S = update(S, target_move);
               generate a new solution*/
12.            best_move = best_move + 1;
13.            tabu_list = addTabuList(target_move);
               add this pair to Tabu list and remove the oldest pair in this list*/
14.       } else if ( Aspiration(target_move) = True) {
                    /*Aspiration procedure*/
15.                 S = update(S, target_move);
16.                 best_move = best_move + 1;
17.                 tabu_list = addTabuList(target_move);
18.       }
19.       if( cost(S) < best_cost) {
                    /*update the information for stop criterion*/
20.                 best_cost = cost(S);
21.                 best_move = inner_iterations;
22.                 best_solution = S;
23.       }
24.  { /*end of while*/
25. return best_solution;  /*get the final and improved solution*/
```

Figure 4.11: Pseudo-code of Tabu Search for FPGA placement

pseudo code in Figure 4.11. The algorithm explores the problem space by moving from the current solution to the next best solution in the current neighborhood. Initially, a target block is randomly chosen in the search window defined by the local improvement method. All candidate blocks within the neighborhood of this target block are exhaustively evaluated. Based on computed estimated gains, the best block associated with the fittest gain is swapped with the target block if they are not in the tabu list. The new solution is produced by the interchange of the target block and the best block found in the candidate set, even if the move deteriorates the solution. Only one tabu list, whose size T_{size} is related to the size of the circuit, is used to keep track of the last pair interchanges and determine whether a move is tabu or not. The tabu list is introduced as a circular list in order to prevent cycling. The addition of new move removes the oldest move kept in the tabu list. To increase the flexibility of the algorithm, while preserving the basic features that allow the algorithm to escape local optimal, aspiration is used to temporarily release a solution from its tabu status.

4.3.1 Neighborhood Move

The neighborhood of local search is defined as the set of all solutions that can be obtained by swapping the target block with all blocks in the search window. The search window size depends on local search technique implementation. Immediate Neighborhood Local Search (INLS) always constrains the move within the vicinity of the target block. In Simple Local Search (SLS), the search window shrinks as the search progresses. In the current implementation, a greedy method is used to search the neighborhood in both local improvement approaches. Only the best

move found in the search window is evaluated to decide if the move is accepted by the Tabu Search algorithm.

4.3.2 Tabu Criteria

To avoid the non-productive moves and improve the search efficiency, tabu restriction (or penalty) is used to discourage the reverse of selected moves. In the current FPGA placement implementation, two tabu criteria are introduced to determine if a move is *tabu*. The first criteria "TC1" treats a move to be tabu if either block is on tabu list. The second criteria "TC2" prevents a move to take place if both blocks are in the tabu list. Figures 4.12 and 4.13 show the effect of different tabu criteria on medium-size and large-size circuits. Since TC2 reduces the search space by setting the stringent Tabu condition than TC1, TC2 forces the algorithm to converge earlier.

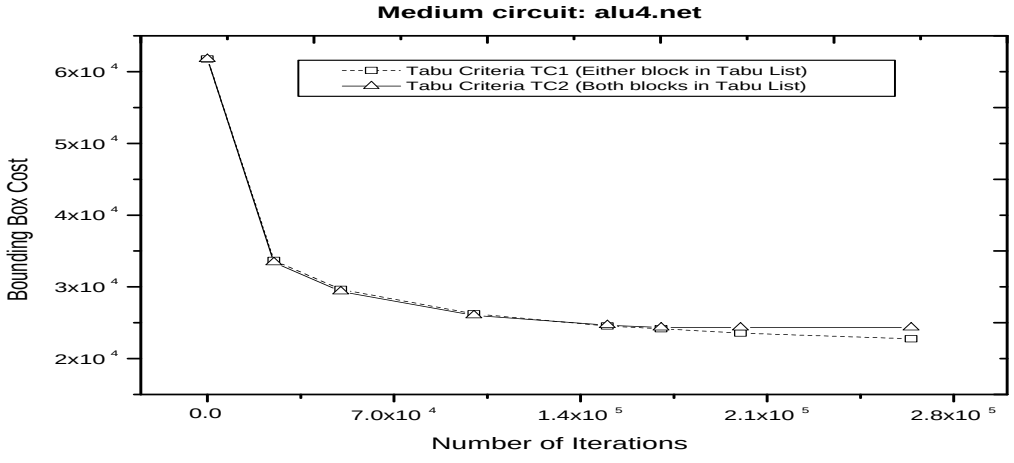


Figure 4.12: Effect of Tabu criteria on a medium-size circuit

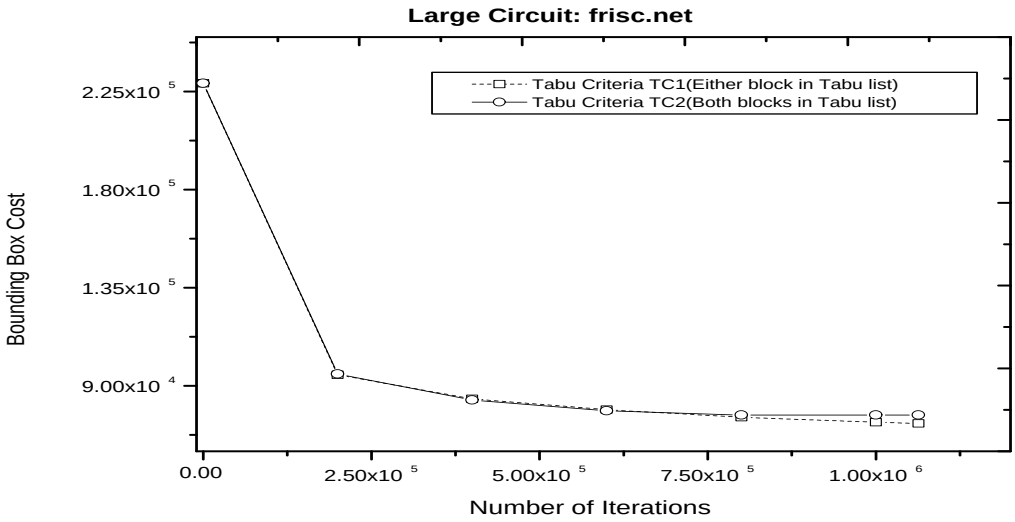


Figure 4.13: Effect of Tabu criteria on a large-size circuit

4.3.3 Tabu List Size

The tabu list size plays an important role in preventing cycling since it contains historical information of previous iterations. The size of tabu list has a great effect on the search efficiency. If the length of the list is too small, this might not prevent cycling. On the other hand, a long size creates too many restrictions such that it degrades the efficiency of the search. In practice, the size of the tabu list for the FPGA placement often grows with the circuit size. However, since the FPGA placement is an NP-hard optimization problem, it is difficult to find a proper value that prevents cycling and at the same time not limit the exploring capability of Tabu Search(TS). The experimental observations show that the best tabu list size can be determined by the following equation:

$$T_{size} = \frac{N_{CLBs} + N_{PADs}}{L_{matrix} \times 4} \tag{4.5}$$

Where N_{CLBs} is the total number of CLBs, N_{PADs} is the total number of pads, and L_{matrix} is the length of FPGA chip matrix. All the parameters in the equation are empirically based on experimentations. Figures 4.14 and 4.15 show the effect of tabu list size on different circuits.

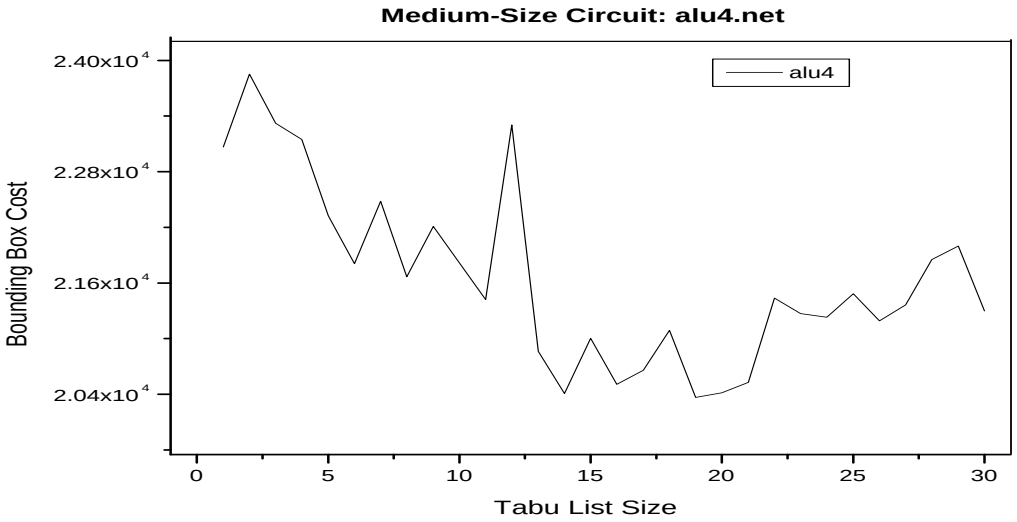


Figure 4.14: Effect of tabu list size on a medium-size circuit

4.3.4 Aspiration Criteria

The tabu restriction imposed by Tabu Search (TS) might be too stringent to allow any exploration of the solution space. The aspiration criteria plays a crucial role in allowing the algorithm to search promising regions within the problem space

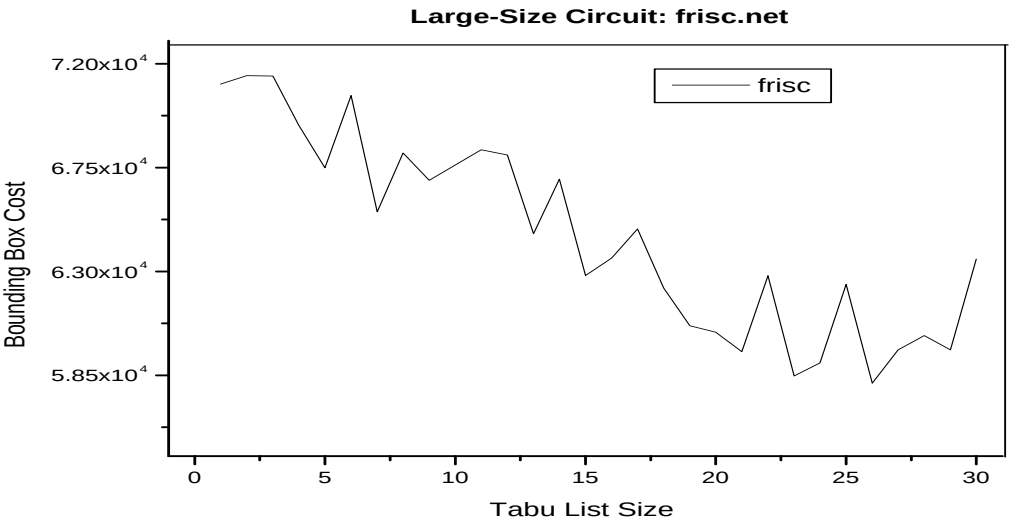


Figure 4.15: Effect of tabu list size on a large-size circuit

that might lead to better solution quality. Usually, the tabu status of a move may be dropped when the cost associated with a tabu move is less than the value of aspiration function associated with the cost of the current move. However, aspiration criteria used in this dissertation is based on the following criteria: a tabu move status is overridden when its cost is better than the best cost found so far during the search (note that a tabu move is not removed from tabu list even if it is accepted during the aspiration phase). Figures 4.16 and 4.17 show the effect of Tabu Search with/without the aspiration function. The aspiration mechanism relaxes the tabu condition and improves the exploration capability of Tabu Search technique.

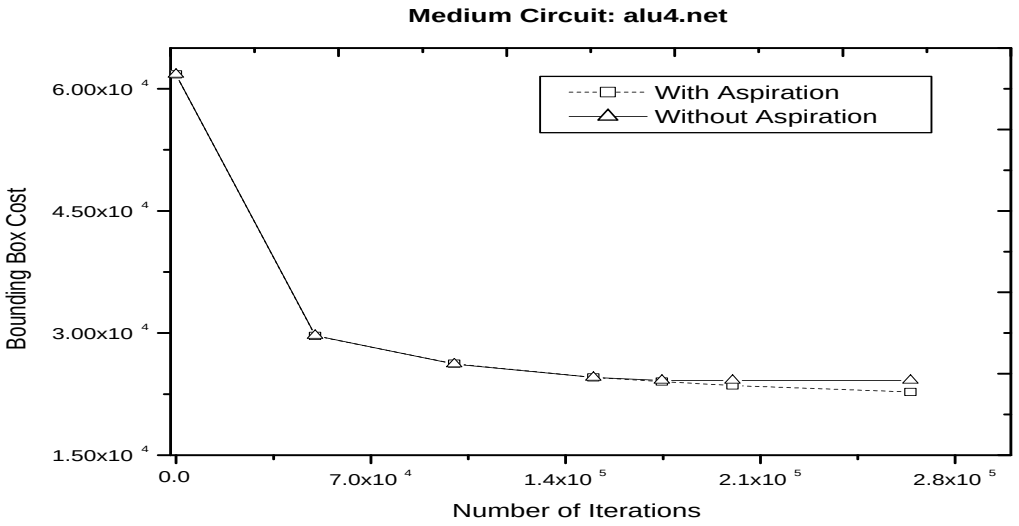


Figure 4.16: Effect of Tabu search with/without aspiration (Medium Circuit)

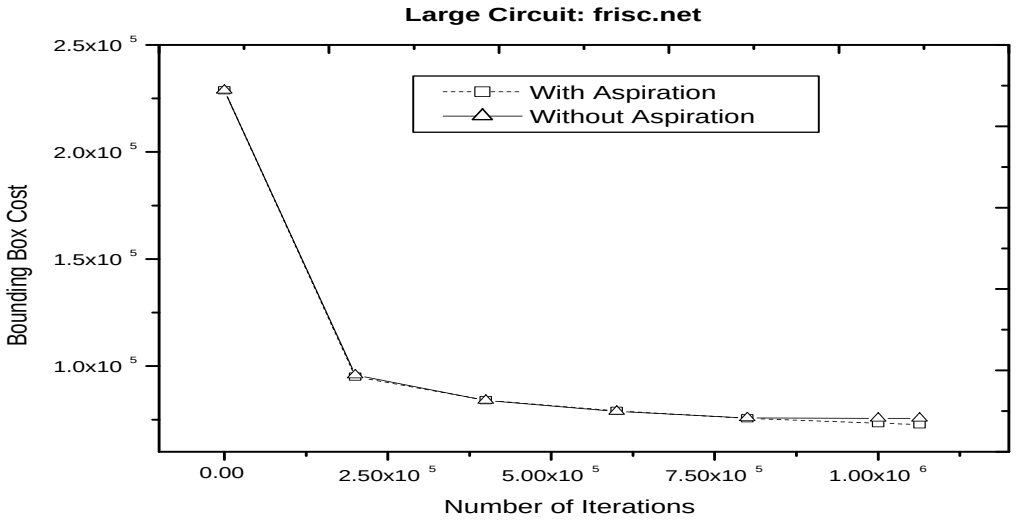


Figure 4.17: Effect of Tabu search with/without aspiration (Large Circuit)

4.3.5 Stopping Criteria

The effectiveness of iterative algorithms partially depends on the number of iterations executed. If a larger number of iterations are allowed by iterative algorithms, they can produce better solutions at the expense of longer CPU runtime, as shown by Figure 4.18. The amount of iterations executed in Tabu Search is associated with the size of the circuit and calculated by the following equation:

$$Iteration_{max} = \alpha \times (N_{CLBs} + N_{Pads}) \tag{4.6}$$

where N_{CLBs} is the total number of CLBs and N_{Pads} is the total number of Pads in the circuit. The parameter α is used to control the depth of the search. The algorithm terminates when a number of iterations have passed without improving the best solution found so far.

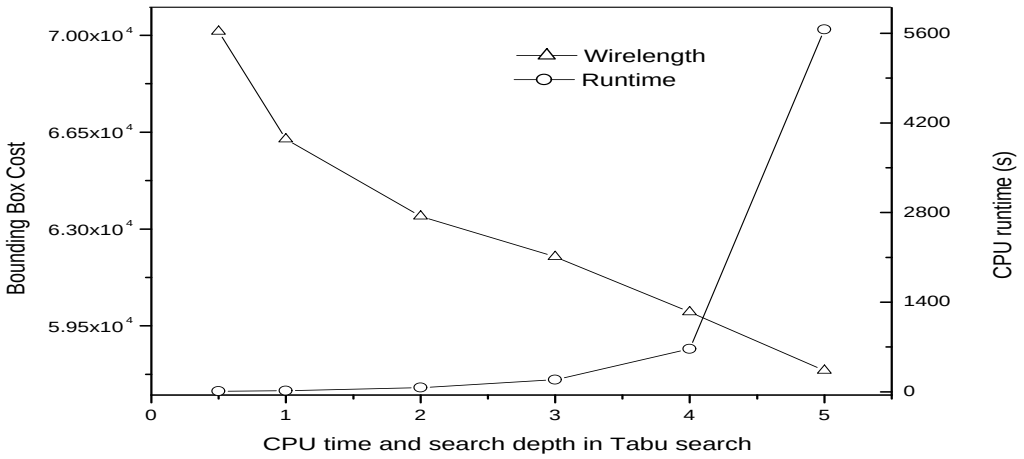


Figure 4.18: Evaluating the Stopping Criteria

4.3.6 Performance of Tabu Search for FPGA Placement

In order to investigate the performance of Tabu Search algorithm for FPGA placement, the algorithm was run 20 times on ten MCNC benchmarks. Tabu Search in effect guided two different neighborhood interchange algorithms, SLS and INLS respectively. Tables 4.6 and 4.7 provide experimental results based on random initial solutions. Table 4.8 presents a comparison between TS based on INLS and SLS. It is clear from the tables that TS based on INLS outperforms TS based on SLS. It achieves on average 12% improvement on medium-size circuits, 30% improvement on large-size circuits and 28% average improvement on all circuits. Figure 4.19 indicates that TS guides INLS to converge to a better solution than SLS. The parameter settings used for the implementation are as follows:

Tabu Search Parameter Settings

Tabu Criteria	TC1 or TC2
Tabu List Size	$T_{size} = \frac{N_{CLBs}+N_{PADs}}{L_{matrix}\times 4}$
Aspiration	Override tabu status if better than best found so far
Stopping Criteria	$Iteration_{max} = \alpha \times (N_{CLBs} + N_{Pads})$

In addition, Cluster Seed Search was used to provide good starting solutions for Tabu Search. Tables 4.9 and 4.10 give results obtained by running the algorithm 20 times. By starting from improved solutions, the algorithm achieves approximately on average 8% improvement compared to solutions based on random starting points.

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64	3207	3471	3084	36	0.3	0.6	0.2	0.1
tseng	11515	12292	10752	236	2	3	1	0.6
ex5p	17733	18684	17071	257	2	4	1	0.6
alu4	22325	23950	21259	504	4	6	2	1
seq	29793	33351	28381	878	5	8	2	2
M.avg	20341	22069	19365	468	3	5	1.5	1
frisc	71828	75016	67555	2053	17	23	12	12
spla	77651	82175	74512	2635	24	31	17	4
ex1010	86293	90563	79346	4008	30	36	23	3
s38584.1	113884	124018	104001	5892	56	81	40	12
clma	211910	215553	204216	4500	100	122	88	12
L.avg	112313	117465	105926	3817	45	58	36	8
Avg	64613	67907	61017	2099	24	31	18	4

Table 4.6: TS Based on INLS with random initial solutions

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64	3355	3598	3172	103	7	9	6	0.6
tseng	12546	13020	12253	248	96	112	82	8
ex5p	18569	19114	17919	324	79	90	71	6
alu4	25124	25573	24830	200	92	103	79	7
seq	34085	35077	33096	555	160	185	143	14
M.avg	22581	23196	22024	332	106	122	94	9
frisc	90769	92466	88873	1345	688	801	595	40
spla	99286	99798	99040	297	664	740	591	62
ex1010	115147	118984	109520	4310	1398	1740	1141	236
s38584.1	174751	175943	172485	1963	3131	3538	2877	356
clma	290146	293309	285695	3967	8749	9121	8225	218
L.avg	154021	156098	151122	2376	2925	3027	2685	128
Avg	86378	87687	84688	1331	1505	1564	1382	92

Table 4.7: TS Based on SLS with random initial solutions

Circuit name	TABU-SLS		TABU-INLS		Improvement	
	B.B.cost	CPU.t(s)	B.B.cost	CPU.t(s)	B.B.cost	CPU.t(s)
e64	3172	6	3084	0.2	3%	96%
tseng	12253	82	10752	1	12%	98%
ex5p	17919	71	17071	1	5%	98%
alu4	24830	79	21259	2	14%	79%
seq	33096	143	28381	2	14%	96%
M.avg	22024	94	19365	1.5	12%	97%
frisc	88873	595	67555	12	24%	97%
spla	99040	591	74512	17	25%	97%
ex1010	109520	1141	79346	23	27%	98%
s38584.1	172485	2877	104001	40	40%	98%
clma	285695	8255	204216	88	28%	99%
L.avg	151122	2685	105926	36	30%	98%
Avg	84688	1382	61017	18	28%	98%

Table 4.8: Comparison between TS based on INLS and SLS

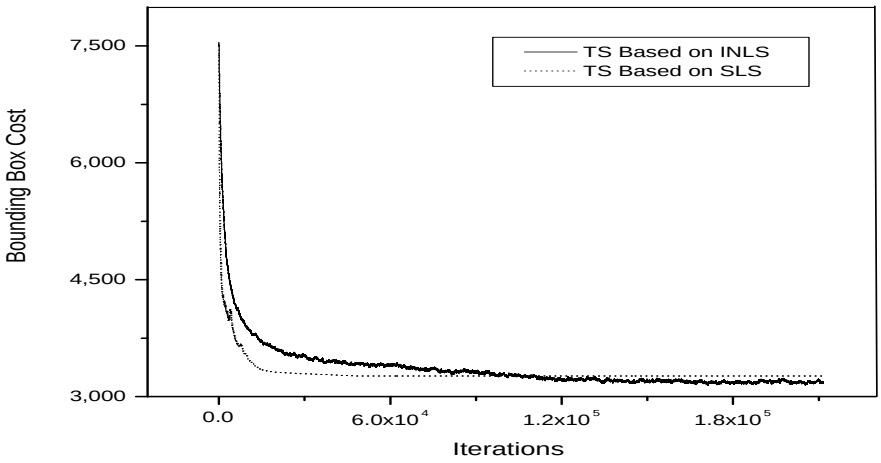


Figure 4.19: TS based on INLS and SLS(circuit: e64-4lut)

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64	3127	3378	2993	51	0.4	0.9	0.2	0.1
tseng	11014	12005	10097	226	2	3	1	0.6
ex5p	16641	18809	16936	257	2	4	1	0.6
alu4	21660	23649	21062	427	3	5	2	1
seq	28651	32602	27864	811	5	8	2	1
M.avg	19566	21766	18989	430	3	5	1.5	0.8
frisc	69310	75016	67555	2053	20	29	15	4
spla	72443	81808	73668	2356	21	30	16	6
ex1010	79920	90813	78463	4418	31	36	18	5
s38584.1	101138	111617	93309	2736	59	69	37	6
clma	191341	199073	184121	6594	92	105	75	15
L.avg	102830	111665	99423	3631	44	53	32	7.2
Avg	59554	64877	57606	1992	24	30	20	3.9

Table 4.9: TS based on INLS with CSS initial solutions

Circuit name	Ave. cost	Max. cost	Min. cost	Cost STDEV	Ave. CPU.t	Max. CPU.t	Min. CPU.t	CPU.t STDEV
e64	3351	3612	3094	108	7	9	6	0.6
tseng	12593	13219	10867	368	92	101	84	5
ex5p	18492	18849	17221	214	74	84	65	6
alu4	25025	25468	24478	335	94	109	84	7
seq	34073	34670	32673	349	160	179	142	14
M.avg	22545	23051	21309	316	105	118	93	8
frisc	89629	91756	88380	1322	684	801	544	109
spla	98406	99200	96527	297	705	839	638	91
ex1010	100852	108580	96164	5399	1223	1370	1113	101
s38584.1	159394	162269	157332	1976	3228	3341	3085	127
clma	264000	265571	261962	1849	8639	9196	8141	223
L.avg	142456	145477	140073	2168	2895	3109	2704	130
Avg	80581	82320	78869	1220	1489	1602	1390	78

Table 4.10: TS based on SLS with CSS initial solutions

Circuit name	Rand-TS-INLS		CSS-TS-INLS		Improvement	
	B.B.cost	CPU.t(s)	B.B.cost	CPU.t(s)	B.B.cost	CPU.t(s)
e64	3207	0.3	3127	0.4	2%	-33%
tseng	11515	2	11014	2	4%	0%
ex5p	17733	2	16641	2	4%	0%
alu4	22325	4	21660	3	3%	25%
seq	29793	5	28651	5	4%	0%
M.avg	20341	3	20166	3	4%	0%
frisc	71828	17	69310	20	4%	-18%
spla	77651	24	72443	21	7%	13%
ex1010	86293	30	79920	31	7%	-3%
s38584.1	113884	56	101138	59	11%	-5%
clma	211910	100	191341	92	10%	8%
L.avg	112313	45	102830	44	8%	2%
Avg	64613	24	59554	24	8%	0%

Table 4.11: Comparison between TS with random/CSS initial solutions

4.4 Genetic Algorithms

In this section, a Genetic Algorithm for the FPGA placement is proposed and the details of the implementation are further discussed. To observe the effect of evolution operators on the performance of placement, several selection method and replacement methods are investigated and implemented.

A Genetic Algorithm implementation for FPGA placement is illustrated in Figure 4.20. The algorithm starts by constructing a set of random initial placement solutions that are coded as the (x,y) co-ordinates of the blocks in the FPGA chip and are called **chromosomes**. Next, this initial/legal population is evaluated according to the placement-specific fitness function. The binary tournament selection method is applied to choose parents for reproduction. By exchanging part of the selected parent's structure, the **Crossover** operator transforms parents into two new

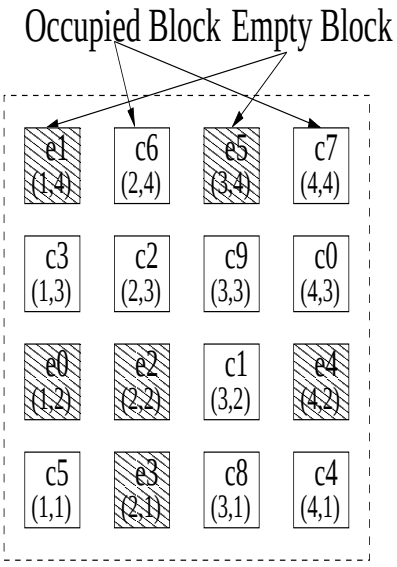
```
GA for FPGA Placement
1. Read Inputfiles ;
2. Set popSize, crossRate, mutateRate, geneSize;
3. Set selection method, replacement method;
4. Construct random initial and legal population;
5. While  ( geneSize is not reached )
6.      For (i=1 to popsize/2)
7.        Select_parents(chrom1,chrom2,selectMethod);
8.        if (random(0,1) ≤ crossover_rate)
10.       if (random(0,1) ≤ mutation_rate)
11.           Mutation(offspring) and evaluate offspring;
12.    End For
13.       Replacement(selectMethod);
14.       generation = generation + 1;
15.   End While
16. Return best placement solution;
```

Figure 4.20: A Genetic FPGA Placement Algorithm

individuals called **offspring**. Each offspring inherits partial features from their parents. The **Mutation** operator follows up by randomly and incrementally changing the genes contained in the offspring. The **Mutation** operator basically expands the exploration capability of the Genetic Algorithm such that the search space is not merely constrained into the finite population size. In this implementation, the replacement method is implemented in several ways. The first implementation attempts to replace the most inferior member of a population by a new superior offspring. The second technique tends to replace the previous population with the newly generated population. A one-point **order crossover** is used since traditional crossover operators tend to genenrate infeasible solutions for the problem.

4.4.1 Encoding Mechanism

The encoding mechanism enables GA indirectly to deal with the optimization problem by utilizing the proper representation of the problem's variables. A good encoding method is very crucial for any GA implementation. In the current implementation, an FPGA placement solution string is represented by the (x,y) co-ordinates of the blocks in the FPGA chip. Figure 4.21(b) illustrates the chromosome of an encoded solution representing the actual placement solution given in Figure 4.21(a).



block	c5	e0	c3	e1	c6	c2	e2	e3	c8	c1	c9	e5	c4	e4	c0	c7
X	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
Y	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

(a) Placement solution (b) String encoding of solution

Figure 4.21: String encoding

4.4.2 Tournament Selection with/without Replacement

In Genetic Algorithms, the selection operator resembles nature's survival of the fittest mechanism. Superior solutions create more offspring and get higher chances to survive in the evolution than inferior solutions. A variety of selection methods, such as *roulette wheel selection, stochastic universal selection* and *binary tournament selection* [Mazu99], have been broadly applied in Genetic Algorithms. In general, selection schemes provide the selection pressure (a drive force) to determine the rate of convergence. In order to avoid premature convergence to sub-optimal solution, **binary tournament selection** is used to exert more selection pressure in later generations when the fitness values of overall individuals become similar. Binary tournament selection method can be performed with or without replacement respectively. Two individuals are chosen randomly from the population, and the better individual with a higher fitness value is considered as a candidate. The "**Without Replacement**" strategy sets aside the two individuals such that they are not merged again into the pool. On the other hand, in the "**With Replacement**" selection, two individuals are immediately replaced into the current population for further selection. Several experiments were carried out on circuits with and without replacement schemes. As shown in Figures 4.22, 4.23 and 4.24, binary tournament selection "**With Replacement**" provides more selection pressure than "**Without Replacement**".

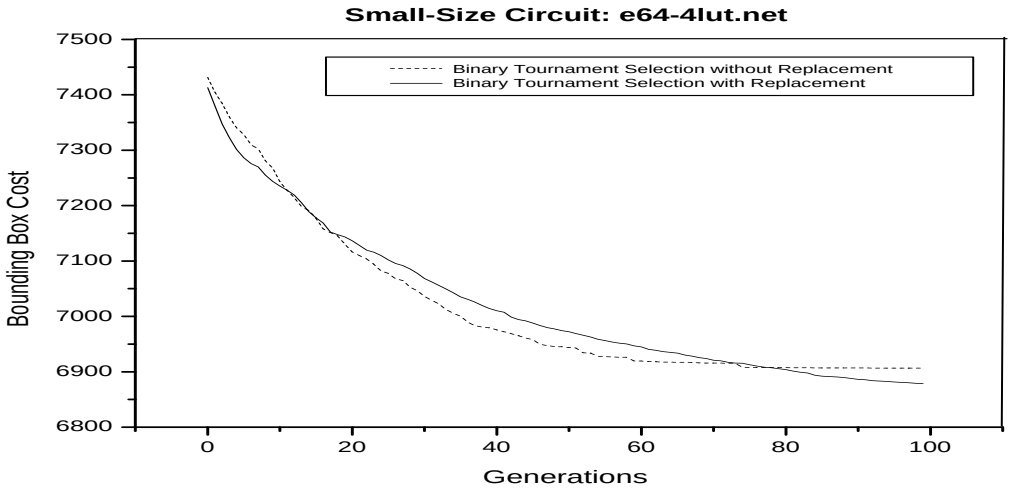


Figure 4.22: Effect of binary tournament selection with/without replacement on a small-size circuit

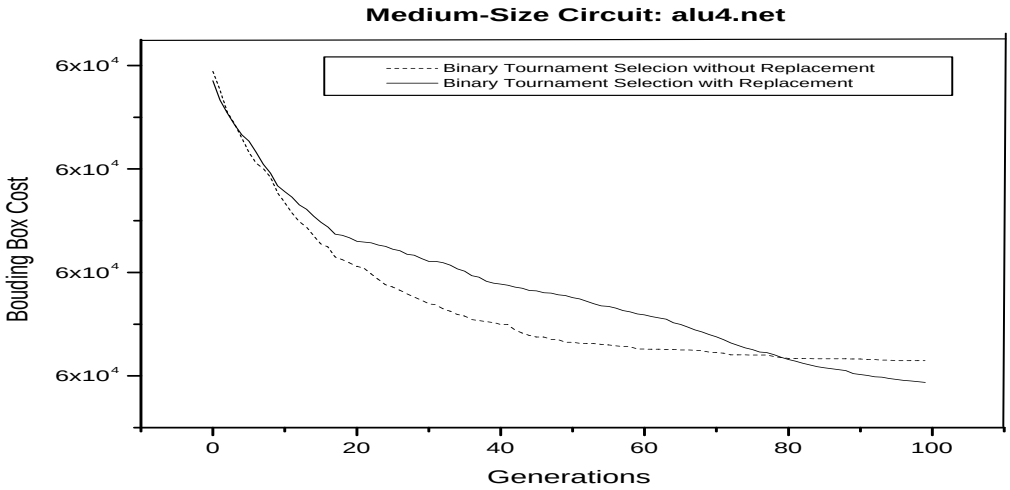


Figure 4.23: Effect of binary tournament selection with/without replacement on a medium-size circuit

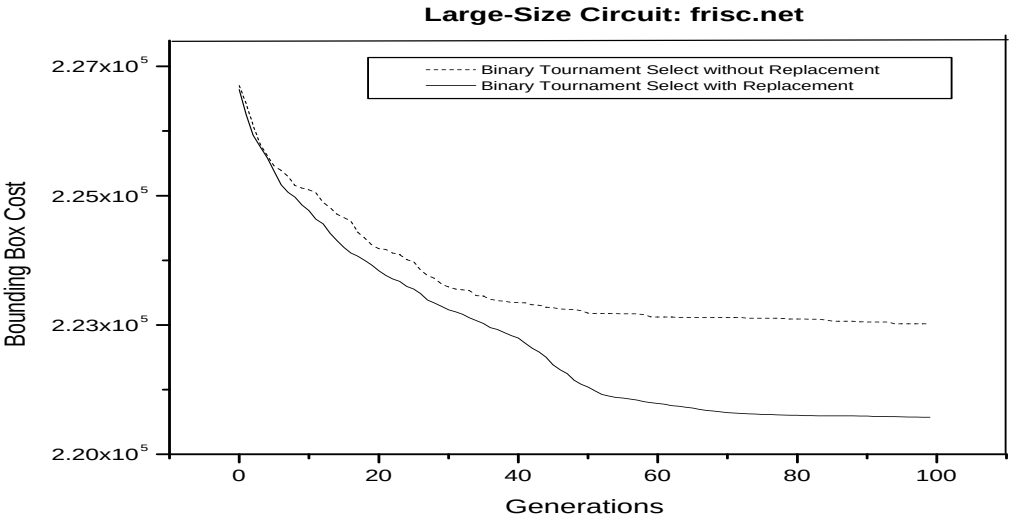


Figure 4.24: Effect of binary tournament selection with/without replacement on a large-size circuit

4.4.3 Crossover

Crossover is GA's crucial operator to generate offsprings, following the selection process. Pairs of individuals are chosen based on the selection strategy used from the population and subjected to the *crossover* operator. Traditional crossover operators that work well with bit string representation may yield illegal solutions for the FPGA placement problem. Therefore, to keep the efficiency of the search process, the order crossover operator [Mazu99] is utilized for the FPGA placement problem. Figure 4.25 illustrates "one-point order crossover" operator where each pair of parents produces two children with a possibility equal to the crossover rate. In this method, a single cut point is generated at random within the pair of parents. The crossover operator then copies the array segment to the left point from one

parent to one offspring. It then attempts to fill the remaining part of the offspring by scanning the other parent and taking those elements that were left out, in order [Mazu99]. Following the selection process, the crossover operator is performed only if a randomly generated number between 0 and 1 is less than crossover rate P_c [1].

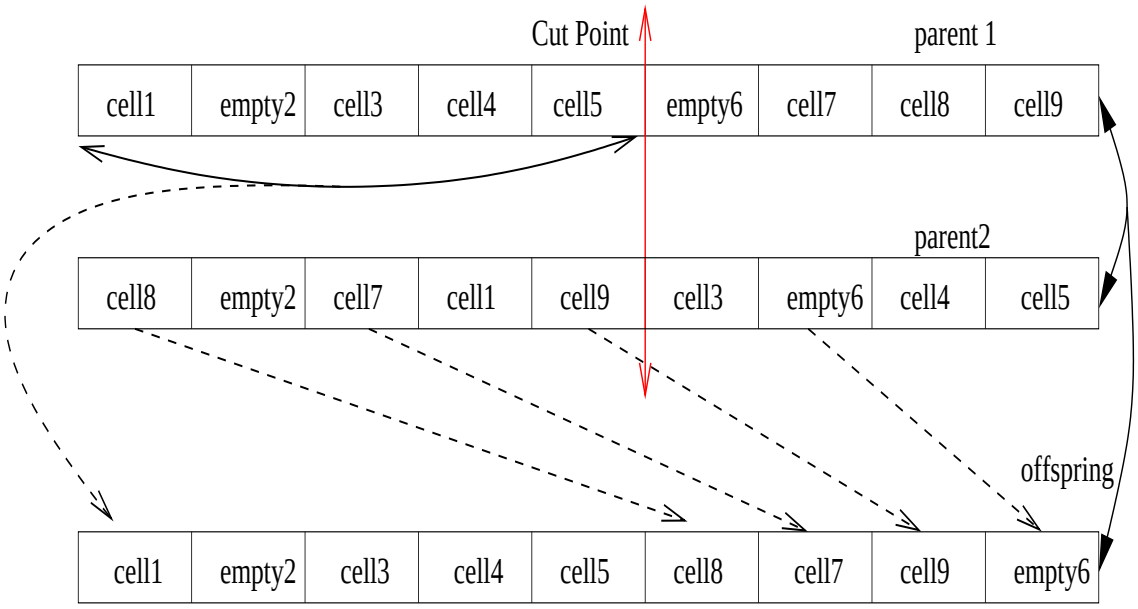


Figure 4.25: One-point order crossover for FPGA placement

4.4.4 Mutation

Following crossover, each offspring is subjected to the mutation operator with a probability equal to the mutation rate P_m. The mutation rate P_m defines the probability of swapping the position of blocks pairwise in solutions. The mutation operator in practice alters incrementally an offspring reproduced through crossover

[1] The crossover rate P_c denote the probability of crossover.

and serves the key role of restoring gene materials lost from the population during the selection process. It also plays a role of creating the gene values that were not presented in the initial population. Figure 4.26 illustrates mutation in FPGA placement, where the coordinate of a randomly selected pair of blocks is exchanged based on the mutation probability.

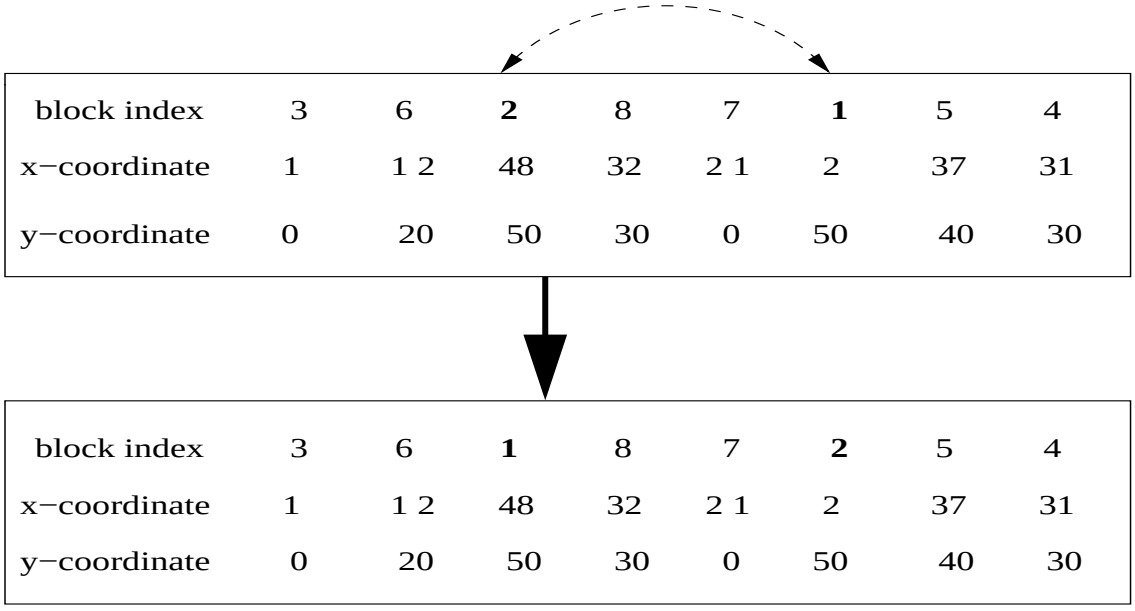


Figure 4.26: Mutation process for FPGA placement

4.4.5 Replacement Method

Following mutation, the population of the next generation is created by combining individuals of parents and offsprings based on a certain strategy. Two replacement methods are applied in the FPGA placement. In order to keep the best individuals within a population, an *elitism* replacement is used, where two parents and two children are compared and the two fittest individuals are restored and injected

back into new population. The *elitism* strategy guarantees that the best individual in the current parents and new offsprings always move to the subsequent generation automatically, protecting the search from regression. The second strategy, *non-elitism* replacement, simply replaces the whole parent population with new offsprings. Figures 4.27, 4.28 and 4.29 show the effect of elitism and non-elitism replacement on the FPGA placement. It is clear from these figures that the replacement strategy based on elitism achieves far better results than those based on no-elitism technique, especially at later stages of the search.

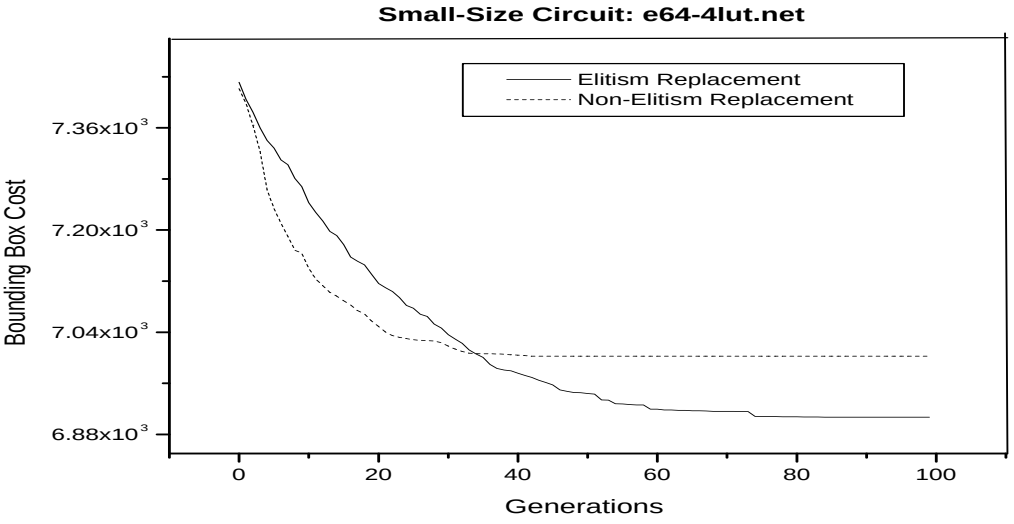


Figure 4.27: Effect of elitism and non-elitism replacement on a small-size circuit

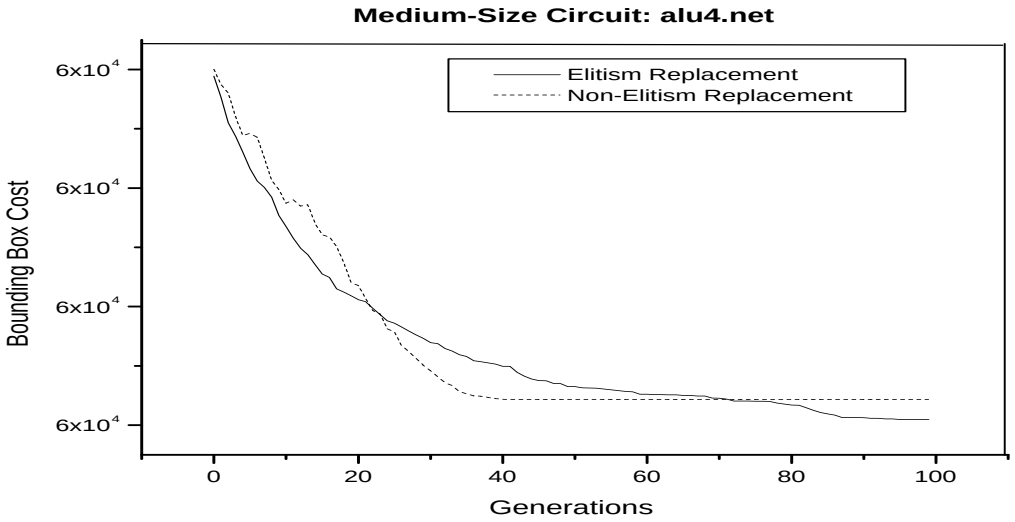


Figure 4.28: Effect of elitism and non-elitism replacement on a medium-size circuit

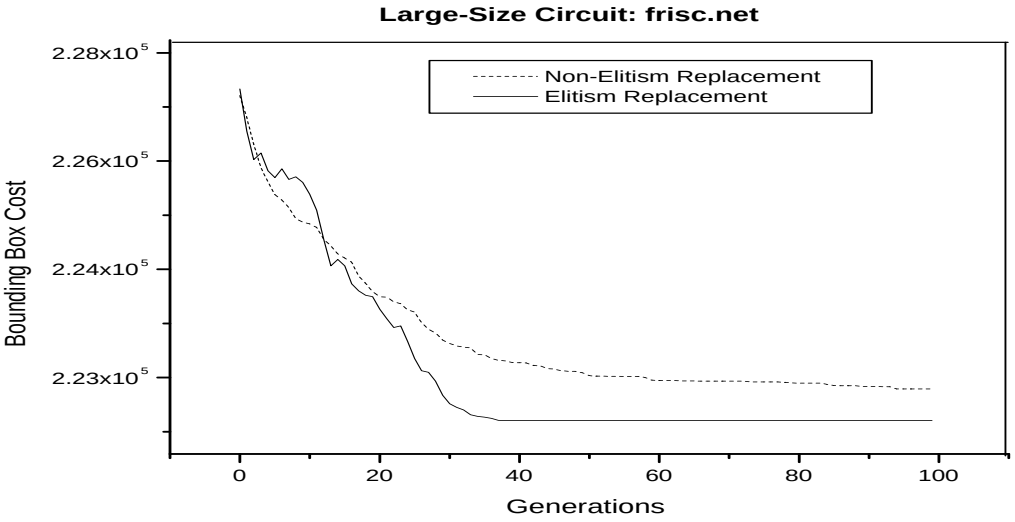


Figure 4.29: Effect of elitism and non-elitism replacement on a large-size circuit

4.4.6 GA Parameter Tuning

To further enhance the performance of the GA proposed in section 4.4, the algorithm was run for different population sizes, number of generations, crossover rates and mutation rates. As shown in Figures 4.30 through 4.35, the performance of the GA improves dramatically as the population size and number of generations are increased. Figures 4.33 through 4.35 also indicate that the Genetic Algorithm shows a very rapid improvement in the beginning of the search and ultimately levels off at the later stages of the search. Hence, if high-quality placement is not an essential issue, the pure Genetic Algorithm can be terminated at an early stage. The effect of crossover rate is illustrated in Figures 4.36, 4.37 and 4.38. The higher the crossover rate (close to 100%), the better the quality of the solution achieved. However, the reverse can be said about the effect of the mutation rate as shown in Figures 4.39 4.40 and 4.41. A too low or too high mutation rate possibly can deteriorate solution quality.

In order to investigate the performance of GA with improved initial solutions, we inject a portion of good initial solutions into a random initial population. The good initial solutions are constructed by the GRASP technique. Experiments conducted on small, medium and large size circuits (as illustrated in Figures 4.42, 4.43 and 4.44) indicate that the quality of solutions obtained by the GA technique are enhanced by the injection of high performance individuals within the initial population. However, increasing the number of good initial solutions may lead to premature convergence and prevent the GA from diversifying the search and limit the exploration capability of the algorithm.

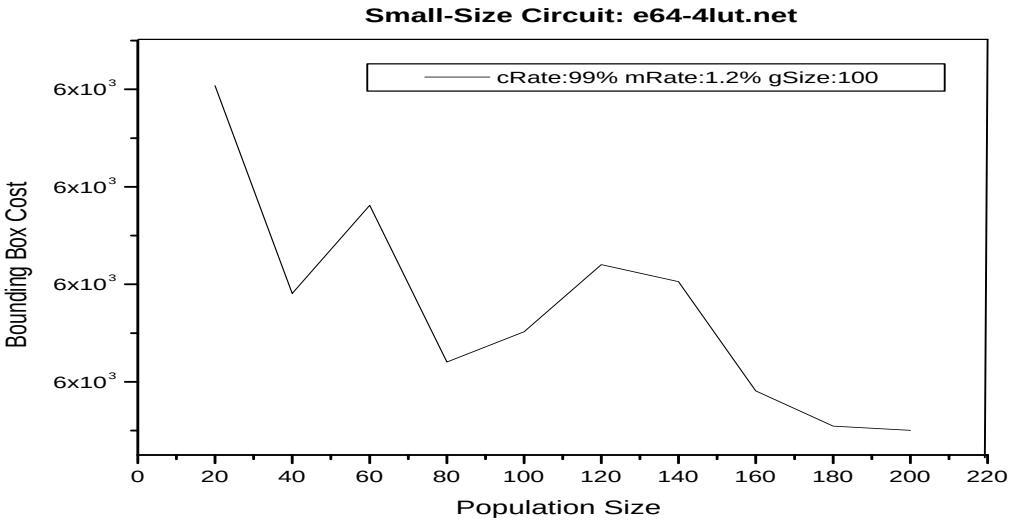


Figure 4.30: Effect of population size on a small-size circuit

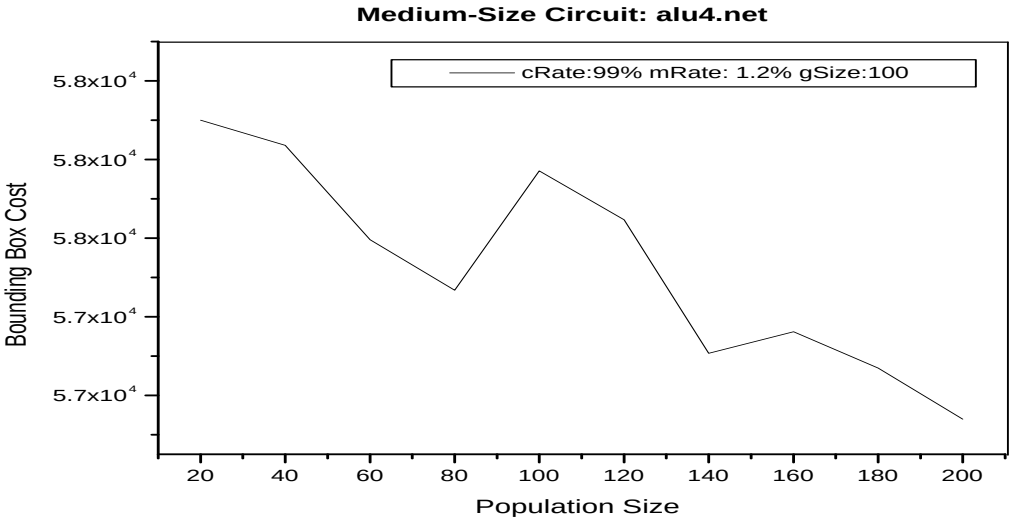


Figure 4.31: Effect of population size on a medium-size circuit

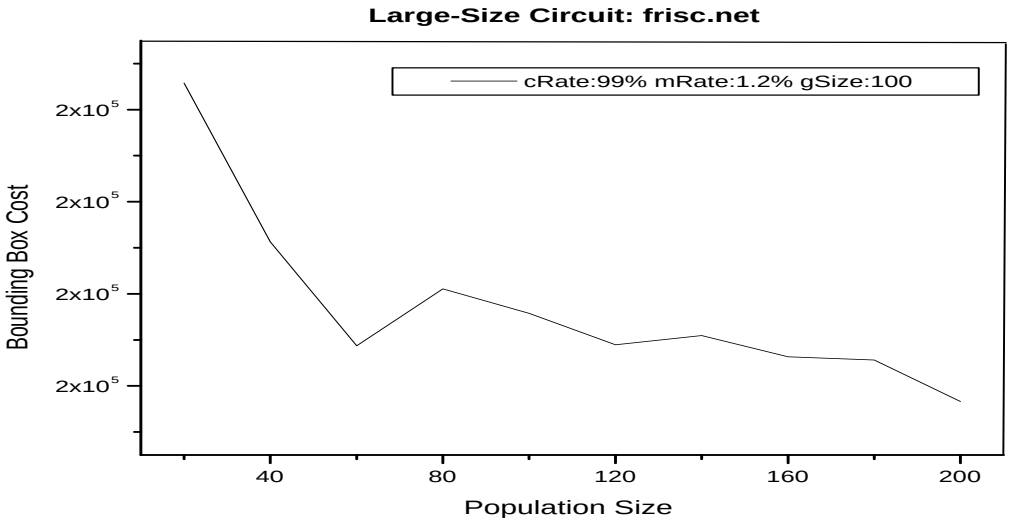


Figure 4.32: Effect of population size on a large-size circuit

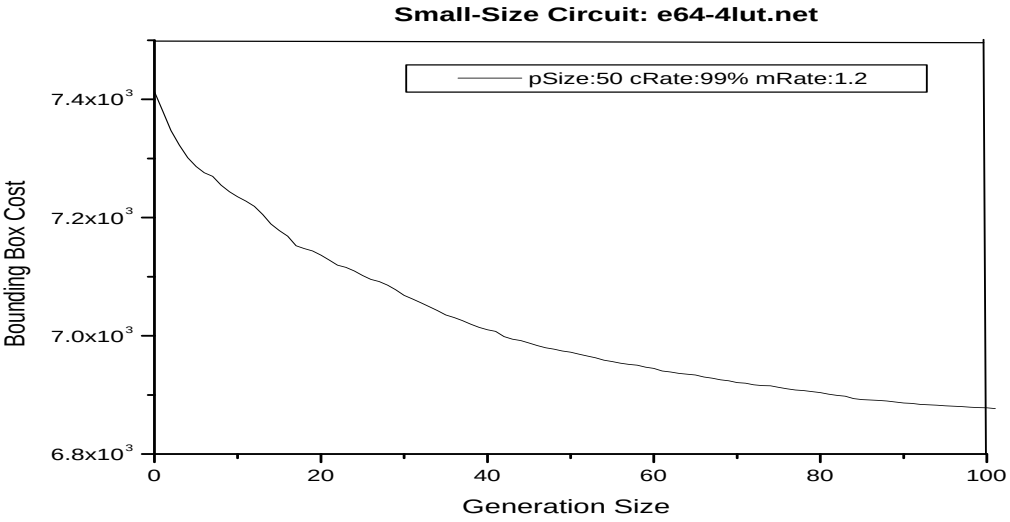


Figure 4.33: Effect of generation size on a small-size circuit

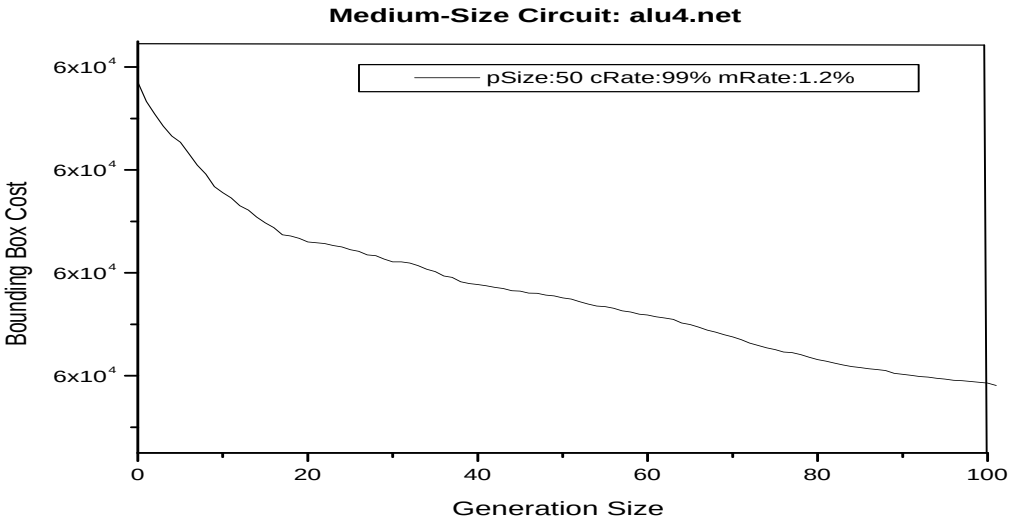


Figure 4.34: Effect of generation size on a medium-size circuit

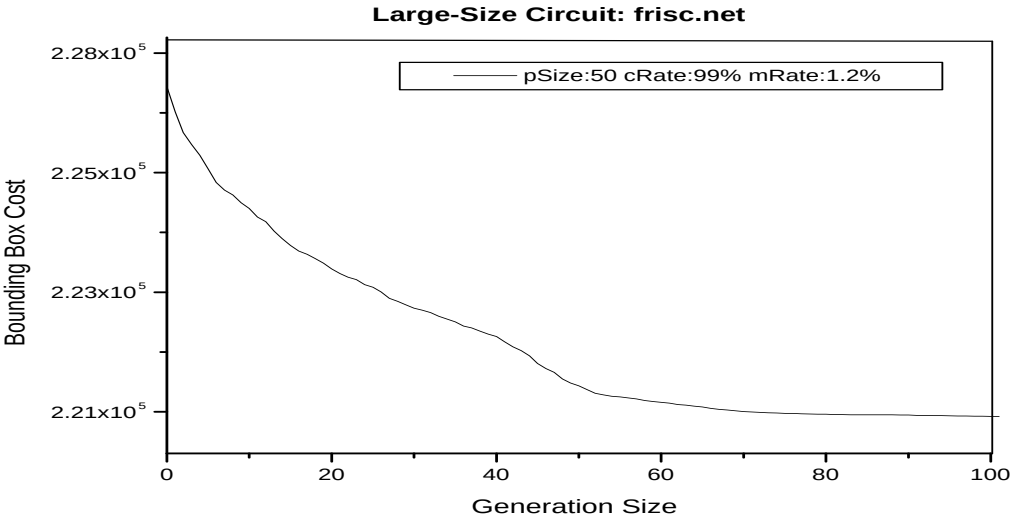


Figure 4.35: Effect of generation size on a large-size circuit

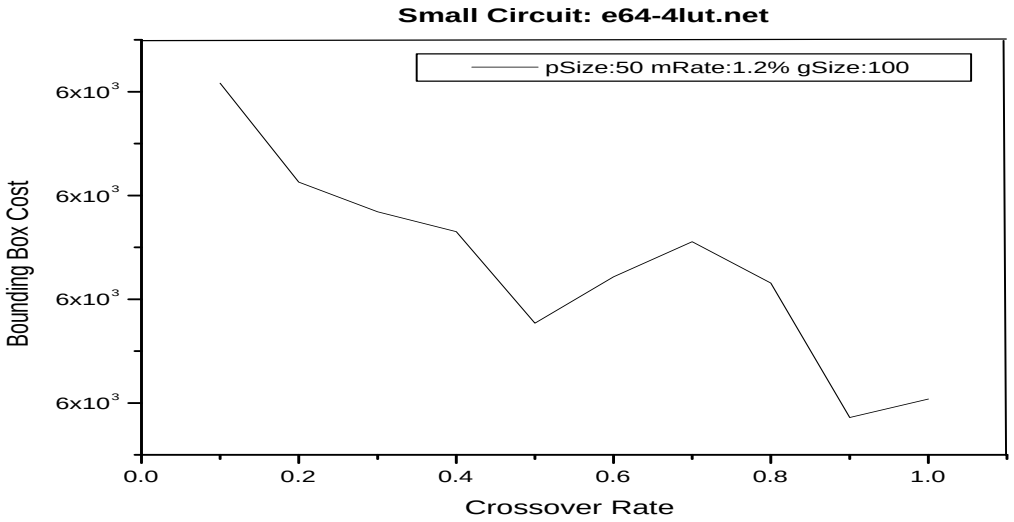


Figure 4.36: Effect of crossover rate on a small-size circuit

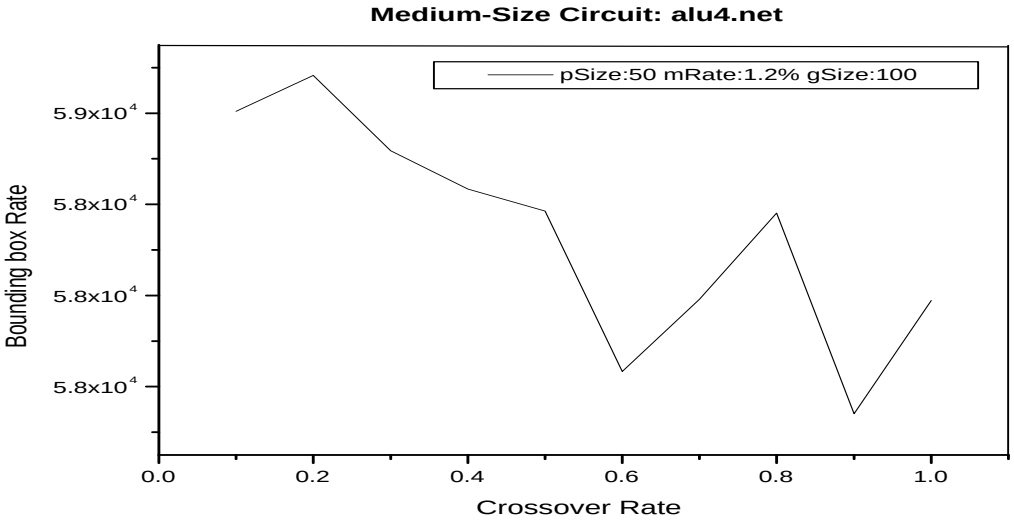


Figure 4.37: Effect of crossover rate on a medium-size circuit

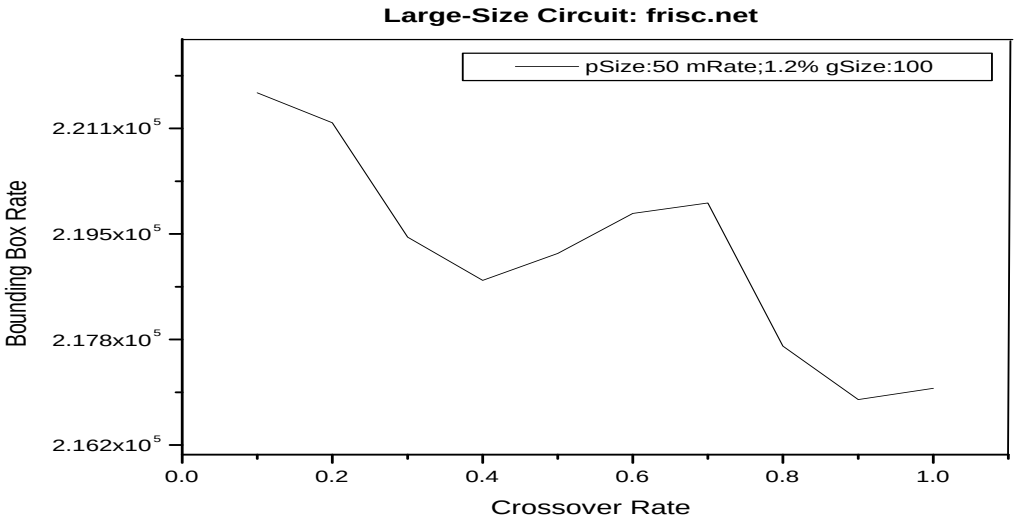


Figure 4.38: Effect of crossover rate on a large-size circuit

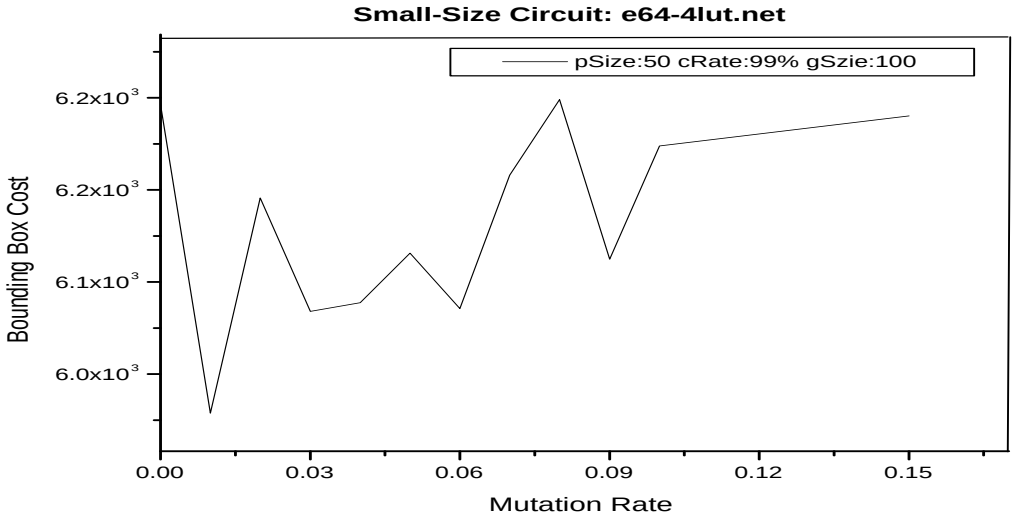


Figure 4.39: Effect of mutation rate on a small-size circuit

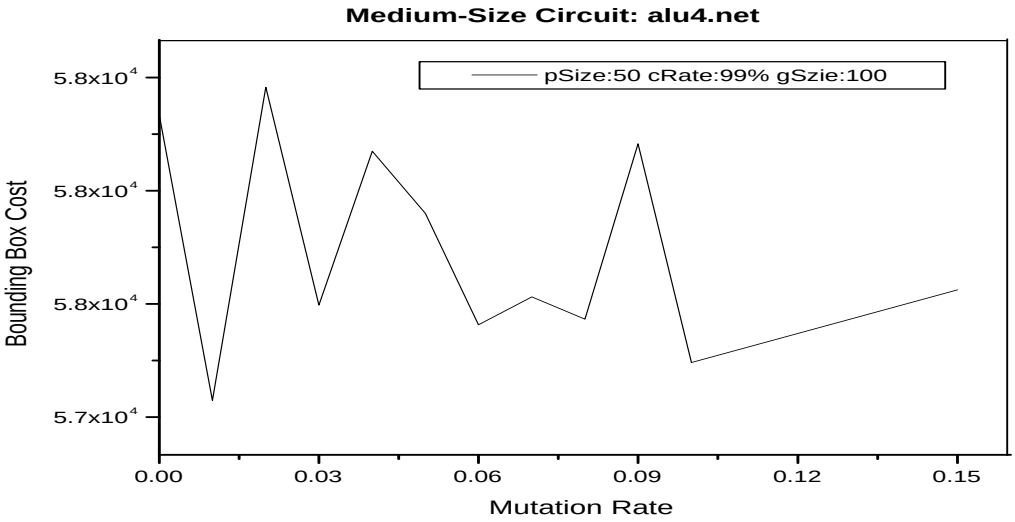


Figure 4.40: Effect of mutation rate on a medium-size circuit

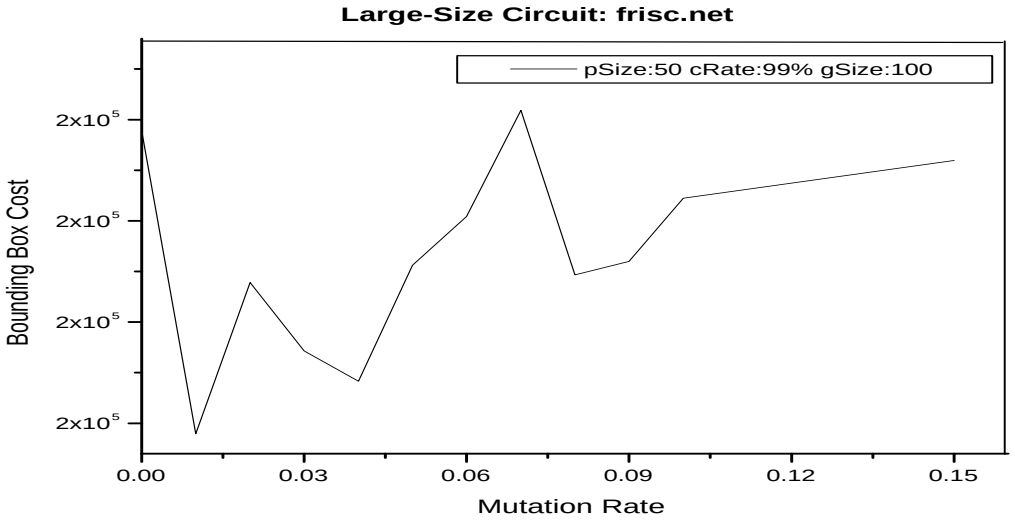


Figure 4.41: Effect of mutation rate on a large-size circuit

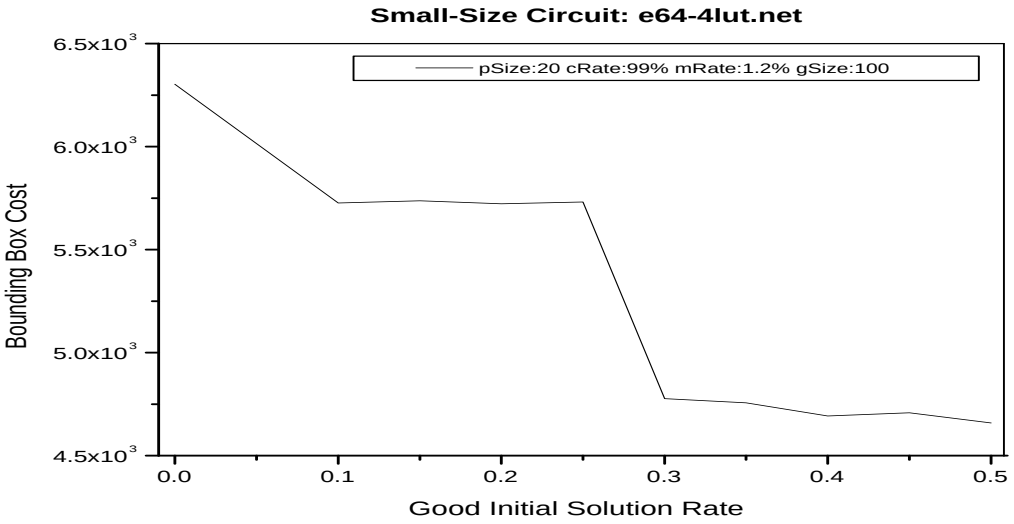


Figure 4.42: Effect of injecting good solutions on a small-size circuit

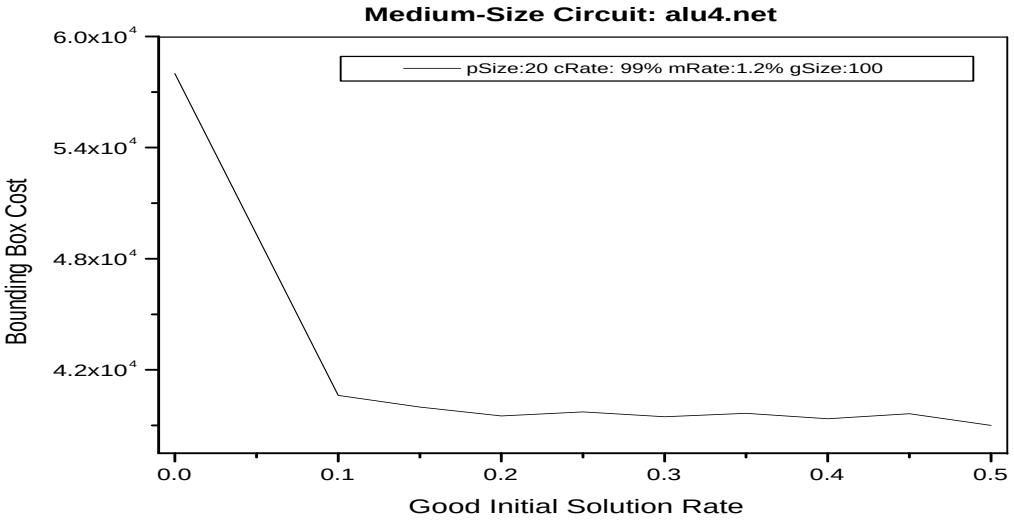


Figure 4.43: Effect of injecting good solutions on a medium-size circuit

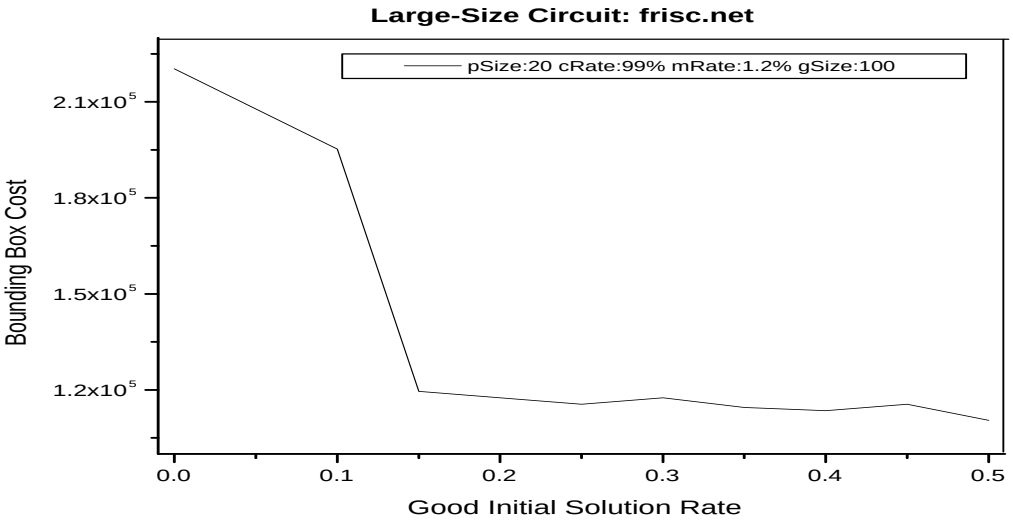


Figure 4.44: Effect of injecting good solutions on a large-size circuit

4.5 Iterative Techniques and Metaheuristics: A Comparison

4.5.1 Flat Level Evaluation

In this section, our goal is to investigate the performance of iterative based algorithms and metaheuristics explained earlier on flat level designs. Experiments conducted were based on both random initial solutions and good starting points constructed by CSS. The parameter settings for Simple SA were described in section 4.2.2 and those settings for TS were described in section 4.3.6. The parameter settings for GA are as follows: population size 50, generation size 200, crossover rate 99%, mutation rate 1.2%. In addition, the GA uses binary tournament selec-

tion with replacement and elitism replacement methods. We also plot placement solutions and corresponding CPU runtime on different sizes of benchmark circuits. As shown in Table 4.12, Figures 4.45 and 4.46, Tabu search based on INLS achieves comparable results to those achieved by Simple SA in a fraction of CPU runtime required by Simple SA. Moreover, the quality of solutions obtained by TS based on INLS are competitive with those obtained by VPR for small and medium sized circuits in less CPU time. Immediate neighborhood local search takes less time to obtain the acceptable quality of placement solutions. Experimental results shown in Figure 4.13 clearly indicate that good starting points enable iterative based techniques to obtain better solutions than randomly generated solutions. In the GA, the crossover and mutation operators do not guarantee that the quality of offsprings are superior to those of parents in each generation. This phenomena lead GA to converge earlier than expected. Figure 4.46 also indicates that CPU time required by the GA increases exponentially as the size of benchmark circuits becomes larger.

Circuit name	Random initial cost	**SLS**		**INLS**		**Simple SA**		**TS-INLS**		**GA**		**VPR**	
		Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)
e64	7542	4006	0.06	4004	0.04	2973	3.9	3084	0.2	6142	2.5	2858	13
tseng	41286	16478	0.32	15803	0.23	11038	19.1	10752	1	36588	16	9394	71
ex5p	42301	21670	0.33	21352	0.24	17474	19.3	17071	1	39499	18	16227	70
alu4	61504	28797	0.46	28635	0.35	21391	32.2	21259	2	57977	49	19161	104
seq	79903	39080	0.62	39096	0.50	27676	36.2	28381	2	73656	89	24736	142
M.avg	46489	26506	0.43	26221	0.33	19395	26.8	19365	1.5	51930	43	17380	97
frisc	229152	102676	1.67	102901	1.28	64316	96.1	67555	12	209470	368	51256	392
spla	236251	111485	1.74	110372	1.38	71498	103	74512	17	211482	401	61046	414
ex1010	332664	138229	2.38	138479	3.0	77838	154	79346	23	295597	534	65493	554
s38584.1	559870	205301	3.97	204574	3.62	96323	216	104001	40	519922	1352	64925	905
clma	796591	332142	6.82	330038	6.07	179416	319	204216	88	736508	4230	140391	1332
L.avg	430905	177967	3.31	177272	3.07	97878	178	105926	36	394595	1377	76802	720
Avg	238697	99986	1.88	99575	1.61	56994	100	61017	18	218683	706	45639	400

Table 4.12: Comparison between iterative techniques based on random solutions

Circuit name	Initial solutions by CSS	SLS		INLS		Simple SA		TABU-INLS		GA	
		Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave cost	Ave. t(s)
e64	6992	3904	0.1	3905	0.02	2977	3.9	2993	0.2	5873	3
tseng	34808	16036	0.41	15729	0.18	10793	18.9	10097	1	33648	19
ex5p	37381	21002	0.42	20445	0.22	17386	19.1	16936	1	36903	20
alu4	53175	27956	0.56	27517	0.27	21295	31.7	21062	2	52651	53
seq	69390	38062	0.81	37174	0.36	27558	36.4	27864	2	67152	94
M.avg	40349	25766	0.55	25216	0.26	19265	26.5	18989	1.5	47566	46
frisc	177393	94479	2.09	94450	1.04	64080	94.4	67555	15	183632	382
spla	181525	101300	2.13	101296	1.65	70587	102	73668	16	189344	421
ex1010	264977	128431	2.82	128115	3.72	77720	149	78463	18	261837	573
s38584.1	478670	194515	5.27	194362	3.94	96243	213	93309	37	470019	1402
clma	631368	319227	9.01	318885	6.14	179001	304	184121	75	672630	4395
L.avg	346786	167594	4.26	167421	3.31	97526	172	99423	32	355492	1434
Avg	193657	94493	2.34	94180	1.76	56767	97	57606	20	197360	736

Table 4.13: Comparison between iterative techniques based on CSS

4.5.2 Hierarchical Performance

Hierarchical design tends to reduce the complexity of circuits by clustering them into less complicated and easily solvable representations. In this section, the hierarchical performance of iterative techniques is also investigated. In the hierarchical experimental setup, the clustering scheme is the clustering level $\mathbf{L = 2}$ and clustering size at each level $\mathbf{S = 4}$. Firstly, the clustering technique proposed by Peng [Du03] groups blocks into clusters. The algorithms performs the placement on this top level. Next, the current hierarchical placement solution is declustered to the flat level. Finally, the algorithms are carried out over the flat level with the same parameter sets.

Table 4.14 provides the results of hierarchical placement, including placement

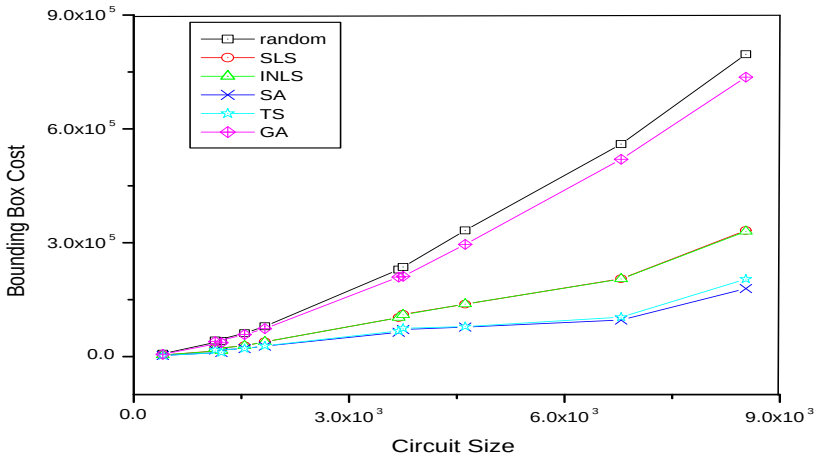


Figure 4.45: Wirelength comparison of iterative techniques

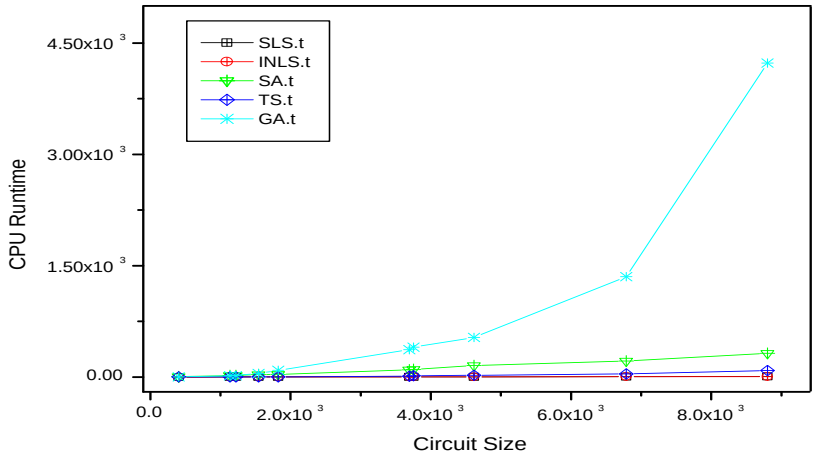


Figure 4.46: CPU time comparison of iterative techniques

quality and CPU time, obtained by SLS, INLS, Simple SA, TS based on INLS and GA. Experimental results from Table 4.14 based on 10 runs indicate that INLS outperforms SLS by limiting the exploring neighbourhood to the adjacent region. Undoubtedly, SA achieves the best solution quality in all approaches. TS achieves comparable solution quality to SA, especially on small and medium sized circuits. On the other hand, GA produces the worst results compared to other techniques.

Circuit name	Random initial cost	**SLS-h**		**INLS-h**		**SSA-h**		**TABU-h**		**GA-h**	
		Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave. cost	Ave. t(s)	Ave cost	Ave. t(s)
e64	7542	3777	0.1	3713	0.08	2998	3.8	3205	0.18	4255	2.8
tseng	41286	14443	0.6	14016	0.4	11553	15	10969	0.9	20635	17
ex5p	42301	20574	0.6	20266	0.5	17780	14	17710	0.8	25313	19
alu4	61504	26539	0.8	26590	0.7	21487	21	22475	1.7	33392	51
seq	79903	36561	1.1	36016	0.8	29074	27	30641	2.2	49632	92
M.avg	46489	24539	0.8	24222	0.6	19973	19	20048	1.3	32243	45
frisc	229152	91952	2.8	89583	2.1	65441	75	70273	9	126278	376
spla	236251	112595	1.9	115121	2.1	71587	84	76578	13	139022	410
ex1010	332664	109906	3.8	96918	4.1	78676	116	81164	19	201646	549
s38584.1	559870	166510	8.1	150715	5.4	98878	183	109376	38	296683	1382
clma	796591	280122	9.1	276081	8.9	185041	250	210853	73	432928	4289
L.avg	430905	152215	5.2	145683	4.5	99524	141	109648	30	237311	1401
Avg	238697	86296	2.9	82901	2.5	58051	79	63324	16	131978	718

Table 4.14: Comparison between hierarchical iterative techniques

Table 4.15, Figure 4.47 and Figure 4.48 give the comparison with its flat counterparts evaluated earlier. For local search techniques, SLS and INLS achieve respectively 13% and 14% average improvement in wirelength cost with about 55% increase in CPU time. Simple SA and TS based on INLS mitigate 10% to 20% CPU time to obtain good placement solutions that deteriorate by 3%. Hierarchical GA achieves 40% average improvement in solution quality with a slight increase in CPU time, compared to a flat GA implementation.

Circuit name	SLS-h		INLS-h		SSA-h		TS-h		GA-h	
	A.cost Imp%	A.CPU Imp%	A.cost Imp%	A.CPU Imp%	A.cost Imp%	A.CPU Imp%	A.cost Imp%	A.CPU Imp%	A.cost Imp%	A.CPU Imp%
e64	+6	-67	+7	-100	-0.8	+2.5	-3.9	+10	+30	-12
tseng	+13	-88	+12	-74	-4.6	+22	-2	+10	+44	-6
ex5p	+5	-82	+5	-108	-2	+27	-3.6	+20	+36	-5.6
alu4	+7.8	-74	+7.1	-100	-0.4	+34	-5	+15	+42	-4
seq	+6.4	-77	+7.8	+60	-5.1	+25	-7.9	+10	+33	-3.4
M.avg	+7.5	-86	+7.6	-81	-3	+29	-3.5	+13	+38	-4.7
frisc	+10	-68	+13	-64	-1.7	+22	-4	+25	+40	-2.2
spla	-0.9	-9	-4.3	-52	-0.1	+19	-2.7	+24	+34	-2.3
ex1010	+21	-60	+30	-37	-1.1	+25	-2.3	+18	+32	-2.8
s38584.1	+19	-104	+26	-49	-2.6	+16	-5.1	+5	+43	-2.2
clma	+16	-33	+17	+46	-3.1	+22	-3.2	+17	+41	-1.4
L.avg	+14	-57	+22	-47	-1.7	+21	-3.5	+17	+40	-1.7
Avg	+13	-54	+17	-55	-1.9	+21	-3.7	+11	+40	-1.7

Table 4.15: Comparison between hierarchical and flat iterative techniques

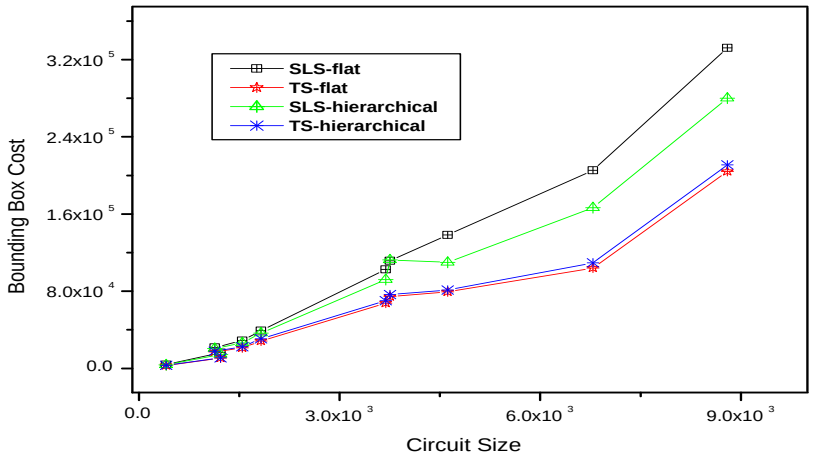


Figure 4.47: Flat cost hierarchical cost obtained by iterative based tenchniques

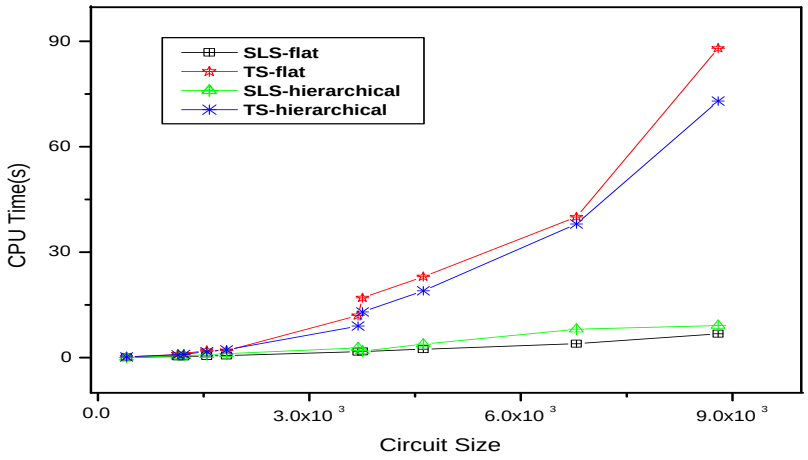


Figure 4.48: Flat CPU time vs hierarchical CPU time by iterative based tenchniques

4.6 Summary

In this chapter, several iterative algorithms for FPGA placement have been investigated. Novel local search algorithms incorporating new features from the traditional local search algorithms were introduced. These features enable LS to run faster and achieve better results than the traditional local search algorithms over some circuits. If this local search is combined with the other new algorithm – Cluster Seed Search, its performance surpasses that of traditional local search. A Tabu Search technique was further developed for the FPGA placement. By employing the novel local search technique within TS, the latter achieves high-quality placement solutions with reasonable CPU time. A Genetic Algorithm implementation was also introduced to solve FPGA placement problems. Results obtained by the GA approach were inferior to those obtained by the other Meta-heuristics. There could be several explanation to this: (i) the encoding method is inefficient (ii) one

point crossover is unable to produce better offsprings (iii) the selection methods used might be driving GA to converge prematurely. A final comparison of all developed heuristic techniques was carried out on flat/hierarchical designs. Results obtained indicate that the developed SLS and INLS approaches perform better in the hierarchical paradigm at the expense of more CPU time. However, the TS and SA based on the hierarchical approach worsened the solution quality but gained speedup of 21% and 11% on average respectively.

Chapter 5

Conclusions and Future Work

The logic capacity of an FPGA device will keep growing according to Moore's law [Trim94], and the complexity of circuit designs based on FGPA will increase accordingly. The time to compile a design is increasingly becoming a major concern for FPGA users. Placement plays a critical role in the design process, and the computational time for placement has a great impact on the effectiveness and efficiency of FPGA design tools. Therefore, it is necessary to develop new algorithms to yield good placement solutions in reasonable amounts of time.

In this thesis, several meta-heuristic algorithms for FPGA placement are presented and investigated. Heuristic techniques are classified into two categories: constructive based algorithms and iterative based techniques. Three constructive methods including Cluster Seed Search (CSS), Greedy Random Adaptive Search Procedure (GRASP) and Partition Based algorithms have been developed and compared. Four iterative methods including Simple Local Search (SLS), Immediate Neighbourhood Local Search (INLS), Tabu Search technique (TS), Simulated An-

nealing (SA) and Genetic Algorithm (GA) have been investigated and applied to flat and hierarchical designs.

Cluster Seed Search (CSS), as a constructive based method, can be implemented in trivial time. In the common VLSI cell placement, constructive placement algorithms are generally based on primitive connectivity. However in the FPGA placement, CSS uses the fanout number criteria to select the best block and create an improved initial and legal placement solution. GRASP is composed of two basic phases: a solution construction phase and local improvement phase. The simplicity of implementing GRASP makes it suitable to deal with the FPGA placement problem. Partitioning based approaches on the other hand run in a comparatively short time by using a divide-and-conquer strategy to reduce the problem space by repeatedly partitioning the problem into subproblems. Obviously, constructive based algorithms result in poor quality of placement (which cannot be accepted as the final solution). Good starting points can help heuristic techniques to converge quickly. Experimental results indicate that local search approaches starting from improved initial solutions can achieves on average 10% improvement over heuristic approaches starting from random initial starting points.

Two enhanced local search techniques for FPGA placement were studied and investigated. The first is implemented as a Simple Local Search (SLS) which uses the simplest iterative improvement strategy. SLS attempts to achieve reduction in wirelength cost by swapping blocks in a window which limits the swapping region. The second implementation is based on an Immediate Neighbourhood Local Search (INLS) paradigm. This technique can achieve high-quality solutions quickly by swapping adjacent blocks around the selected blocks. These local improvement

techniques greatly mitigate the runtime in FPGA placement process, while yielding acceptable quality of placements. To further enhance solution quality, a Tabu Search (TS) technique was applied to the FPGA placement. Experimental results indicate that Tabu Search (TS) based on INLS outperformed that based on SLS. The effectiveness of Tabu Search relies heavily on the definition and size of tabu list and stopping criteria. TS based on SLS and INLS achieves on average 65% and 75% improvement respectively. On the other hand, SLS and INLS yield on average 58% and 59% improvement. With fine tuned parameters, Tabu Search was capable of providing promising placement results. To explore solution space effectively, A Genetic Algorithm was implemented and investigated for the FPGA placement. The application of GA on hierarchical designs proved to be effective where it was possible to achieve 40% on average improvement compared to flat GA.

5.1 Future Work

Similar to other design styles, FPGA design requires tradeoffs. Users may be willing to trade placement quality for reduction in runtime. Our work investigates the performance of different meta-heuristic techniques for the FPGA placement. However several areas within this research can be further investigated and improved upon.

In the future, SLS performance can be further investigated by utilizing the adaptive block selection. When the window is large, blocks are randomly selected. As the window decreases in size, the selection strategy becomes deterministic. Another interesting area for future work involves the hierarchical and hybrid implementa-

tion of different heuristic techniques. A hierarchical approach allows the placement algorithms to operate on a reduced problem size by packing FPGA modules into clusters. Computation time is greatly reduced by an order of magnitude compared to operating on flat levels. The hybridization of meta-heuristic algorithms provides very powerful search methods for the FPGA placement. By applying different search techniques on each hierarchical level, high-quality placement solutions can be obtained in a short time. Furthermore, current techniques may be modified easily to solve more complex dynamic placement reconfigurable architecture environments. The parallelization of the placement algorithms is yet an interesting area worth investigation. By making use of the power of distributed/parallelled processors, a large job can be subdivided into smaller parts that can be executed in parallel which can greatly shorten the overall execution time.

Appendix A

Acronym Glossary

ASIC: Application-Specific Integrated Circuit

CAD: Computer-Aided Design

CSS: Cluster Seed Search

FPGA: Field-Programmable Gate Array

GA: Genetic Algorithm

GRASP: Greedily Randomized Adaptive Search Procedure

INLS: Immediate Neighbourhood Local Search

MCNC: Microelectronic Corporation of North Carolina

SLS: Simple Local Search

SSA: Simple Simulated Annealing

TS: Tabu Search

VLSI: Very Large Scale Integration

VPR: Versatile Placement and Routing tool for FPGAs

Appendix B

Routing Results

The quality of placement solutions greatly impacts the following routing process that is also crucial to FPGA design. To investigate the effect of placement quality on the routing process, VPR routing package is used to route placement solutions obtained by iterative based techniques. Table B.1 shows the wirelength of solutions and the track number required indicating successful routing.

Benchmark Circuit	**SLS**		**INLS**		**TABU**		**SSA**		**VPR**	
	W.length	N.track	W.length	N.track	W.length	N.track	W.length	N.track	W.length	N.track
Small-Size(e64)	4105	12	4063	11	3108	9	3071	9	2858	8
Medium-Size(teng)	17354	13	16851	12	11329	10	10818	9	9394	7
Large-Size(frsic)	112539	27	106681	25	68817	16	65734	15	51256	13

Table B.1: Routing results

Bibliography

[Arei01] S. Areibi, M. Thompson, and A. Vanneli, "A clustering untility based approach for asic design," In *14th Annual IEEE International ASIC/SOC Conference, pp.12-15*, September, 2001.

[Arei93] S. Areibi and A. Vannelli, "Circuit partitioning using a tabu search approach," In *IEEE International Symposium on Circuits and Systems*, pp. 1643–1646, 1993.

[Argu97] M.F Arguello, J.F Bard, and G.Yu, "A grasp for aircraft routing in response to groundings and delays," pp. 211–228, 1997.

[Arts03] E. Arts and J. K. Lenstra, *Local Search in Combinatorial Optimization*, Princeton University Press, 2003.

[Bao04] Xiaojun Bao and Shawki Areibi, "Constructive and local search techniques for fpga placement," In *IEEE, CCECE Conference*, May 2004.

[Bard89] J.F. Bard and T.A. Feo, "Operations sequencing in discrete parts manufacturing," pp. 249–255, 1989.

[Betz00] V. Betz and J. Rose, "Vpr and t-vpack: Versatile packing, placment and routing for fpgas package ver 4.30," http://www.eecg.toronto.edu/ vaughn/chanllenge/challenge.html, 2000.

[Betz97] Vaugh Betz and Jonathan Rose, "**VPR**: A new packing, placement and routing tool for fpga research," In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, 1997.

[Betz99] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, *Architecture and CAD for Deep-Submicon FPGAs*, Kluwer Academic Publishers, 1999.

[Bina98] S. Binato, G.C Oliveira, and J.L. Araujo, "A greedy randomized adaptive search procedure for transmission expansion planning," 1998.

[Blum03] Christian Blum and Andrea Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," In *ACM Computing Survey*, pp. 268–308, September 2003.

[Brow92] S. Brown, R. Francis, J. Rose, and Z. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[Cart86] W. Carter, "A user programmable reconfigurable gate array," In *Proceedings of Custom Intergrated Circuits Conference, pp. 233-235*, May. 1986.

[Chan03] Pak K. Chan and Martine D. F. Schlag, "Placement: Parallel placement for field-programmable gate arrays," In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, February 2003.

[Chan96] Y. W Chang, D. Wong, and C.Wong, "Universal switch modules for fpga design," In *ACM transactions on Design Automation of Electronic Systems, vol, 1, pp, 80-101*, January, 1996.

[Chen92] K.C. Chen, J. Cong, Y. Ding, A. Kahng, and P. Trajmar, "Dag-map: Graph-based fpga technology mapping for delay optimization," September, 1992.

[Chen93] C. Cheng, "A accurate and efficient placement routability modeling," In *ICCAD*, pp. 422–425, 1993.

[Cong94] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in loopup-table based fpga designs," January, 1994.

[Du03] Peng Du, "A fast heuristic technique for fpga placement based on multilevel clustering," In *M.A.Sc. Thesis, University of Guelph, Department of Computer & Information Science*, 2003.

[Du04] Peng Du, "Partitioning based fpga placement," In *Technical Report, Univerisity of Guelph*, April 2004.

[Feo89] T.A. Feo and M.G.C Resende, "A probabilistic heuristic for a computationally difficult set covering problem," pp. 67–71, 1989.

[Feo95] T.A. Feo and M.G.C Resende, "Greedy randomized adaptive search procedures," pp. 109–133, 1995.

[Fidu82] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," In *Proceedings of the nineteenth design automation conference*, January 1982.

[Fisk71] Fisk, C. M., Mattheyses, and R.M., "Accel: Automated circuit card etching layout," In *Proceedings of IEEE*, November, 1971.

[Fran91a] R. Francis, J. Rose, and Z. Vranesic, "Technology mapping lookup table-based fpgas for performance," 1991.

[Fran91b] R. Francis, J. Rose, and Z.Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based fpgas," 1991.

[Gare79] M.R. Garey and D.S. Johnson, "A guide to the theory of np-completeness," In *Computers and Intractability*, 1979.

[Glov86] F. Glover, "Future paths for integer programmingand links to artificial intelligence," In *Computers and Operations Research*, pp. 1276–1290, 1986.

[H04] Sun H and S. Areibi, "Global routing for vlsi standard cells," In *Canadian Conference on Electrical and Computer Engineering (CCECE 2004)*, May 13 2004.

[Holl75] J.H. Holland, "Adaptation in natural and artifiical systems," In *Univeristy fo Michigan Press*, 1975.

[Huan86] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, "An efficient general cooling schedule for simulated annealing," pp. 381–384, 1986.

[Huan97] D. Huang and A. Kahng, "Partitioning-based standard-cell global placaement with an exact objective," In *Proceedings of ACM international symposium on Physical Design*, April 1997.

[Kang03] Sung-Mo Kang and Yusuf Leblebici, *CMOS Digital Intergrated Circuits Analysis and Design*, Mc Graw Hill, 2003.

[Kang83] S. Kang, "Linear ordering and application to placement," pp. 457–464, 1983.

[Karg86] P.G. Karger and B.T. Preas, "Automatic placement: A review of current techniques," pp. 622–629, Las Vegas, Nevada, 1986.

[Kern70a] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," In *Bell System Technical Journal*, Feburary, 1970.

[Kern70b] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," In *The Bell Systerm Technical Journal*, pp. 291–307, 1970.

[Kirk83a] S. Kirkpatrick, C. Gelant, and M. Vecchi, "Optimization by simulated annealing," In *Science*, May, 1983.

[Kirk83b] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," In *Science*, May, 1983.

[Lagu94] M. Laguna, T.A. Feo, and H.C. Elrod, "A greedy randomized adaptive search procedure for the two-partition problem," pp. 677–687, 1994.

[Lagu98] M. Laguna and R. Marti, "Grasp and path relinking for 2-layer straight line crossing minimization," 1998.

[Lam88] J. Lam, J. Delosme, and C. Sechen, "Performance of a new annealing schedule," In *IEEE transactions on Computer-Aided*, March 1988.

[Liu98] Huiqun Liu, Kai Zhu, and D. F. Wong, "Circuit partitioning with complex resource constraints in fpgas," In *Proceedings of the 1998 ACM sixth international symposium on Field programmable gate arrays*, March 1998.

[Maid03] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan, "Compilation techniques for reconfigurable devices: Fast timing-driven partitioning-based placement for island style fpgas," In *Proceedings of the 40th conference on Design automation*, June 2003.

[Mall89] S. Mallela and L.K. Grover, "Clustering based simulated annealing for standard cell placement," 1989.

[Mazu99] Pinaki Mazumber and Elizabeth M. Rudnick, *Genetic Algorithms for VLSI Design, Layout and Test Automation*, Prentice Hall PTR, 1999.

[Mulp01] Chandra Mulpuri and Scott Hauck, "Runtime and quality tradeoffs in fpga placement and routing," In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, February 2001.

[Nag95] Sudip Nag and Rob Rutenbar, "Performance-driven simultaneous place and route for island-style fpgas," In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, December 1995.

[Osma96] I.H. Osman and G. Laporte, "Metaheuristics: A bibliography," In *Ann. Oper. Res*, pp. 513–623, 1996.

[Part01] G. Parthasarathy, M. Marek-Sadowska, A. Mukherjee, and A. Singh, "Interconnect complexity-aware fpga placement using rent's rule," In *Proc. of System Level Interconnect Prediction*, March, 2001.

[Pasi98] E.L Pasiliao, "A greedy randomized adaptive search procedure for the multi-criteria radio link frequency assignment problem," Department of ISE, Univeristy of Florida, 1998.

[Prea88] Bryan Preas and michael Lorenzetti, "Physical design automation of vlsi systems," The Benjamin/Cummings Publishing Company, Inc, 1988.

[Rese97] L.I.P. Resende and M.G.C Resende, "A grasp for frame felay pvc routing," 1997.

[Rese98] M.G.C Resende, "Computing approximate solutions of the maximum covering problem using grasp," pp. 161–171, 1998.

[Rose91] J. Rose and S. Brown, "Flexibility of interconnection stuctures for field-programmable gate arrays," In *JSSC, pp, 475-478*, March, 1991.

[Sank99] Yaska Sankar and Jonathan Rose, "Trading quality for compile time: ultra-fast placement for fpgas," In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, February 1999.

[Shah91] K. Shahookar and P. Mazumder, "Vlsi cell placement techniques," In *ACM Computing Surveys (CSUR)*, June 1991.

[Sun95] W. Sun and C. Sechen, "Efficient and effective placement for very large circuits," In *IEEE transactions on Computer-Aided Design Automation Conference, vol. 14 No.3, PP. 349-359*, March 1995.

[Swar90] W. Swarz and C. Sechen, "New algorithms for the placement and routing of macro cells," In *ICCAD*, 1990.

[Tess02] Russell Tessier, "Fast placement approaches for fpgas," 2002.

[Trim94] Stephen M. Trimberger, *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers, 1994.

[Tsen92] B. Tseng, J. Rose, and S. Brown, "Using architectural and cad interaction to improve fpga routing architecture," In *ACM Workshop on FPGAs, pp 3-8*, 1992.

[Yang02] Z. Yang and S. Areibi, "Global placement for vlsi standard cell design," pp. 243–247, San Diego California, November 2002.

[Yang91] S. Yang, "Logic synthesis and optimization benchmarks," In *Technical Report, Microelectronics Center of North Carolina*, 1991.