A Fast Heuristic Technique for FPGA Placement based on Multilevel Clustering

by

Peng Du

A thesis

presented to the University of Guelph

in fulfilment of the

thesis requirement for the degree of

Masters of Science

in

Department of Computing & Information Science

Guelph, Ontario, Canada, 2004

I hereby declare that I am the sole author of this thesis.

I authorize the University of Guelph to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Guelph to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Guelph requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Field-Programmable Gate Arrays (FPGAs) are semiconductor chips that can realize most digital circuits on site by specifying programmable logic and their interconnections. The use of FPGAs has grown almost exponentially because they dramatically reduce design turn-around time and start-up cost for electronic products compared with traditional Application-Specific Integrated Circuits (ASICs).

A set of Computer-Aided Design (CAD) tools is required to *compile* hardware description into bitstream files that are used to configure the target FPGA to implement the desired circuits. Currently, the compile time, which is dominated by *placement* and *routing* time can easily take hours or even days to complete for large (8-million gate) FPGAs. With 40-million gate FPGAs on the horizon, these prohibitively long compile times may nullify the time-to-market advantage of FPGAs.

This thesis presents two novel placement heuristics that significantly reduce the amount of computation time required to achieve high-quality placements, compared with VPR [Betz99] [Betz97b], which is considered to be a state-of-the-art FPGA placement and routing tool. The first algorithm is an enhancement of simulated-annealing and attempts to solve the placement problem top-down by considering all blocks. The second algorithm is a hierarchical approach, which is based on a two-step procedure that first proceeds bottom-up and then top-down. The bottom-up technique uses *clustering*, and involves grouping of highly connected blocks into clusters and then combines clusters into larger clusters. The goal of the top-down method is to determine the locations for all the clusters, and eventually the locations of all blocks within those clusters. The overall effect is to reduce the number of entities that need to be considered at each level, making time-consuming methods, e.g. simulated-annealing, feasible for large problems.

Acknowledgements

My sincere thanks go to my advisor, Prof. Dilip Banerji. Without his help, this work would never have been possible. I have enjoyed a wonderful research experience under his supervision, who has gone beyond the duties of a supervisor to act as a mentor as well as a supporter.

I would also like to thank my co-advisor Prof. Shawki M. Areibi for offering a wealth of support and guidance. I am indebted to him for setting high technical standards for his students.

I also appreciate the help of my co-advisor Prof. Gary Grewal. His infectious zeal and unwavering confidence has not only helped me to drive through this research but would also influence my future work.

I am also grateful to Prof. William Gardner for advising and revising this thesis with great meticulosity.

I have also greatly benefitted from the wisdom of my colleagues: Amit Khosla, Yu Zeng (Yuki), Hua Liang, W.D.Keerthi Perera, Zhen Yang (Judy), Xiaojun Bao, Brian Zhou, Guangfa Luo and others. I am particularly obligated to Amit Khosla and Yu Zeng (Yuki) for advising and revising this work.

I would like to thank my friends Yu Zeng (Yuki), Jun Ren, Xiangtong An, Wei Zhou, Chang, Wei Zhou, and Dongliang Wang who gave me unconditional support and played badminton together to keep me physically as well as spiritually fit.

To

my parents

whose love and encouragement helped accomplish this thesis.

This thesis is dedicated to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Very Large Scale Integration (VLSI)[1] technology has opened the doors to the implementation of extremely complicated digital circuits at a relatively low cost. It is now possible to manufacture chips with hundreds of millions of transistors, as exemplified by the most powerful microprocessors.

A VLSI design includes both *logic* design and *physical* design of a circuit. Logic design implements a circuit using gates, flip-flops, and other logic components while remaining technology independent. On the other hand, physical design which ultimately implements the logic design is very much technology dependent.

Currently, there are several different technologies that can implement a VLSI design. Figure 1.1 illustrates the most common commercially available physical design technologies.

This is the most basic and time-consuming physical design method. All parts of such a design are carefully tailored by hand and Computer-Aided Design (CAD)

[1] A glossary of all acronyms used in this thesis is provided in Appendix C.

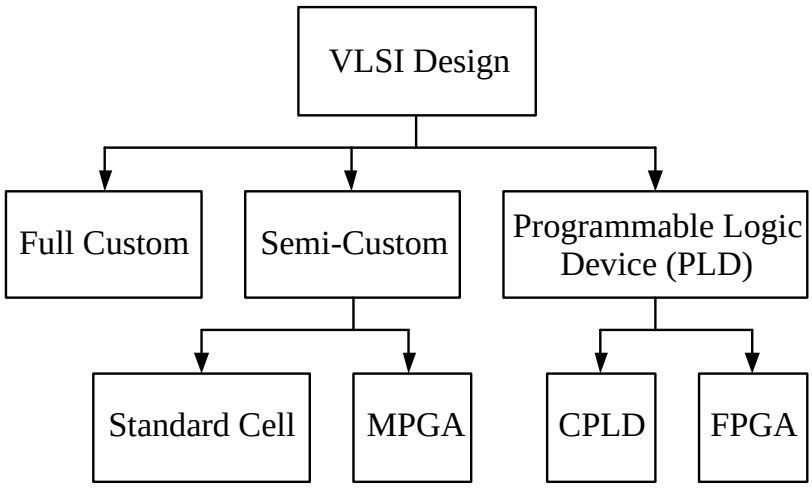

Figure 1.1: Common VLSI physical implementations

tools to meet specific requirements. Since the design process is completely under the control of the designers, very compact and efficient chips can be created.

Semi-Custom:

This approach uses a mix of designer-created as well as pre-designed components. Therefore, it provides an easier way of designing and manufacturing Application-Specific Integrated Circuits (ASICs). Standard Cell and Mask-Programmed Gate arrays (MPGAs) are two major technologies included in this category.

In the standard cell technology, a logic design is first mapped to "standard" cells, which are included in libraries or customer pre-designed. Typically, such a library includes commonly used functional blocks such as full adders and multiplexers. In their layout geometry, these standard cells are restricted to have the same height, while their width can vary depending on the design. Once the circuit is mapped, these cells are arranged in horizontal rows within the chip boundary. Spaces between the rows, called channels, are used to implement the interconnec-

tions between the cells. Standard cell methodology is very popular in VLSI design, since it facilitates the physical design by providing existing well-tuned elements.

MPGA devices consist of an array of uncommitted elements that can be interconnected according to a user's specification. The most popular MPGAs consist of rows of transistors that can be interconnected to implement a desired circuit. User-specified connections are available both within the rows (to implement basic logic gates) and between the rows (to connect the basic gates together). In an MPGA, all the mask layers are pre-defined by the manufacturer, except those that specify the final metal layers which are customized to connect the transistors in the array. MPGAs still have large non-recurring engineering (NRE) cost because of the need to fabricate the metal mask layers and *pack* chips in foundries.

Programmable Logic Devices (PLDs):

The previous two approaches require extensive manufacturing efforts, taking a long time from design to final product. This high overhead results in a high cost for each unit unless large volumes are produced. In industry, the time-to-market should be made as short as possible, so it is essential to reduce the development and production time. Furthermore, the possible financial and technical risks incurred in the development of a new product could be prohibitively high for small companies.

Programmable Logic Devices (PLDs) have emerged as the ultimate solution, in which the final logic structures can be directly configured by the end user, without the use of any integrated circuit fabrication facility. These devices can be mapped "instantly" by a set of CAD tools, leading to low start-up cost, quick time-to-market, no NRE cost, and easier modification. In addition, most PLDs can be re-programmed many times which not only enables a fast recovery from design

errors but also makes it easier to add new features to systems that have been manufactured. However, the benefits provided by PLDs come at a price. They are slower and require more silicon area than their ASIC counterparts [Brow92]. These disadvantages are mainly due to the fact that the logic in PLDs is configured and connected via programmable switches, while for other technologies, the logic is directly configured and connected with metal wires. Regardless, PLDs have revolutionized the methodology to design and build digital hardware today.

Sophisticated PLDs can be largely divided into two types: Complex Programmable Logic Devices (CPLDs) and Field-Programmable Gate Arrays (FPGAs). A CPLD usually is an arrangement of multiple Simple Programmable Logic Devices (SPLDs). The latter mainly consist of two levels of programmable logic−an AND plane and an OR plane. FPGAs have a more general structure that allows very high logic capacity and flexibility. Whereas CPLDs feature logic resources with a wide number of inputs and outputs (AND and OR planes), FPGAs offer broader logic resources including a higher ratio of flip-flops to logic resources than do CPLDs (details of FPGA architectures are presented in Chapter 2).

Since their commercial introduction in the mid-1980's, FPGAs have gained rapid acceptance and experienced a phenomenal growth in industry. Improvements in VLSI processing technology have upgraded FPGAs from "glue" circuits with a handful of TTL equivalent logic gates to System-On-a-Chip (SOC) with a capacity of a few million effective gates, which continues to sprout embedded cores, such as CPUs, memories, and interfaces. With this abundance of logic resources available, many applications traditionally held by other VLSI technologies have become feasible for implementation on FPGAs. The current FPGAs allow almost any application to be instantly realized or reconfigured, such as device controllers, com-

munication decoders, filters, processors and so on. Another major use of FPGAs is prototyping of designs. The low cost of implementation and short time needed to physically realize a given design provide enormous advantages over traditional approaches which have to be completed in foundries.

1.1 Problem Definition and Motivation

In addition to logic circuit design, implementation of a design on an FPGA requires a set of integrated CAD tools. These tools transform or *compile* a design from its hardware description or schematic into a bitstream that is downloaded to the target device in order to configure it. The work in this thesis focuses on handling a part of this highly complex task of design implemetation.

As noted earlier, one key advantage of FPGAs over full custom and semi-custom devices is that they provide relatively quick implementation from concept to physical realization. However, with the recent announcement of some FPGAs that contain the equivalent of 40-million gates, new challenges emerge. One of the challenges is the compile time for designs, which is dominated by *placement* and *routing* time. While current CAD algorithms provide high-quality solutions, they require great amounts of CPU time. In fact, the compile time seems to be growing more rapidly than the available computation power. For many complex circuits, this compile time can be in the order of tens of CPU hours.

Such a long turn-around time adversely impacts the use of FPGAs by hardware designers. This provides a compelling motivation to explore new methods for fast compilation of designs. As we move to sub-micron designs, circuit delay, as well as power dissipation are dominated by interconnections between logic elements.

This problem is specially severe for FPGAs which use programmable switches and connections for implementing a *netlist*. Poor solutions, even derived quickly, are often not acceptable in industry. There is a great need for CAD tools that execute in a reasonable amount of CPU time, while still generating high-quality solutions. In this thesis, we focus on the placement phase of the FPGA-based design process. We present two adaptive placement algorithms, aiming to obtain the same quality[2] as the best placement tool while minimizing the runtime as much as possible.

1.2 Proposed Research Approach and Contributions

Our research is concerned with finding a fast algorithm for placement that produces high-quality results. We measure the performance of our algorithm against an existing package, Versatile Placement and Routing tool for FPGAs (VPR)[3] [Betz99] [Betz97b], which is an open source program. VPR provides high-quality placement and routing solutions over a large suite of benchmark circuits in a reasonable amount of CPU time [Mulp01]. We make a fair evaluation of how well our tools perform with respect to both runtime and placement quality of VPlace, by running both algorithms on the same computation platform with the same suite of benchmark circuits and the same FPGA architecture.

VPlace is based on the simulated-annealing algorithm that has found wide usage both in academia and industry. While obtaining the best placement quality, it suffers from relatively long runtimes [Mulp01]. This stochastic heuristic can cause

[2]In this thesis, placement quality is measured in terms of total wirelength.

[3]The placement part of VPR is referred to as VPlace.

the search procedure to spend a disproportionally large amount of time to examine poor solutions. What is desired is some greedy search heuristic that can reduce the number of solutions to be evaluated in order to accelerate the convergence of a search.

Our "flat" placement algorithm takes advantage of some of the best features from VPlace. In addition, by utilizing a memory structure, which records a short period of search history, it behaves in a more greedy fashion than the simulated-annealing algorithm. This new heuristic results in tremendous time saving compared to VPlace, while reaching equal placement quality.

We also investigate a hierarchical approach to placement by utilizing multi-level clustering and de-clustering. Very little related work has previously been done in this area of FPGA placement, despite the success of clustering and de-clustering when applied to standard cell placement, such as in Timberwolf95 [Sun95]. Clustering of nodes (standard cells or logic blocks) into supernodes reduces the number of entities necessary to be placed, compared to the flat circuit. The reduced search space makes the use of time-consuming algorithms, such as simulated-annealing, feasible for large problems. Again, this approach achieves great speedup in CPU time with a small loss in solution quality.

1.2.1 Contributions

The work done for this thesis makes the following contributions:

1. A novel adaptive greedy iterative heuristic, which uses a short term memory to speed up the convergence of placement.

2. A hierarchical approach, which makes use of a combination of suitable itera-

tive heuristics and clustering scheme to achieve high-quality placement within a short amount of time. Little work has been done in the domain of FPGA hierarchical placement, although this approach has been applied heavily to the circuit partitioning problem and standard cell placement.

3. A novel method to determine the proper start parameters of simulated-annealing algorithm automatically, when given a good initial placement.

These ideas can also be used in other similar problems, e.g. *floor planning*, or even general combinatorial optimization problems. Most importantly, this work has greatly enhanced an understanding of FPGA placement, especially for large circuits.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 contains an introduction to FPGA architectures, generic CAD procedure for FPGA-based designs, and definitions of basic terms. We also discuss some previous work done in FPGA placement, as well as prior work done in the related area of hierarchical clustering. Chapter 3 describes our proposed placement algorithm, and presents experimental results that show how our algorithm leads to a significant speedup in CPU time with no loss in the placement quality, as compared to VPlace. Chapter 4 focuses on the hierarchical approach where our placement algorithm described in Chapter 3 still plays an important role. With a novel adaptive method to automatically adjust proper starting parameters, alogrithms employed at each level of hierarchy can be linked together smoothly, which is crucial for any hierarchical approach.

Finally, Chapter 5 highlights some key conclusions and contributions of our work and proposes possible directions for future research.

Chapter 2

Background and Previous Work

This chapter discusses the FPGA physical design procedure and provides details of FPGA placement problem, followed by a brief description of some of the relevant previous work done in the placement domain.

2.1 FPGA Architectures

There is a wide variety of architectures for FPGAs from different vendors including Xilinx, Altera, Actel, Lucent, QuickLogic and so on. Although the exact structure of these FPGAs varies from each other, all FPGAs consist of three fundamental components:

1. Logic blocks that are capable of implementing multiple logic functions.
2. I/O blocks or I/O pads for communication with the outside world.
3. Fixed as well as programmable routing resources used to realize all required interconnections between the blocks.

What comprises the logic blocks and how the routing resources are organized define a specific FPGA architecture.

The complexity of logic blocks can be classified into two types: *coarse-grained* and *fine-grained*. A coarse-grained logic block contains substantial logic structures, often has a clustered architecture including a few look-up tables (LUTs) and flip-flops or a few Programmable Logic Device (PLD) modules. The more complex the logic block, the more functions it can implement. The 4-input look-up table (4-LUT) is most widely employed in these architectures [Brow92]. In fine-grained architectures, there is a large number of relatively simple logic blocks, which usually contain only a few basic gates and multiplexers.

In terms of logic block and routing resource layout, FPGAs can be classified into one of the following four categories [Brow92].

Row-based:

Logic blocks are arranged in rows, and routing resources consist of horizontal wire segments of various lengths, which are seperated by routing switches. Also, a few vertical wire segments exist for routing between rows. A counterpart in ASIC design is the standard cell architecture.

Hierarchical:

Logic blocks and routing resources are deployed in a hierarchical mode or macrocell mode. A two-dimensional array of programmable logic blocks is used to implement multilevel logic functions. Both intra-level and inter-level interconnections are provided.

Sea-of-Gates:

Logic blocks, usually fine-grained, are organized as a symmetrical array. Routing resources are overlaid on top of the blocks. This structure resembles the Sea-Of-Gates architecture used in MPGAs.

Island Style:

Many commercially available FPGAs employ this architecture, in which logic blocks, referred to as Configurable Logic Blocks (CLBs), are arranged as a symmetrical array. Routing tracks have *Manhattan* geometry, that is, they are either horizontal or vertical. The CLBs, typically coarse-grained[1], are separated by programmable routing switches.

Figure 2.1 shows a generic model of the FPGA architecture assumed in this thesis [Brow92]. This model is very similar to Xilinx[2] architectures. The model is general enough to be suitable for other layout styles, with appropriate modifications. Many researchers and CAD tools employ this model as their prototype [Betz99] [Betz97b] [Sank99].

Figure 2.2 shows the details of the routing structure, which consists of three components: *connection box*, *switch box*, and *channel segment*. A connection box is used to connect a CLB to the routing channels via programmable connections. The pins of each CLB pass uninterrupted through the connection box and have the option of "fusing" to any channel segments. The switch box is a switch matrix that is used to connect wires in one channel segment to

[1]The exact internal architecture of CLBs can vary from vendor to vendor. We provide an example in Section 3.4.1, which is the FPGA architecture used in our experiments.

[2]Xilinx is the inventor and currently one of the leading vendors of FPGA chips.

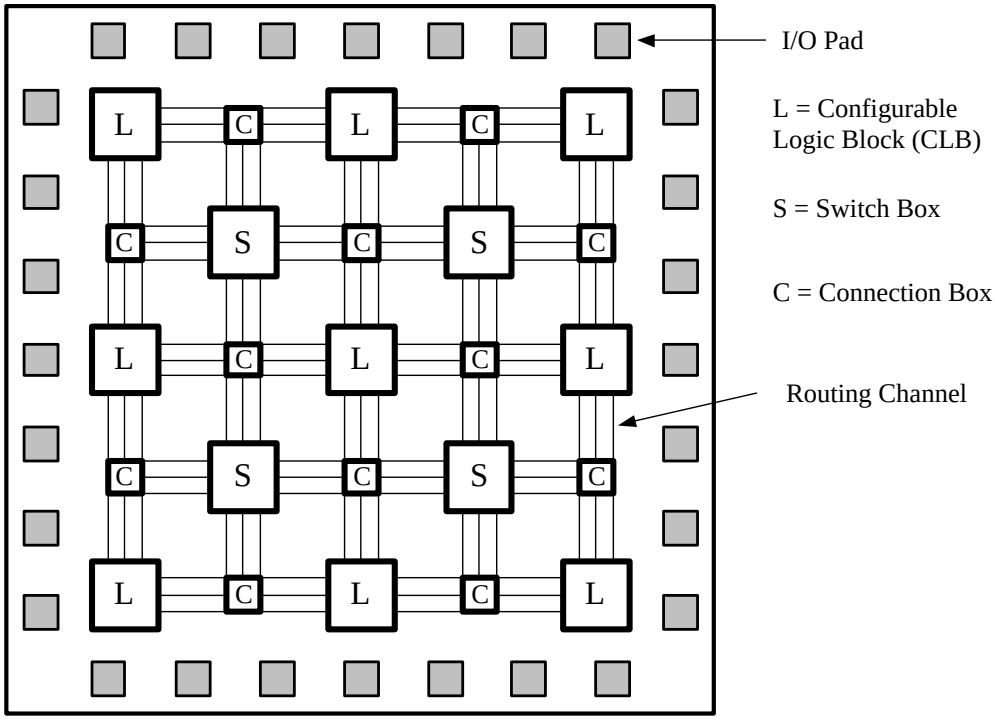

Figure 2.1: Island style FPGA architecture [Brow92]

those in another. Depending on the topology [Chan96], each wiring segment on one side of a switch box may be connected to some or all of the wiring segments on the other three sides. This flexible routing structure enables every CLB to have connections with any other CLB or I/O pad, depending on the number of tracks in the routing channels.

2.2 CAD for FPGA Design

Implementing a design on an FPGA involves a sequence of steps, each assisted by a CAD tool. A typical design procedure employed by most commercial FPGA tools is shown in Figure 2.3. A brief description of the steps follows.

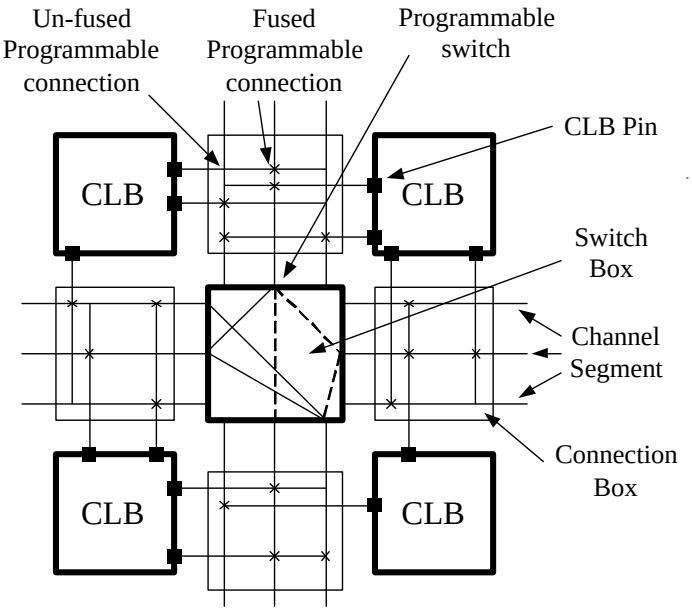

Figure 2.2: Detailed FPGA routing architecture

Design Entry:

The description of a logic circuit to be implemented can be specified by using a hardware description language (HDL) such as VHDL or Verilog. Alternative ways of specifying a circuit are to use a state machine language or a schematic capture tool.

Synthesis & Optimization:

If the target design is specified in terms of behavioral or logical description at the design entry level, it is *synthesized* into a logic level design first. If entered as a schematic, then the logic design already exists. In either case, the logic design is passed through an optimizer to remove redundant logic, while maintaining its functionality.

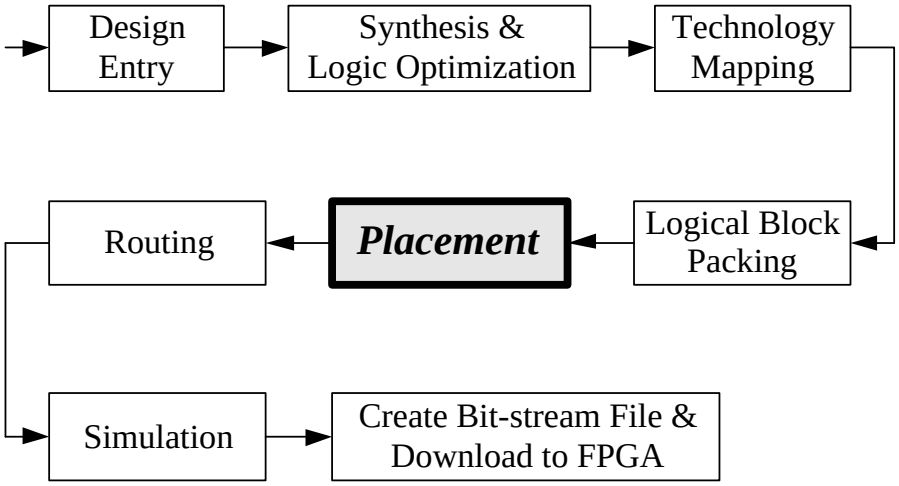

Figure 2.3: Typical FPGA CAD flow

Technology Mapping:

Once the design is optimized, a *technology-dependent* mapping [Cong94] tool is used to transform the basic logic functions into k-LUT-sized groups, where k is based on the specific FPGA architecture on which the design is to be implemented.

Logic Block Packing:

In clustered FPGA architectures, a logic block consists of more than one Basic Logic Element (BLE) [Betz99] [Betz97b]. A BLE is the most basic brick of an FPGA, which usually includes a k-LUT and a latch. The major objectives of the packing process are combining k-LUTs and latches into BLEs and grouping the BLEs into CLBs. Packing algorithms must take advantage of the faster internal routing resources by balancing constraints including the maximum number of inputs and BLEs per CLB, while attempting to minimize the number of signal connections between CLBs [Betz99] [Betz97b].

Placement:

When the circuit has been reduced to a list of blocks and a netlist describing the connectivity between these blocks, a placement tool [Sech85] [Sun95] [Betz99] [Betz97b] [Sank99] is used to determine the physical location of each block within the target FPGA.

Routing:

FPGA routing [McMu88] [Brow96] is the process of assigning specific routing resources to each net to realize the required connections among logic blocks. Routing a net corresponds to finding a path between the start node (source) and the end nodes (sink) in a graph. Because a connection represents a physical path that a signal will occupy, the routing resources assigned to any net are exclusive. A design is acceptable and workable only if 100 percent routing can be achieved within the target FPGA.

Simulation:

After routing, the implemented design is simulated to ensure its functionality and verify the timing constraints. Design errors can be found and corrected at this stage.

Create Bitstream File & Download to FPGA:

Once all the previous steps are completed successfully, a CAD tool is used to create bitstream files that are downloaded to the target FPGA to implement the logic and interconncetion configuration. After this stage, the FPGA is ready for use.

FPGA placement has been proven to be an NP-hard problem [Gare79]. If there are N blocks that need to be placed, the number of different possible solution combinations for the placement problem is $N!$. It is a crucial phase and one of the most intricate problems in the design process. The quality of placement has a great impact on the final performance of the design.

This thesis presents two new methods for placement based on the island style model. The effectiveness of these methods is demonstrated by applying them to a set of MCNC [Yang91] benchmark circuits. CPU time and solution quality are compared to an existing high-quality placement tool, VPlace[Mulp01] [Betz97a], which is a part of VPR package [Betz99] [Betz97b].

2.3 The FPGA Placement Process

FPGA placement usually begins with a netlist of logic blocks, which includes CLBs and I/O pads, and their interconnections. The result of placement is the physical assignment of all blocks and pads on the target FPGA that minimizes one or more specific objective cost functions.

2.3.1 FPGA Placement Objectives

The most basic objective for FPGA placement is to minimize the routing cost, which is the total wirelength required to complete the routing. Routing cost is used because reducing it actually reduces a number of associated design parameters. By reducing the routing length, the routing resources required by all interconnections are also reduced. This results in an increase in circuit speed due to the reduction in connection capacitance and resistance. Power consumption, which is another

very important parameter to measure the quality of an FPGA implementation, is reduced too. A placement intending to minimize the routing cost is also referred to as *wirelength-driven* placement. There are other objective terms that can be added to the original cost function to directly optimize the various design goals [Sait95]. For example, placement can be done to minimize the length of a critical path to meet timing constraints, referred to as *timing-driven* placement [Marq00] [Swar95], or to balance the wire density across the FPGA device, referred to as *routability-driven* placement [Part01].

Most commercially available FPGA placement tools are timing-driven, which has been proven in the work of Marquardt et al. [Marq00] and Swartz et al. [Swar95]. This approach is more efficient in improving the speed of a FPGA-based circuit than wirelength-driven placement. In this thesis, however, we use the wirelength-driven approach as the first step in our FPGA placement research.

We provide two figures in Appendix B, borrowed from VPR package, to have a visual illustration of the FPGA placement problem.

2.3.2 Half-perimeter Wirelength Model and Bounding Box Cost

In the placement phase, it is computationally too expensive to determine the exact configurations of routing resources to realize physical connections for CLBs and I/O pads. It is actually another NP-hard problem. For this reason, the routing cost is approximated during placement. The speed and accuracy of estimation have a significant effect on the performance of a placement tool. There are various cost approximation techniques available [Sait95], such as *Steiner tree*, *minimum*

spanning tree, and *half-perimeter wirelength* model.

A Steiner tree [Arei01b] is the shortest route to connect a set of terminals. It provides the most accurate means for wirelength estimation. In this method, a wire can branch from any point available along its length to connect to other terminals in the net. Unfortunately, since the problem of finding the Steiner tree itself has been proven to be NP-hard, this technique is normally not used due to its heavy computational requirements.

Unlike the Steiner tree, in a minimum spanning tree, branches are only allowed at the terminals. For an n-terminal net, the tree can be constructed by determining the distances between all possible pairs of terminals, and connecting the smallest $(n-1)$ edges that do not form a cycle. A minimum spanning tree can be found in polynomial time by Kruskal's algorithm [Krus56] or Prim's algorithm [Prim57].

Half-perimeter Wirelength(HPWL) model is the most widely used method to estimate the wirelength of a net [Shah91]. The wirelength is approximated by half the perimeter of the smallest bounding rectangle that encloses all terminals in the net, as shown in Figure 2.4.

In a Manhattan routing structure, the HPWL of a net approximates the length of a Steiner tree, which is the lowest bound on the final routing cost on a net. Given a block b with coordinates (x_b, y_b), the half-perimeter of net i is calculated as follows:

$$HPWL_i = (MAX_{b\in i}\{x_b\} - MIN_{b\in i}\{x_b\} + 1) + (MAX_{b\in i}\{y_b\} - MIN_{b\in i}\{y_b\} + 1) \tag{2.1}$$

For a net with two or three terminals, the routing cost obtained by HPWL model is exactly the same as that obtained by Steiner tree model. When there are more

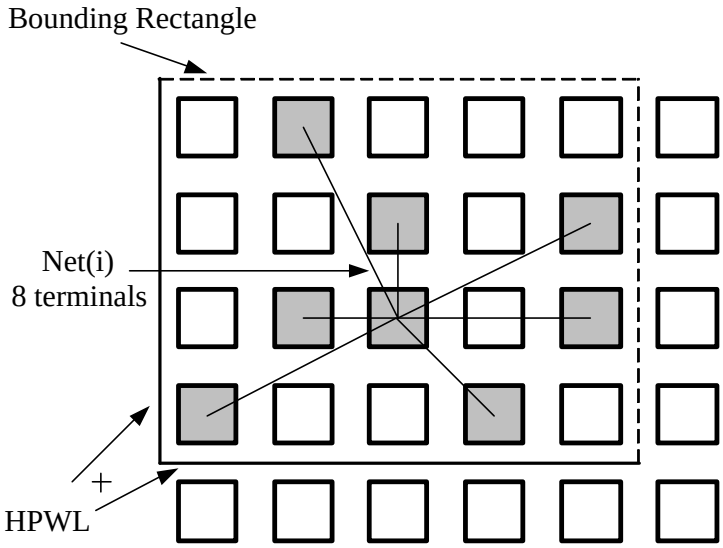

Figure 2.4: Half-perimeter wirelength model

than three terminals in a net, a $q(i)$ factor [Chen94] is introduced to compensate for the fact that the HPWL model underestimates the wirelength necessary to connect all blocks. The value of $q(i)$ depends on the number of terminals in net i. $q(i)$ is 1 for nets with 3 or fewer terminals, and slowly increases to 2.79 for nets with 50 terminals. For exceptionally heavy fanout nets that have more than 50 terminals, the value of $q(i)$ linearly increases at the rate of:

$$q(i) = 2.7933 + 0.02616 * (TerminalNumber - 50) \quad \text{[Betz00]} \tag{2.2}$$

Therefore, the final cost function, called the *bounding box cost*, takes the following form:

$$Cost_{bounding\_box} = \sum_{i=1}^{N_{nets}} q(i) * HPWL_i \tag{2.3}$$

Consequently, the placement problem pertaining to this thesis is equivalent to the

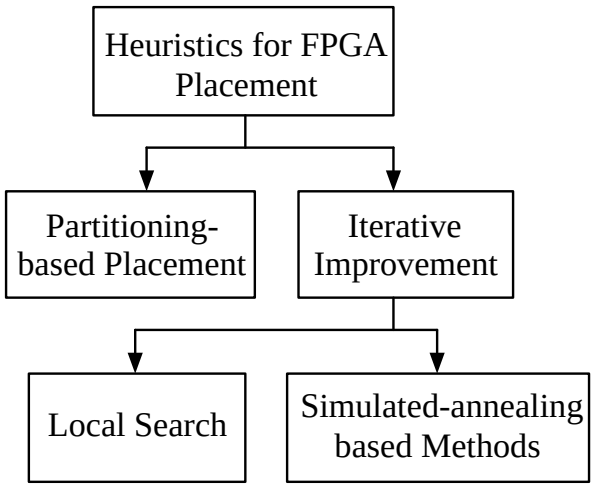

Figure 2.5: Taxonomy of approaches to FPGA placement

problem of minimizing the *bounding box cost*.

2.4 Heuristic Approaches for FPGA Placement

FPGA placement is a hard combinatorial optimization problem, and no polynomial-time algorithm is known that can produce an exact solution to the problem [Shah91]. Except for very small circuits, exact solution methods such as exhaustive enumeration are of little practical use. Approximation methods, also referred to as heuristics, provide sub-optimal solutions but can be executed in a relatively short amount of runtime. Almost every commercially available FPGA placement tool is of this type, since design time is of great importance in industry. In recent years, many heuristic techniques have been developed in an attempt to obtain acceptable solutions in a reasonable amount of runtime. Figure 2.5 shows a taxonomy of the most common approaches to solve this problem.

Historically, FPGA placement methods have been divided into two classes:

partitioning-based placement [Klei91] [Ganl95] and *iterative improvement* [Betz99] [Betz97b]. In partitioning-based placement, a circuit is recursively bi-sected, minimizing the number of *cuts* of the nets that connect components between partitions, while leaving highly-connected blocks in one partition. Eventually, the partition size reaches a few blocks to obtain improvement by grouping the highly-connected blocks together.

Iterative improvement methods start with initial (legal) placements and seek improvements by searching for small perturbations to the placements that result in better solutions. In the FPGA placement problem, these perturbations are location swaps (pairwise move) between blocks.

Heuristic methods are discussed in more detail in the following sections. Simulated-annealing based placement algorithms, which belong to iterative improvement methods, have achieved as good or higher quality solutions compared to other methods [Mulp01] [Betz97a]. Consequently in this thesis, we focus on simulated-annealing based algorithms.

There are some popular approaches used for standard cell placement that have not been successfully implemented in the FPGA placement yet, such as constructive methods [Karg86], analytical methods [Alpe97a], tabu search [Song92], and genetic algorithms [Coho86].

2.4.1 Partitioning-based Placement Algorithms

Partitioning-based placement methods, also referred to as *min-cut* methods, are based on graph partitioning algorithms such as Fiduccia-Mattheyses (FM) algorithm [Fidu84], and Kernighan–Lin (KL) algorithm [Kern70]. An FPGA is divided

into two halves, and a circuit partitioning algorithm is applied to determine which logic block goes to which half to minimize the number of cuts in the nets that connect the blocks between partitions, while leaving highly-connected blocks in one partition. This recursive process is repeated until each partition contains only a few blocks to group the highly-connected blocks together in order to decrease placement cost.

A divide-and-conquer strategy is used in these heuristics. By partitioning the problem into sub-parts, a drastic reduction in search space can be achieved. On the whole, these algorithms behave in the *top-down* manner, placing blocks in the general regions where they should belong to.

Partitioning-based placement algorithms are good from a "global" perspective, but they do not directly attempt to minimize wirelength. Therefore, the solutions obtained are sub-optimal in terms of wirelength. However, these kind of algorithms run very fast. They are normally used in conjunction with other search techniques for further quality improvement.

2.4.2 Local Search

Local search methods are the most basic iterative heuristics for finding approximate solutions to large scale combinatorial optimization problems [Aart03]. Starting from an initial (legal) solution, these heuristics achieve improvements by searching the solution *neighbourhood* for a better solution[3].

The basic principle underlying a local search algorithm is that it always moves

[3]A *neighbouring* solution, in FPGA placement, can be any location swap of two blocks from the current placement, which is "close" to the original solution. The solution neighbourhood is the aggregated set of all possible neighbouring solutions.

from the current solution to the next better solution in the neighbourhood in a greedy manner.

Local search algorithms use two common strategies to move to a better solution:

First improving (non-deterministic):

Accept the first improving solution encountered during random evaluation. This strategy is also referred to as stochastic and has the advantage of being fast; the negative side is lack of aggressiveness in finding the best solution in the neighbourhood.

Most improving (deterministic):

Scan and evaluate the whole candidate neighbourhood set and accept the best solution. This is the most traditional selection method that guarantees the best possible move in the neighbourhood. However, the exhaustive comparison process in making a choice can be time-consuming.

Non-deterministic local search methods usually stop after a sufficient number[4] of solutions has been evaluated. Whereas, deterministic local search methods terminate when no further improvement can be achieved.

Local search methods can run fast if well implemented. The weakness of these methods is that they can end up in local minima. When the number of local minima/maxima is large, the probability that a local minimum/maximum is also globally minimum/maximum is small. One possible solution to this problem is to use a multi-start search, which performs a local search many times, starting from different configurations to explore better solutions. However, this search strategy

[4]The value of this number could be adaptive or user pre-defined.

still fails in problems which have a large number of local minima/maxima. This seems to be a generic feature of many of the classical hard combinatorial problems [Aart03] including the placement problem.

For these reasons, local search is normally not used as the primary solution method, but is used to assist other solution techiques [Sank99] [Aart03].

2.4.3 Simulated-annealing Placement

Simulated-annealing [Kirk83] is one well-developed and widely used algorithm for solving combinatorial optimization problems, including those arising in VLSI physical design [Wong88]. As the name suggests, this algorithm mathematically mimics the process of carefully cooling molten metal in order to obtain a good crystalline structure. An ideally annealed crystal should be in the lowest-energy ground state, which corresponds to the globally optimal configuration in a combinatiorial optimization problem.

As one of the methods belonging to *non-deterministic* category, simulated-annealing accepts any randomly encountered solution in the neighbourhood, with a defined probability. A new neighbouring solution is created incrementally from the current solution. If the new cost, derived from a specific objective function, is reduced the new solution is accepted. However, for an inferior cost, the new solution may still be accepted with a probability of $e^{-\Delta C/T}$, where ΔC is the change in cost, and T is analogous to temperature in the metal crystallization process. Parameter T is used to control the acceptance probability of cost-increasing "bad" solutions. The rate of change of T is referred to as *annealing schedule* which has a great influence on the quality of the final solution as well as runtime. Initially, T is set

to a high value such that most inferior solutions can be accepted. As the process goes on, T is gradually decreased (simulating cooling), reducing the probability of accepting poor solutions. In the final stages, T is only a small fraction of its original value and only improving solutions are accepted most of the time. The simulated-annealing algorithm is characterized by its ability of *hill-climbing* to escape local optima in the cost function. These local optima usually cause simple local search algorithms to terminate.

Theoretical analysis [Mitr86] shows that simulated-annealing converges with probability 1 to the globally optimal solution, by imposing certain conditions on the number of iterations evaluated at each T and a certain rule to update the value of T. In addition, it is much easier to add new optimization objectives or constraints to a simulated-annealing based placement algorithm compared to most other algorithms. However, these features provide little information on how to set the proper parameters in a particular implementation. Furthermore, the runtime to find the globally optimal solution can become very large. Consequently, most of the current applications of simulated-annealing employ simple yet effective approaches to obtain some good sub-optima as the final solutions [Sech85] [Betz99] [Betz97b].

2.4.4 VPR Placement Algorithm

The VPR package [Betz00] [Betz99] [Betz97b] is known for leading to high-quality [Betz97a] [Mulp01] results for FPGA packing, placement, and routing. A simulated-annealing based placement tool is built in to VPR, and is called *VPlace*. Figure 2.6 shows the pseudo-code for VPlace.

VPlace adopts some good features from previous work in [Sech85] [Huan86]

```
1.  S = InitPlacement();
2.  T = InitTemperature();
3.  R_limit = InitR_limit();   /* R_limit is set to the whole chip initially */
4.  while(ExitCriterion() == false)    /* outer loop */
5.  {       while(InnerLoopCriterion() == false)    /* inner loop */
6.          {       S_candidate = GenerateMove(S_current, R_limit);
                    /* create a candidate solution from the current one by a */
                    /* random pairwise move within the window R_limit */
7.                  ΔC = Cost(S_candidate) - Cost(S_current);
                    /* calculate the changes in cost */
8.                  r = random(0, 1);
                    /* create a random float number between (0, 1) */
9.                  if(r < e^(-ΔC/T))
10.                         S_current = S_candidate;   /* accept this candidate */
                    /* if ΔC ≤ 0, the move is always been accepted. */
                    /* otherwise, the probability of acceptance is given by */
                    /* e^(-ΔC/T). Initially, T is very high so almost all */
                    /* moves are accepted. T is gradually decreased */
                    /* and eventually makes the acceptance of a "bad" */
                    /* move very difficult. (finally becomes greedy) */
11.         }   /* end of inner loop */
12.         Update(T);   /* T_new = α * T_old */
13.         Update(R_limit);
14. }   /* end of outer loop */
    /* get final placement solution S */
```

Figure 2.6: Pseudo-code for simulated-annealing [Sech85] [Betz97b] [Betz99]

[Swar90] [Lam88]. It also includes a new temperature *update schedule* and a new *exit criterion*, as well as a time-saving *incremental net bounding box update* technique. The initial solution is created by placing CLBs and I/O pads randomly into the physical locations within the target FPGA. Some CLBs and I/O pads may remain unused; these blocks are marked as void blocks. Then the placement is iteratively improved by random pairwise swapping of locations and evaluating the "goodness" of each swap using the bounding box cost function introduced in

Section 2.3.2.

Inherited from the work of Huang et al. [Huan86], the inital temperature T is set to 20 times the standard deviation in cost after a set of N_{blocks} pairwise moves have been attempted, where N_{blocks} is the total number of CLBs and I/O pads in the circuit. This temperature T_0 is high enough to ensure that almost every early swap is accepted. The number of new configurations evaluated at each temperature T is set to:

$$MovesPerT = innerNum * (N_{blocks})^{4/3} \quad \text{[Swar90]} \tag{2.4}$$

where the scaling factor $innerNum$, which by default is 10, allows a trade-off between CPU time and placement quality.

It is shown in [Swar90] [Lam88] that the most desirable annealing schedule is one that keeps the acceptance rate of moves near 0.44 for as long as possible. VPlace accomplishes this by utilizing the value of the acceptance rate α to control a range limiter R_{limit}, which follows the work of Lam et al. [Lam88].

$$\begin{aligned} &R_{limit}^{new} = R_{limit}^{old} * (1 - 0.44 + \alpha) \qquad and \\ &R_{limit} \in [1,\ maximum\ FPGA\ dimension] \end{aligned} \tag{2.5}$$

Any attempted swap of blocks is allowed only within a square window, where the length of each side of this window eqauls R_{limit}. A small value of R_{limit} ensures that only blocks close together are considered for swap. These "local" swaps tend to result in an increase of acceptance possibility. R_{limit} spans the entire FPGA chip in the beginning, shrunk gradually as the process continues, finally reducing to "1" where only "local" refinement is necessary.

A robust FPGA placement tool must be able to handle a wide variety of circuits

of different sizes. Consequently, as the core of any simulated-annealing based implementation, the annealing schedule must automatically adapt to different circuits. The VPlace annealing schedule is based on the following methodology: When the temperature T is so high that almost every move is accepted, the FPGA configurations randomly move from one to another with little benefit obtained in cost. Conversely, at the end, very few moves are accepted due to the extremely low temperature T and the fairly high-quality of current placement. Very little improvement is obtained at this stage. VPlace works fast by relatively increasing the amount of time spent on exploring the solution space in the medium stage, which is more productive; it bypasses the "futile" periods when little benefit in solution quality is expected.

The exact update schedule of T in VPlace is as follows:

$$T_{new} = \begin{cases} 0.5 * T_{old}, & \text{acceptance rate} > 0.96 \\ 0.9 * T_{old}, & 0.8 < \text{acceptance rate} \leq 0.96 \\ 0.95 * T_{old}, & 0.15 < \text{acceptance rate} \leq 0.8 \\ 0.8 * T_{old}, & \text{acceptance rate} \leq 0.15 \end{cases} \tag{2.6}$$

Finally, the placement tool terminates when the temperature T falls below a certain fraction of the average cost per net. This makes acceptance of any cost increasing move almost impossible.

$$T_{end} = 0.005 * \frac{\textit{bounding box cost}}{\textit{total number of nets}} \tag{2.7}$$

Currently, amongst the academic tools, VPlace holds the record of obtaining the best placement quality within a reasonable amount of CPU time.

2.4.5 Move Evaluation Technique in VPlace

Evaluating pairwise moves is a time-consuming task. In most iterative FPGA placement heuristics, the vast majority of CPU time during the optimization is spent on calculating the effects caused by such perturbations [Betz99]. It is, therefore, crucial that this computation should be made as fast as possible.

The cost function used in this thesis is the most commonly used wirelength-driven bounding box cost described in Section 2.3.2. Consider the computation of ΔC caused by a location swap between two blocks. The only terms in Equation 2.3 that change are those corresponding to the nets attached to the two swapped blocks. The bounding boxes of all the nets attached to these two blocks must be recomputed to determine ΔC. The most straightforward method is to relocate all the terminals for each net exhaustively, which is a $O(k)$ operation for a k-terminal net.

In VPlace, a new technique called "incremental bounding box evaluation" is introduced as an alternative to this brute-force computation. For each net, a data structure contains not only the coordinates of the four sides of the corresponding bounding box but also contains the number of terminals in the net that lie on each side. This extra information is used to determine the new bounding box of the net after a swap by only looking at the moved terminals, rather than all k terminals. Reported in their work [Betz99], this update technique, on average, yields a five fold speedup compared to VPlace without the incremental bounding box evaluation, over ten large MCNC benchmark circuits.

2.5 Multilevel Clustering

In the past few years, the size of FPGA-based circuits has been increasing at a phenomenal rate. Even the outdated benchmarks available to the academic world are large in the combinatorial sense. The long runtime of traditional placement tools is definitely nullifying the time advantage of FPGA based products.

Generally, three strategies are employed to deal with this more intricate problem. The most obvious method is tweaking some parameters to acquire faster placement tools at the cost of obtaining low-quality solutions. For example, by reducing the number of moves per temperature ($innerNum$) in VPlace, we can have a speedup by trading quality for CPU time [Betz99] [Betz97b]. Designing more efficient heuristics or accelerating the convergence of existing FPGA placement algorithms could be another approach (to be discussed in Chapter 3). The third approach is to reduce the complexity of large circuits by *clustering* them into less complicated and easily solvable forms, which helps to decrease the time required to obtain good solutions for the overall problem.

2.5.1 Hierarchical Algorithm

The hierarchical approach is a two step procedure: first proceeding bottom-up then top-down. The bottom-up technique is clustering which involves grouping highly-connected blocks into clusters. Then a top-down method is applied to largely determine the locations for all the clusters. The simplified problem makes the implementation of a time-consuming top-down method, like the simulated-annealing algorithm, more feasible. A de-clustering process proceeds to restore the original FPGA layout according to the previous placement result of clusters. In this process,

the flattened blocks should be placed as close to their *center-of-gravity* as possible. Finally, a localized improvement heuristic is executed to perform trimming work by moving blocks in small regions to achieve the final solution.

2.5.2 Reducing Complexity by Multilevel Clustering

Hagen et al. [Hage92] suggested that the advantage offered by clustering permits the placement algorithms operating on a reduced problem to focus on the most difficult and time-consuming portions. During clustering, FPGA modules (CLBs) are packed into clusters and the original netlist is transformed to a corresponding condensed netlist, which renders the placement problem to be "smoothed" by reducing the number of local minima. Both circuit partitioning [Shin93] [Kary97] and placement [Arei01a] [Sun95] [Mall89] [Thom01] have reported a decrease in computation time by an order of magnitude compared to operating on a flat netlist, through effective clustering techniques.

Early methods of clustering were applied to a single level. As the circuit sizes have grown larger, recent research [Alpe97b] [Arei01a] [Kary97] [Sank99] [Thom01] has shown that adding extra levels of clustering, referred to as multilevel clustering, shown in Figure 2.7, is more manipulable and produces superior results. With multilevel clustering, compaction of the original netlist can proceed more gradually and produce more gentle clusters at each level. In the *coarse* (high) level, a top-down method places blocks in the general regions that they should belong to. As the refinement moves from coarse to *fine* (low) level, the small sized clusters and more detailed steps enable a localized heuristic to find a good final solution [Alpe97b]. Furthermore, all blocks contained in a cluster are assumed to lie at the same loca-

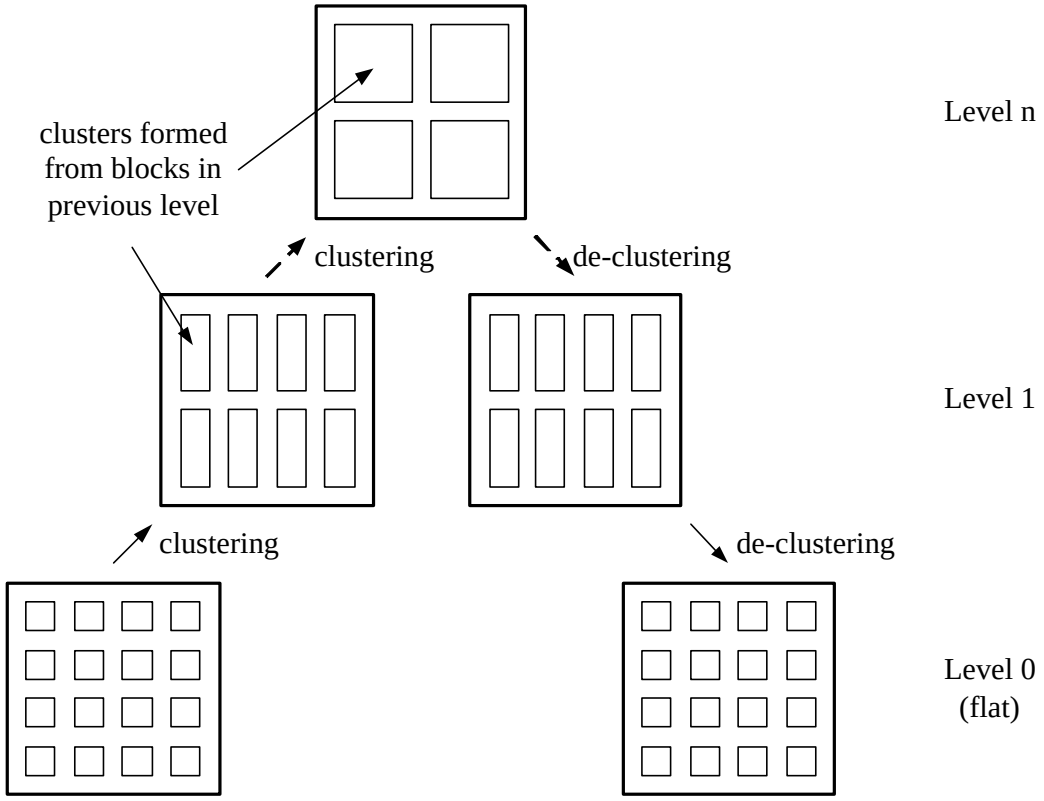

Figure 2.7: Multilevel clustering

tion and, consequently, only one coordinate is attached to this "parent" cluster. All "child" blocks would have their own coordinates when the cluster is de-clustered. Most of these coordinates, however, are close to but not exactly the same as the coordinates of their "parent". For large clusters in a single-level clustering method, the difference between the positions of clustered blocks and the corresponding positions of the same blocks in the flat level can be substantial. Significant future refinement is thus necessary. In a multilevel approach, smaller blocks usually result in much smaller difference, which assists superior quality solutions to be achieved in shorter amount of time [Thom01].

2.6 Previous Clustering Based Approaches

The clustering approach was first applied to the linear placement problem in 1972 by Schuler et al. [Schu72]. Since then, it has found its own position in the problems of circuit partitioning [Shin93] [Kary97] and VLSI standard cell placement [Sun95] [Mall89] [Arei01a] [Thom01]. Only recently, it has been applied to the FPGA placement problem in a limited way[Sank99]. In the rest of this section, we present a general clustering quality measurement and a brief review of two successful clustering methods that have been applied to the placement problem.

2.6.1 Clustering Quality Measurement

Presently, there is no commonly accepted metric to directly measure the quality of clusters obtained by a clustering method. The only agreed-upon measure of clustering quality is the amount of the final improvement obtained with a specific method, which is obviously not an ideal solution. This is because the final placement is affected by many variables, including the size of the circuit being placed, the placement heuristic used, and CPU time consumed.

For circuit clustering, we say that a net is *absorbed* by a cluster if all blocks attached to that net are contained within that single cluster. In [Sun95] and [Sank99], the percentage of nets that would be completely absorbed by a clustering method resulting from a single level of clustering, is ranked as a crucial measurement for the performance of a clustering method. This approach, shown in Figure 2.8, not only decreases the number of entities that need to be improved, but also tends to increase the "internal" connectivity in each cluster, while reducing the number of external interconnections. However, these nets still connect the same modules, so

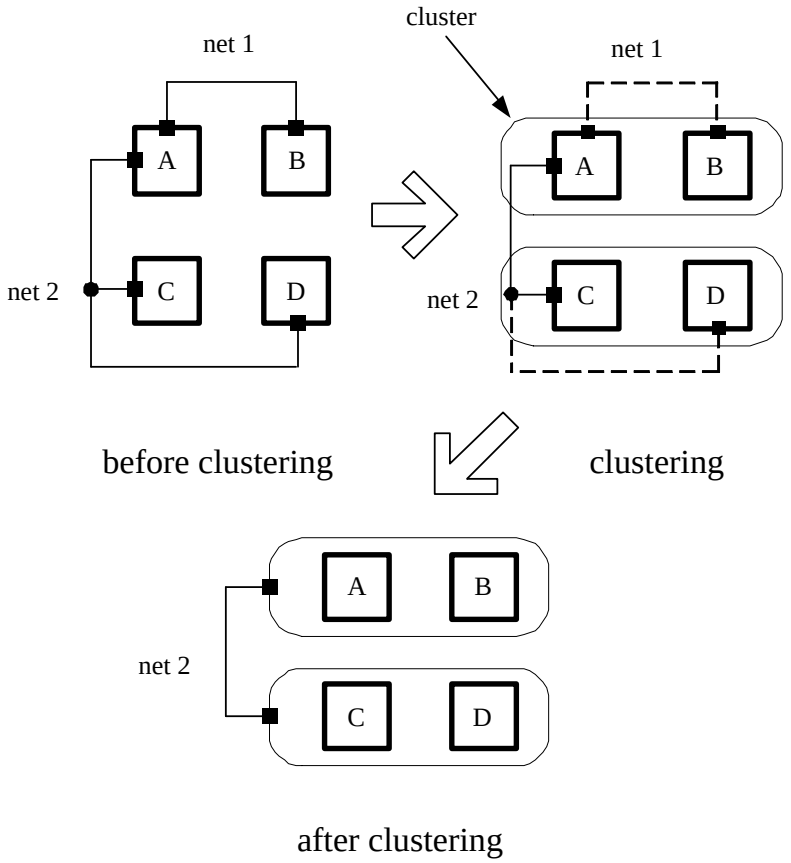

Figure 2.8: Nets absorbed or their terminals reduced. In this exmaple, net1 is completely absorbed (eliminated) and the terminals of net2 are reduced from 3 to 2 in this level of clustering.

a later *de-clustering* procedure is required to restore the concised network to the original form in the lower levels.

Runtime is also used as a measure to evaluate a clustering method. Only a very small portion of runtime should be consumed in the clustering stage compared to the overall placement phase.

2.6.2 Timberwolf95 Hierarchical Approach for Standard Cell Placement

Timberwolf95 [Sun95] is regarded as one of the state-of-the-art placement tools for standard cells. It produces very high-quality results in a fraction of CPU time compared to the corresponding flat version of Timberwolf [Sech85]. A one-level clustering method is used prior to its two-level simulated-annealing placement. Clusters are constructed based on a cost function, which is designed for nets with low fanout to be absorbed into a single cluster. These small nets are easier to fit in a single cluster than nets with larger fanout. First, each net i is assigned a weight w_i, which is inversely proportional to its fanout as follows:

$$w_i = \frac{1}{|F_i| - 1} \tag{2.8}$$

where F_i is the number of terminals for net i. A tree model for multi-terminal nets is used, which assumes that an n-terminal net has $n-1$ edges. If a net spans m clusters, then there are $m-1$ inter-cluster edges existing and if there are j terminals of that net contained in one cluster, then that cluster has $j-1$ edges. Let B_k be the set of all terminals contained within cluster k. The bucket weight W_k of cluster k is then defined as the sum of all the edge weights in that cluster as follows.

$$W_k = \sum_{\langle \forall_i | (F_i \cap B_k) \neq \phi \rangle} (F_i \cap B_k - 1) * w_i \tag{2.9}$$

The first component of the term in the summation represents the number of edges of net i that are contained within cluster k. If the entire net was absorbed by cluster k, the total edge weight contributing to this bucket weight always equals to

1. Let N be the desired number of clusters to be constructed at a specific level. The aggregated bucket weight C would be:

$$C = \sum_{k=1}^{N} W_k \tag{2.10}$$

which is the objective cost function to be maximized for the clustering algorithm.

Although specified for standard-cell architecture, this clustering method can be adapted FPGA placement with little modification. In theory, every evaluation of this clustering cost function can be completed in linear time. However, the use of a simulated-annealing algorithm [Lam88] to optimize the cost function in their implementation is decidedly slow. On the whole, the Timberwolf95 clustering method produces very good results, but it is fairly time-consuming.

2.6.3 Sankar's Hierarchical Approach for FPGA Placement

In [Sank99], an ultra fast hierarchical FPGA placement tool is introduced. A greedy multilevel clustering method, which emphasizes even further the absorbtion of small nets, is performed first. Then, a non-deterministic simple local search algorithm is employed to achieve improvements in the clustered levels. Finally, a low temperature VPlace simulated-annealing based placement is performed to obtain the final solution. The main goal of [Sank99] is to provide an FPGA placement prediction tool that would produce a placement solution ultra fast, with tolerable loss of placement quality[5], to forecast if a circuit can fit in a target FPGA chip.

The clustering algorithm begins by randomly choosing a logic block as the seed,

[5]According to [Sank99], on average, their ultra-fast tool suffered a 22% loss in placement quality, compared to VPlace over twenty MCNC benchmark circuits.

and assigning it to the first available slot in a cluster. Each unclustered block that has connections with the cluster is assigned a weight that rates how high the probability is that it belongs to this cluster. Let J represent the set of nets shared between the candidate block b and the cluster c being constructed. P_j denotes the set of terminals in net $j \in J$, and A_{bc} denotes the set of nets that would be absorbed if block b were assigned to cluster c. The weight, w_b, for each candidate block b is specified as follows:

$$w_b = (\sum_{j \in J} \frac{1}{|P_j| - 1}) + |A_{bc}| \tag{2.11}$$

The first component of the term is the number of connections between the candidate block b and cluster c, with each connection weighted by the fanout of the net in which it lies. It is the same as in the method of Timberwolf95. The second component is the number of nets that would be completely absorbed if this candidate is merged into the cluster c. This gives an extra bonus to blocks with low fanout nets and in the nets that are about to be absorbed when building clusters. The candidate block with the highest score is assigned to the next available slot in cluster c and this selection is repeated until cluster c is full. Next, a new cluster is created by randomly picking another yet unclustered block as the seed. This recursive process is repeated until all blocks are in clusters. Furthermore, the creation of higher level of clusters in a multilevel hierarchical heuristic can proceed in the same manner. The corresponding pseudo-code is shown in Figure 2.9.

Figure 2.10 illustrates the clustering algorithm through an example. Each cluster can hold up to four blocks (clustering size is 4) and three slots have already been filled in the cluster currently being constructed. In order to fill the remaining slot, all "unfixed" blocks that share direct connections with the cluster are exam-

```
    INPUT:
          Init flat netlist;
          Number of clustering levels;
          Size of each cluster;

1.   while(allLevelClustered == false)
2.  {       while(allBlocksAtThisLevelIncludedIn == false)
3.        {       initOneNewClusterStructureAsCurrentCluster();
4.                randomlyPickOneBlockIntoCurrentCluster();
                  /* this seed block must not be included in */
                  /* any other clusters at this level of clustering */
5.                while(thisClusterIsFull == false)
6.               {       calculateBucketWeightForAllCandidateBlocks();
                         /* all candidate blocks must not be included in */
                         /* any other clusters at this level of clustering */
7.                       greedilyPickTheBestBlockIntoCurrentCluster();
8.                } /* end of one cluster creation */
9.        } /* end of one clustering level creation */
10. } /* end of all clustering level creation */
    /* get final clustered netlists at all levels */
```

Figure 2.9: Greedy clustering algorithm [Sank99]

ined. Block x is not considered as a candidate since it has already been "fixed" in a completed cluster, even though it has a direct connection with the target cluster. Block d is not considered either, since it actually does not directly connect to the cluster under construction. The ranking, from high to low, for the available candidate blocks, based on Equation 2.11, is $b = 3$, $a = 2.33$, and $c = 1.33$. Thus, block b would be labeled as "fixed" and will be assigned to the empty slot to finish the construction. If none of the remaining unclustered blocks shares any connection with the cluster under construction (possibly because most blocks are already fixed in clusters), then a candidate block is selected randomly from "unfixed" blocks to fill the free slot in that cluster.

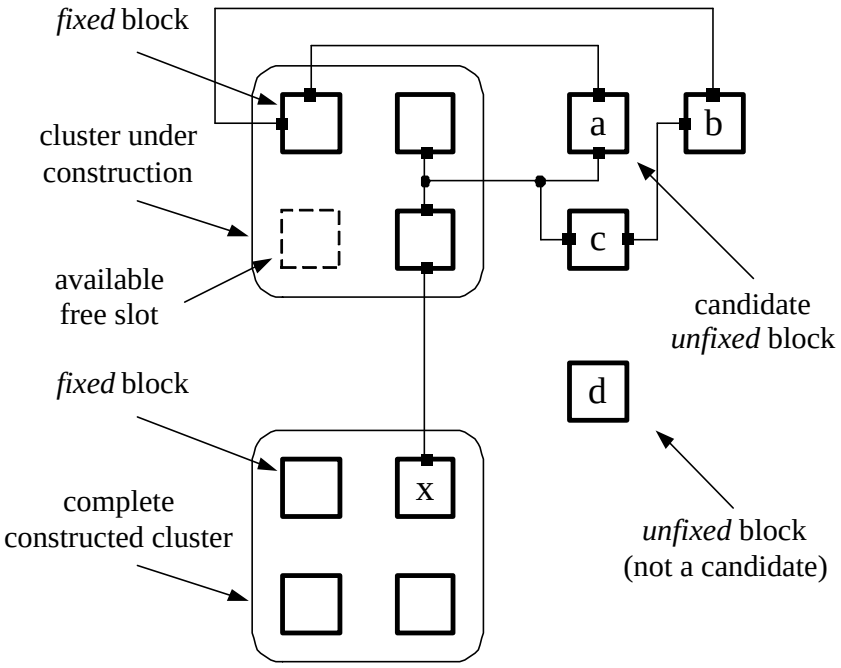

Figure 2.10: Cluster construction example

Unlike the simulated-annealing method used in Timberwolf95, Sankar's method is a greedy constructive method. Only the current best candidate is accepted, which enables the clustering procedure to be completed very quickly.

2.7 Summary

In this chapter, we have presented an overview of FPGA physical design automation in general and FPGA placement in particular. Placement is a complex problem that not only affects the routability of a design but also assists an FPGA-based circuit to meet timing constraints. We also discussed different layout topologies existing in commercially available FPGAs. Each of these layout styles imposes design and optimization constraints, which affect the implementation of a circuit.

The topology used in this thesis is an island style model, which is a generalization of Xilinx products and one that is widely used for such research. We then reviewed some of the previous work in FPGA placement, including simulated-annealing and partitioning-based methods. VPlace, which is a simulated-annealing based tool, was discussed in some detail. The performance of our placement algorithm will be compared with that of VPlace in the next two chapters. Finally, we reviewed some prior work in hierarchical approach that uses clustering to simplify the circuit partitioning and placement problems.

In the following two chapters, two new approaches to deal with FPGA placement are presented. In the next chapter, a "greedy" search heuristic is presented and analyzed. In Chapter 4, one of the existing clustering methods is embedded into our placement tool. This enables us to take a hierarchical approach for placement where our placement algorithm, described in Chapter 3, is used along with clustering and de-clustering, to obtain high-quality solutions.

Chapter 3

Iterative Improvement Techniques

3.1 Introduction

In this chapter, in order to demonstrate the requirement of advanced iterative methods, some basic iterative FPGA placement methods are implemented, i.e., local search techniques. Next, a novel adaptive FPGA placement algorithm, called GSA, is proposed. The algorithm is based on an enhancement to the simulated-annealing algorithm employed in the state-of-the-art placement tool VPlace, which is embedded in the VPR package [Betz00] [Betz99] [Betz97b]. GSA utilizes a suitable memory structure to "remember" part of the short term search history, which helps to guide the search into more promising neighbouring search space.

The final placement quality as well as runtime obtained by local search techniques and GSA are compared with those generated by VPlace to determine the performance of each algorithm. On average, GSA placement tool (with default parameters), achieves a 69% reduction in CPU time (3.2x faster) compared with VPlace over ten MCNC benchmark circuits, while obtaining almost the same final

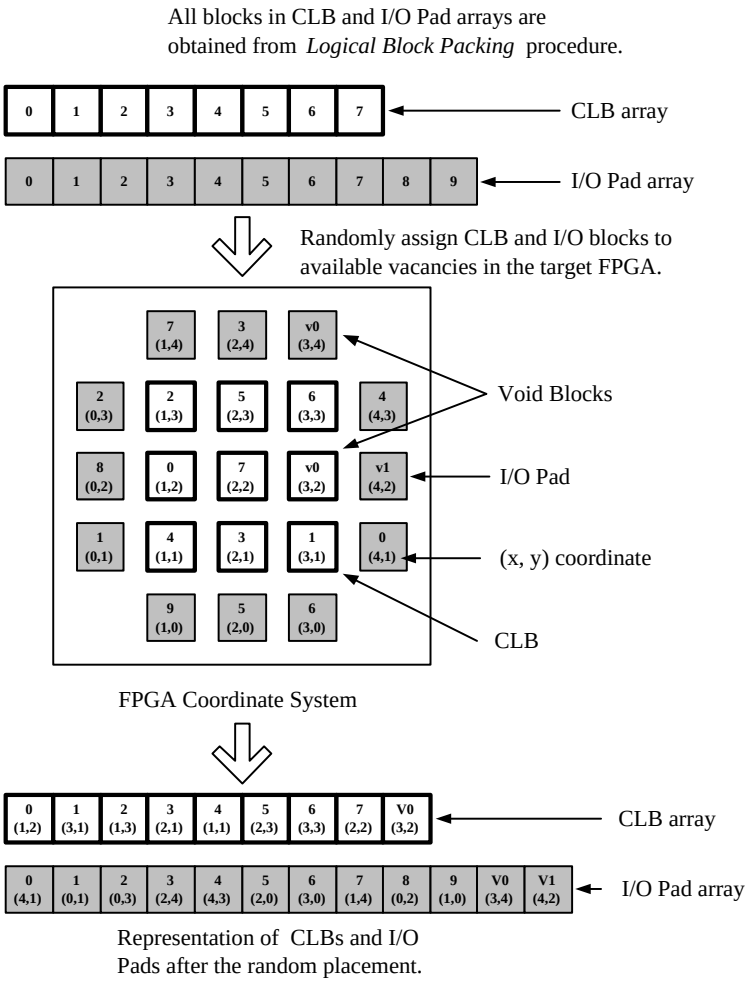

Figure 3.1: Create a random placement solution.

placement quality.

3.2 Iterative Approach

Iterative placement methods start with an initial feasible solution (placement), shown in Figure 3.1. The original blocks in the CLB and I/O pad arrays are obtained from the result of *Logical Block Packing*, which is the previous procedure in

FPGA physical design. Next, these blocks are (randomly) assigned to the available vacancies in the target FPGA chip to generate an initial placement solution. Because of the pre-fabricated architecture of an FPGA, some unused CLBs or I/O pads may exist, which are marked as void blocks[1]. In addition, each block (including void block) is given a coordinate based on the FPGA coordinate system.

A large number of neighbouring solutions will be made and evaluated in an iterative heuristic to gradually obtain improvements. In FPGA placement, there are two ways to create a neighbouring solution: to swap between two CLBs shown in Figure 3.2 or to swap between two I/O pads shown in Figure 3.3. If the attempted swap is accepted, the locations of two blocks are switched, otherwise, this attempted swap will be discarded and the current layout of placement remains unchanged.

In our implementation, the choice of which neighbouring solution generation method (swap CLBs or swap I/O pads) is used is randomly determined. Furthermore, the overall percentage is made corresponding to the ratio of the number of I/O pads to the number of CLBs [Betz00][2]. For example, if there are 8 CLBs and 2 I/O pads existing in an FPGA, and the proposed number of iterations $n = 1000$, then there will have 800 swaps of CLBs and 200 swaps of I/O pads in any random sequence.

Generally, an improving neighbouring solution (a solution reduces the placement cost) is always accepted in an iterative heuristic. While for non-improving solutions (solutions that do not reduce placement cost), the acceptance strategy varies for

[1]These void blocks still can swap with other blocks.

[2]Usually, the number of I/O pads is only a very small fraction of the number of CLBs, especially for large benchmarks. For example, their ratio in circuit "*clma*", the largest benchmark in our test suite, is 144 : 8383.

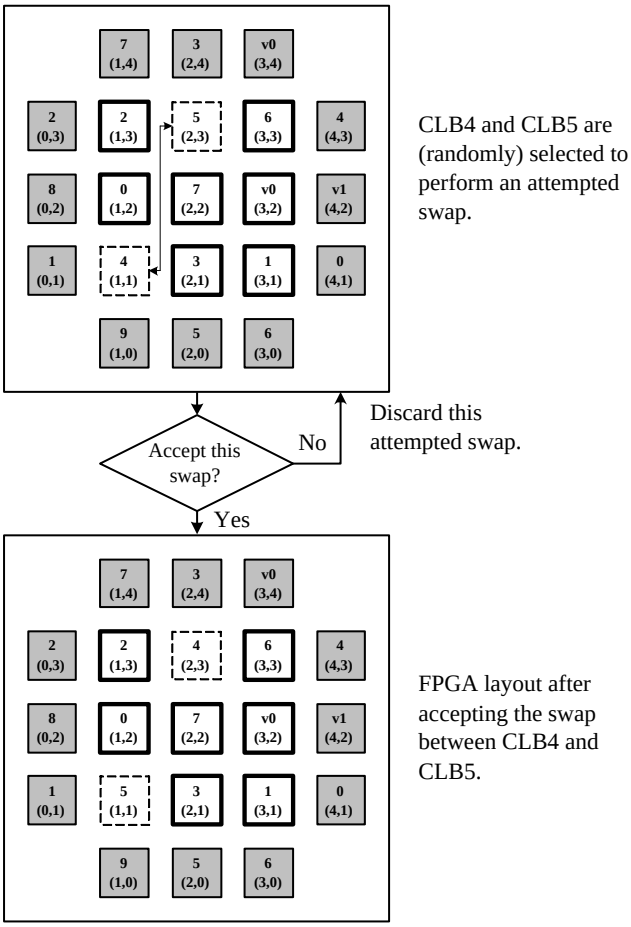

Figure 3.2: Create a neighbouring solution by swapping two CLBs.

different algorithms. In simple local search techniques, a non-improving solution is simply discarded. However, the acceptance of such a non-improving solution in the simulated-annealing is determined by a controlled possibility (*annealing schedule*). Initially, the probability of accepting a deteriorating move is set to a high value so that most evaluations can be accepted. This probability is gradually decreased as the placement is refined so that eventually the chance of accepting a move that makes the placement worse is very low. This randomness could be beneficial, since it allows a good heuristic to unrestrictedly direct the search to find a high-quality

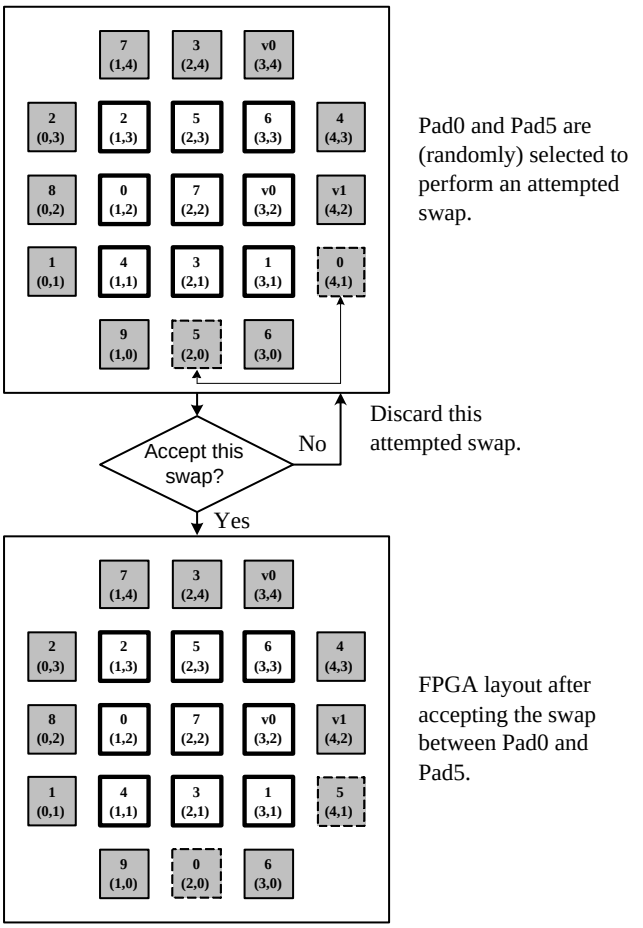

Figure 3.3: Create a neighbouring solution by swapping two I/O pads.

local minimum. On the other hand, excessive randomness may cause the search procedure to spend a disproportionally large amount of time in examining poor solutions or revisiting solutions seen before. What we propose is a type of greedy heuristic that accelerates the convergence of a search. This is our major objective in this chapter.

3.3 Simple Local Search algorithms

The performance of simple local search algorithms, which are the most basic iterative methods in dealing with combinatiorial optimization problems, is always worthwhile to explore.

The basic principle underlying a local search algorithm is that it always moves from the current solution to the next better solution in the neighbourhood in a greedy manner. As mentioned in Section 2.4.2, local search algorithms use two common strategies to move to a better solution, stochastic and deterministic. Local search methods with the first strategy are referred to as non-deterministic or stochastic local search, which accept the first improving solution encountered during random evaluation. These kind of search methods typically stop after a sufficient number of solutions have been evaluated. In our implementation, the number of solutions to be evaluated, $N_{iteration}$, is determined the same as VPlace, which is as follows:

$$N_{iteration} = 10 * (N_{blocks})^{1.33} \quad \text{[Swar90]} \tag{3.1}$$

Where N_{blocks} is the total number of CLBs and I/O pads. Local search methods with the second strategy are referred to as "most improving deterministic local search", which exhaustively enumerate and evaluate the whole neighbourhood set systematically, and accept the best solution. However, the associated computation cost with these methods is prohibitively large for an NP-hard problem where the neighbourhood set is large, such as FPGA placement for large circuits. Let N_{block} be the number of blocks that need to be placed. The number of possible solutions in such a neighbourhood set is $(N_{block} * (N_{block} - 1)/2)$, which grows in the order of $O(N_{block}^2)$. An alternative way to implement most improving deterministic

local search method, which is referred to as "first improving deterministic local search", is exhaustively enumerating and evaluating the whole neighbourhood set systematically, but accepts any improving solutions encounted during the evaluation. Deterministic local search methods terminate when no further improvement can be obtained.

The pseudo-codes for the three implementations are shown in Figures 3.4 to 3.6. Note, the choice of which neighbouring solution generation method (swap CLBs or swap I/O pads) is used in the deterministic method is pre-determined and evaluated systematically.

```
1. S = InitPlacement();
2. while(ExitCriterion() == false)    /* start of loop */
3. {       S_candidate = GenerateMove(S_current);
           /* create a candidate solution from the */
           /* current one by a random pairwise move */
4.         ΔC = Cost(S_candidate) - Cost(S_current);
           /* calculate the change in cost */
5.         if(ΔC < 0)  /* greedily accept only improving swaps */
6.               S_current = S_candidate;  /* accept this candidate */
7. }       /* end of loop */
   /* get final placement solution S */
```

Figure 3.4: Pseudo-code for non-deterministic local search

3.4 Performance of VPlace

The effectiveness of our placement algorithms is measured against an existing high-quality placement tool, VPlace, which has to be evaluated firstly. We make a fair evaluation of how well our tools perform with respect to both runtime and placement quality compared to VPlace, by running all algorithms on the same

```
1. S = InitPlacement();
2. while(ExitCriterion() == false)    /* start of loop */
3. {      S_best = ExhaustivelyGenerateMove(S_current);
          /* find the best solution by exhaustively evaluating */
          /* every possible solution in the whole neighbourhood */
4.        S_current = S_best;
          /* accept the best neighbouring solution */
5. }       /* end of loop */
   /* get final placement solution S */
```

Figure 3.5: Pseudo-code for most improving deterministic local search

computation platform with the same suite of benchmark circuits and the same FPGA architecture.

3.4.1 Target FPGA Architecture

In our approach, an *island style* FPGA model described in Section 2.1, with each configurable logic block (CLB) containing a single 4-input lookup table (4-LUT) and a single D flip-flop is used. The I/O pad pitch-to-logic block ratio[3] [Betz00], which is the number of pads available at each marginal block location, is set to 2. Each CLB has 6 pins: 4 inputs, 1 output, and 1 clock (global) as shown in Figure 3.7.

The exact orientation of each pin on the edge of a CLB is not considered in placement. Furthermore, we assume the FPGA has dedicated resources for routing the *clock*, *reset* and other *global* nets.

[3]A visual pitch-to-logic block ratio example is provided in Appendix B.

```
    /* assuming there are n CLBs (from 1 to n) and */
    /* m I/O pads (from 1 to m) need to be placed */
1.  S = InitPlacement();
2.  flag; /* flag is the exit criterion */
3.  do  /* outer loop */
4.  {       flag = true;  /* initial flag */
          /* start one round systematical CLB evaluation */
5.        for(i = 1; i ≤ n - 1; i++)
6.      {       for(j = i + 1; j ≤ n; j++)
7.            {       S_candidate = Swap(CLB_i, CLB_j);
                    /* find a candidate solution from current one */
                    /* by a determined swap between CLB_i and CLB_j */
8.                  ΔC = Cost(S_candidate) - Cost(S_current);
9.                  if(Δ C < 0)    /* only accept improving swaps */
10.                 {       S_current = S_candidate;  /* accept this candidate */
11.                       flag = false;   /* update flag */
12.                 }
13.           }
14.       }  /* end one round of CLB evaluation */
          /* start one round systematical I/O pad evaluation */
15.       for(i = 1; i ≤ m - 1; i++)
16.       {       for(j = i + 1; j ≤ m; j++)
17.           {       S_candidate = Swap(Pad_i, Pad_j);
                    /* find a candidate solution from current one */
                    /* by a determined swap between Pad_i and Pad_j */
18.                 ΔC = Cost(S_candidate) - Cost(S_current);
19.                 if(Δ C < 0)    /* only accept improving swaps */
20.                 {       S_current = S_candidate;  /* accept this candidate */
21.                       flag = false;   /* update flag */
22.                 }
23.           }
24.       }  /* end one round of I/O pad evaluation */
25. }   /* end outer loop */
26. until( flag == true);
    /* if there is no update of S_current (flag == true), */
    /* the algorithm terminate and export final placement */
```

Figure 3.6: Pseudo-code for first improving deterministic local search

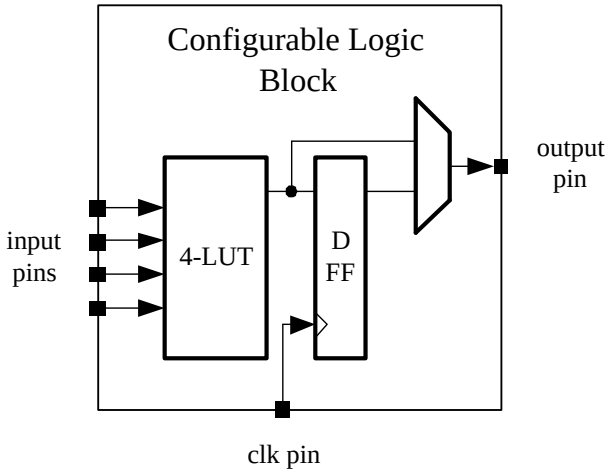

Figure 3.7: Basic CLB architecture

3.4.2 Test Methodology and the Performance of VPlace

We demonstrate the efficacy of our tools by performing head-to-head comparison with VPlace [Betz99]. The comparison is done by running all placement tools over the same suite of circuits on the same computation platform. Ten MCNC benchmark circuits ranging from a few hundred CLBs to nearly ten thousand CLBs are used as test suite (a detailed description of these benchmarks is presented in Appendix A). The benchmarks are organized into three groups: small, medium and large, separated by horizontal lines. GSA is implemented in $C++$ using the *GNU* $g++$ compiler, while VPlace is implemented in C using the *GNU gcc* compiler. All experiments were conducted on the a Sun Sparc 10 dual alpha CPU workstation with Solaris UNIX 8.0 operating system.

For a stochastic heuristic, results obtained with different initial placements are not guaranteed to reach the same local minimum every time. Consequently, our results are obtained by running both placement tools a few times (randomly) to

produces a more meaningful comparison.

VPlace is run with the options "-place_algorithm bounding_box" and "-nodisp". The first option defines the bounding box cost as the placement objective function. The original placement cost function in VPlace is "linear congestion", which is a variation of bounding box cost and used where the capacity of routing channels is irregular. For instance, some FPGA architectures have wider channels in the center regions. However, in this thesis, we assume that the capacity of routing resources is distributed evenly within the whole FPGA, so the "linear congestion" cost function is changed[4] to pure bounding box cost function. The second option turns off the synchronized visual display in VPlace during placement, while GSA does not have any such real-time graphic output.

Table 3.1 provides the detailed results produced by VPlace. These are going to be used as the yardstick throughout the thesis. The table includes columns of arithmetic average of placement cost[5] and CPU time[6]. The values of the best and worst placement and STDEVs obtained through 5 runs are recorded as well.

The measure of quality for an approximation heuristic, like simulated-annealing, is the distance of the solution from the globally optimal solution. However, an optimal solution is usually unachieveable for NP-hard problems, like placement. But, by allowing VPlace to run for a longer period of time, it is possible to obtain an optimal or near-optimal solution. Increasing the runtime of VPlace is accomplished by increasing the $innerNum$ in VPlace from 10 (default) to 100 to perform an extensive search for better results.

[4]This change is made in the original C code of VPlace.

[5]Placement cost is measured in terms of bounding box cost.

[6]CPU time is measured in seconds.

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**2858**	2882	2840	16	**13**	13	13	0
tseng	**9394**	9439	9365	30	**71**	73	69	2
ex5p	**16227**	16329	16131	85	**70**	71	69	1
alu4	**19161**	19233	19066	72	**104**	107	102	2
seq	**24736**	24758	24700	26	**142**	144	137	3
M.avg	**17380**	17440	17316	53	**97**	99	94	2
frisc	**52156**	52936	51554	559	**392**	402	383	7
spla	**61046**	61529	60523	408	**414**	420	404	6
ex1010	**65493**	65679	65360	119	**554**	567	543	9
s38584.1	**64925**	65478	64378	431	**905**	923	887	14
clma	**140391**	141876	139288	1167	**1332**	1349	1297	21
L.avg	**76802**	77500	76221	537	**720**	732	703	11
Avg	**45639**	46014	45321	291	**400**	407	390	6

Table 3.1: Performance of VPlace with default parameters

In this more exhaustive mode of VPlace, the overall runtime is roughly ten times longer than running it with default parameter settings. The averaged results, based on five runs, together with the comparison with VPlace default mode are shown in Table 3.2 The values in column "Impro. %" are obtained as follows: $(Value_{(VPlace)} - Value_{(exhaustive\_VPlace)})/Value_{(VPlace)} * 100$. Symbol "+" in the column denotes improvement obtained by exhaustive VPlace compared with default VPlace [7].

Table 3.2 indicates that there is little benefit gained (1.6%) in terms of solution quality. These results do suggest that the bounding box costs obtained by default VPlace, shown in Table 3.1, are very good, and are difficult to improve upon.

[7]Throughout this thesis, when we say that the result of method one is compared with method two, the comparison is always determined as: $(Value_{(method\_two)} - Value_{(method\_one)})/Value_{(method\_two)} * 100$. Symbol "+" denotes that method one performs better result, while symbol "-" denotes the reverse.

Circuit name	Default avg. cost	Exhaustive avg.cost	Impro. %
e64	2858	2844	+0.49
tseng	9394	9119	+2.93
ex5p	16227	16064	+1.01
alu4	19161	18961	+1.05
seq	24736	24568	+0.68
M.avg	17380	17178	+1.16
frisc	52156	51216	+1.80
spla	61046	59108	+3.18
ex1010	65493	64751	+1.13
s38584.1	64925	64130	+1.22
clma	140391	138336	+1.46
L.avg	76802	75508	+1.69
Avg	45638	44910	+1.60

Table 3.2: Comparison between VPlace (default parameter) and VPlace exhaustive version

3.5 Performance and Conclusion of Simple Local Search Methods

The weakness of the simple local search methods is that they easily end up in local minima. When the number of local minima is large, the probability that a local minimum is also globally minimum is small. Our solution to this problem is to use a multi-start search strategy, which performs a local search 20 times starting with different initial placement over the test benchmark suite to explore better solutions.

As we mentioned in Section 3.3, for the most improving deterministic local search methods, the whole neighbourhood has to be evaluated exhaustively to decide the next step. However, the size of a neighbourhood grows exponentially and results in extra heavy computations. For example, it takes more than one day to complete the placement for the largest ciruit in our test benchmark suite. Consequently, the first improving deterministic local search algorithm is investigated

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**4119**	4247	4006	64	**0.06**	0.07	0.06	0
tseng	**16799**	17062	16478	168	**0.33**	0.34	0.32	0
ex5p	**22179**	22555	21670	203	**0.34**	0.35	0.33	0
alu4	**29168**	29757	28797	297	**0.47**	0.47	0.46	0
seq	**39897**	40380	39080	330	**0.63**	0.64	0.62	0
M.avg	**27011**	27439	26506	250	**0.44**	0.45	0.43	0
frisc	**104306**	105356	102676	768	**1.72**	1.81	1.67	0.02
spla	**113239**	115075	111485	1012	**1.82**	1.89	1.74	0.04
ex1010	**146714**	151926	138229	3131	**2.45**	2.55	2.38	0.08
s38584.1	**210848**	216215	205301	2470	**4.10**	4.37	3.97	0.09
clma	**336998**	341984	332142	2917	**6.94**	7.10	6.82	0.12
L.avg	**182421**	186111	177967	2060	**3.40**	3.55	3.31	0.07
Avg	**102427**	104456	99986	1136	**1.88**	1.96	1.84	0.04

Table 3.3: Performance of non-deterministic local search

instead.

Tables 3.3 and 3.4 show the the performance of non-deterministic and first improving local search methods, respectively. In addition, Table 3.5 illustrates the performance comparison between these two methods.

Tables 3.3 to 3.5 indicate that the placement quality obtained by the non-deterministic local search, on average, is only 21% worse than those obtained by the first improving deterministic method, while the runtime of the former heuristic is significantly reduced compared with the latter (more than 99% faster). While not guaranteed to return a local minimum solution, non-deterministic local search algorithms are more effective than exhaustive methods, like deterministic local search. They are not restricted by the need to cover the entire search space systematically, which actually is not necessary and time-consuming when little improvement can be obtained further (as the solution is already very close to a local minimum). Table 3.6 shows the performance comparison between non-deterministic and VPlace. Al-

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**3706**	3852	3536	82	**0.60**	0.74	0.44	0
tseng	**13632**	14058	13179	219	**13**	18	10	1.8
ex5p	**19630**	20091	19179	272	**15**	19	12	2.0
alu4	**25341**	26211	24761	347	**26**	32	20	3.4
seq	**34572**	35214	33797	360	**38**	46	30	4.2
M.avg	**23294**	23894	22729	300	**23**	29	18	2.8
frisc	**86550**	88164	84126	1048	**215**	299	166	33.4
spla	**95562**	99358	92926	1447	**221**	272	191	25.6
ex1010	**105705**	113546	101054	3072	**537**	742	373	75.0
s38584.1	**156552**	160803	151292	2346	**945**	1288	847	106.1
clma	**267510**	277685	257242	5284	**1899**	2814	1558	262.1
L.avg	**142376**	147911	137328	2639	**763**	1083	627	100.5
Avg	**80876**	83898	78109	1448	**391**	553	321	51.4

Table 3.4: Performance of first improving deterministic local search

Circuit name	**Ave.cost impro.%**	Max.cost impro.%	Min.cost impro.%	**Ave.CPU t impro.%**	Max CPU t impro.%	Min CPU t impro.%
e64	**-11**	-10	-13	**+90**	+90	+86
tseng	**-23**	-21	-25	**+97**	+98	+97
ex5p	**-13**	-12	-13	**+98**	+98	+97
alu4	**-15**	-13	-16	**+98**	+98	+98
seq	**-15**	-15	-16	**+98**	+99	+98
M.avg	**-17**	-15	-17	**+98**	+98	+97
frisc	**-20**	-20	-22	**+99**	+99	+99
spla	**-18**	-16	-20	**+99**	+99	+99
ex1010	**-39**	-34	-37	**+99**	+99	+99
s38584.1	**-34**	-34	-36	**+99**	+99	+99
clma	**-25**	-23	-29	**+99**	+99	+99
L.avg	**-28**	-25	-29	**+99**	+99	+99
Avg	**-21**	-20	-23	**+99**	+99	+99

Table 3.5: Comparison between non-deterministic and first improving deterministic local search

Circuit name	**Ave.cost impro.%**	Max.cost impro.%	Min.cost impro.%	**Ave.CPU t impro.%**	Max CPU t impro.%	Min CPU t impro.%
e64	**-44**	-47	-41	**+99**	+99	+99
tseng	**-79**	-81	-75	**+99**	+99	+99
ex5p	**-36**	-38	-34	**+99**	+99	+99
alu4	**-52**	-55	-51	**+99**	+99	+99
seq	**-61**	-63	-58	**+99**	+99	+99
M.avg	**-57**	-59	-55	**+99**	+99	+99
frisc	**-100**	-99	-99	**+99**	+99	+99
spla	**-85**	-87	-84	**+99**	+99	+99
ex1010	**-124**	-131	-111	**+99**	+99	+99
s38584.1	**-225**	-230	-218	**+99**	+99	+99
clma	**-140**	-141	-139	**+99**	+99	+99
L.avg	**-134**	-137	-130	**+99**	+99	+99
Avg	**-95**	-97	-91	**+99**	+99	+99

Table 3.6: Comparison between simple non-deterministic local search and VPlace

though non-deterministic simple local search can be completed very quickly, the quality of placement obtained on average is 95% worse than that obtained by VPlace, which is definitely untolerable for the FPGA placement problem. Furthermore, a general conclusion can be made according to this evidence that the more complex (large) a circuit is, the less effectively the local search performs. For medium size circuits, on average, the "distance" between the two algorithms is 57%, while for large size circuits, it increases to 134%.

We also conclude that simple local search algorithms cannot be used as primary methods for FPGA placement as they easily get trapped into local minima. However, this heuristic can be finished in a very small amount of CPU time, which enables it to be embedded into other techniques as a complementary method.

In the following sections, we discuss the methodology and performance of another iterative improvement method, GSA, which is characterized by the ability to escape local optima, while still convergs quickly.

3.6 Greedy Simulated-Anealing Algorithm

As one of the best iterative methods, simulated-annealing is regarded both in industry and academia as the dominant method for performing FPGA placement. However, while earning the reputation for achieving the best placement quality, simulated-annealing suffers from the problem of slow convergence. Consequently, it requires more CPU time compared with other popular methods [Mulp01].

Our inspiration mainly comes from the VPlace simulated-annealing algorithm (described in Section 2.4.3). Additionally, a memory structure, which records a short term of search history, is employed to help our new placement tool converg more quickly than VPlace. Typically, all improving attempts would be accepted in an iterative algorithm. Simulated-annealing accepts a non-improving move depending on a controlled possibility. In our approach, a non-improving move would be recorded as search history rather than be accepted or rejected instantly. When the history, which is made of consecutive non-improving moves, reaches the maximum length, the least non-improving one among all evaluated moves recorded in the history is mandatorily accepted. Furthermore, once an acceptance (including improving moves) is made, the history is erased.

This is the "first improving or least non-improving among D_{greedy} neighbouring moves" strategy, where the parameter D_{greedy} is an integer. The value of D_{greedy} (maximum length of history) is adaptively changed and is usually much less than the total number of solutions included in the neighbourhood, to prevent exhaustive scanning. The detailed methodology of our algorithm is described as follows:

Start from an initial placement, as in any other iterative approach, two logic blocks are selected randomly to create a new solution in the neighbourhood of the

current solution. If the swap results in a decrease in cost, the move is always accepted and the locations of these two blocks are switched, which is exactly the same tactic as in the simulated-annealing algorithm. However, if the move results in an increase in cost, the two blocks together with the change of cost are recorded in a suitable structure, if the structure is empty. If the structure is not empty, then the change in cost of this move is compared with corresponding value already recorded in the structure. Only the better or less deteriorating move is preserved in the structure. A large number of such moves are made and evaluated to gradually obtain improvements. The number of continuous non-improving moves is counted until the number reaches the value specified by the parameter D_{greedy}. Then the move recorded in the structure is mandatorily accepted even though it makes the placement worse. Additionally, the contents of the memory is erased once any acceptance (including the acceptance of improving moves) is made.

D_{greedy}, used to select the best move from D_{greedy} non-improving evaluations, is a key parameter in controlling the aggressiveness of the placement tool. Initially, D_{greedy} is very small so non-improving moves are easily to be accepted; however, D_{greedy} is gradually increased as the placement is refined so that eventually the probability of accepting a move that makes the placement worse is very low. It is due to the algorithm's annealing-like search strategy and the inherent greedy characteristic that we name our algorithm the *Greedy Simulated-Annealing* (GSA). The ability to accept non-improving moves allows GSA to escape local minima in the cost function, while still converging quickly. It is more greedy than the conventional simulated-annealing algorithm, which results in a much faster convergence in placement. The pseudo-code for GSA algorithm is shown in Figure 3.8.

The update of D_{greedy}, the exit criterion to terminate the search, the number

```
1.  S = InitPlacement();
2.  D_greedy = InitD_greedy();
3.  R_limit = InitR_limit();   /* R_limit is set to the whole chip initially */
4.  S_best = S;   /* S_best is used to record the overall best solutions so far*/
5.  BestInOneD_greedyRound = NULL;
    /* BestInOneD_greedyRound keeps a record of a "bad" move. */
    /* However, it is the best so far in one D_greedy evaluation round */
6.  IterationOneD_greedyRound = 0;
    /* it counts the number of iterations in one evaluation round */
7.  while(ExitCriterion() == false)    /* outer loop */
8.  {       while(InnerLoopCriterion() == false)    /* inner loop */
9.          {       S_candidate = GenerateMove(S_current, R_limit);
                    /* create one candidate solution from current one by */
                    /* a random pairwise move within the limiter R_limit */
10.                 IterationOneD_greedyRound++;
11.                 ΔC = Cost(S_candidate) - Cost(S_current);
                    /* calculate the change in cost */
12.                 if(ΔC < 0)   /* if this move is beneficial */
13.                 {       S_current = S_candidate;   /* accept this candidate */
14.                         IterationOneD_greedyRound = 0;
15.                         BestInOneD_greedyRound = NULL;
16.                 } else if(ΔC < ΔC(BestInOneD_greedyRound))
                            /* it is better than the best we obtain in this round */
17.                         BestInOneD_greedyRound = S_candidate;
                            /* record this S_candidate to BestInOneD_greedyRound */
18.                 if(IterationOneD_greedyRound == D_greedy)
                            /* this IterationOneD_greedyRound has reached its end */
19.                 {       S_current = S(BestInOneD_greedyRound);
                            /* mandatory accept the S_candidate recorded in the */
                            /* BestInOneD_greedyRound, which is the best one in */
                            /* this D_greedy round even if it increases the cost */
20.                         IterationOneD_greedyRound = 0;
21.                         BestInOneD_greedyRound = NULL;
22.                 }   /* end of one round of D_greedy evaluation */
23.         }   /* end of inner loop */
24.         if(S_current ≥ S_best)    /* adaptive update mechanism */
25.         {       Update(D_greedy);   /* D_greedy_new = α * D_greedy_old */
26.                 Update(R_limit);   /* R_limit_new = β * R_limit_old */
27.         } else
28.                 S_best = S_current;   /* update S_best */
29. }   /* end of outer loop */
    /* get final placement solution S */
```

Figure 3.8: Pseudo-code for Greedy Simulated-annealing (GSA)

of moves attempted at each D_{greedy} (*inner loop* line 8-23 in the pseudo-code) and a method to restrict the generation of potential moves (to be discussed in Section 3.6.2) are defined by an *update schema*.

Many effective optimization algorithms, as well as our GSA, make use of combinations of exploration and exploitation search strategies to find the "global" optimum. Exploration investigates new areas in the search space, and exploitation takes advantage of the knowledge found previously to achieve better solutions. A pure random search is good at exploration, but performs no exploitation. On the other hand, an absolute hill-climbing method is good at exploitation, but does little exploration. A good balance of these two strategies is crucial to obtain good results in a reasonable amount of CPU time. In our approach, it is determined by an update schema, which we discuss next.

3.6.1 Search Greedy Degree

Our update schema incorporates some of the best features from other works, including Swartz et al. [Swar90] and Betz et al. [Betz99] [Betz97b]. In VPlace simulated-annealing, the number of solutions evaluated at each temperature is referred to as $MovesPerT$, which is based on Equation 2.4. Let N_{blocks} be the total number of CLBs and I/O pads in the circuit. Just as the parameter $MovesPerT$, the number of pairwise moves attempted at each value of D_{greedy} ($MovesPerD_{greedy}$)in GSA is set to:

$$MovesPerD_{greedy} = innerNum * (N_{blocks})^{1.33} \tag{3.2}$$

where the parameter $innerNum$ can be overridden on the command line to allow different trade-offs between CPU time and placement quality. Reducing the value of

$MovesPerD_{greedy}$ has the effect of reducing the runtime at the expense of quality. In contrast, increasing $MovesPerD_{greedy}$ has the effect of consuming more CPU time to obtain more high-quality quality solutions. Furthermore, GSA utilizes the same move evaluation technique as in VPlace, which is "incremental bounding box evaluation" method (presented in Section 2.4.5).

3.6.2 Local Search Window

Limiting the scope of moves (swaps) within the region of the original block positions has been shown to give superior results compared to unrestricted moves when a globally good placement is already achieved [Lam88]. This follows from the fact that when a placement is globally good, most blocks are within the locality of where they should be. By implementing a swap distance limiter between the source and destination blocks, the final locations of blocks can be found faster and the speed of convergence to a corresponding local minimum is increased. On the other hand, if the placement begins from a globally poor configuration, such as an absolutely random start, the use of the limiter would deteriorate the final solution quality, since a large number of blocks are far away from their ideal positions. In this case, the limiter only prevents those blocks from reaching their good locations.

An example of implementation of this move restriction is shown in Figure 3.9. A source block is selected randomly at first, then a square window, centered by the source block, is laid on the FPGA block matrix. Then a candidate block other than the original one within the window is randomly picked to perform a location exchange with the source block. The size of the square window depends on the value of the parameter R_{limit}, which indicates the maximum distance between the

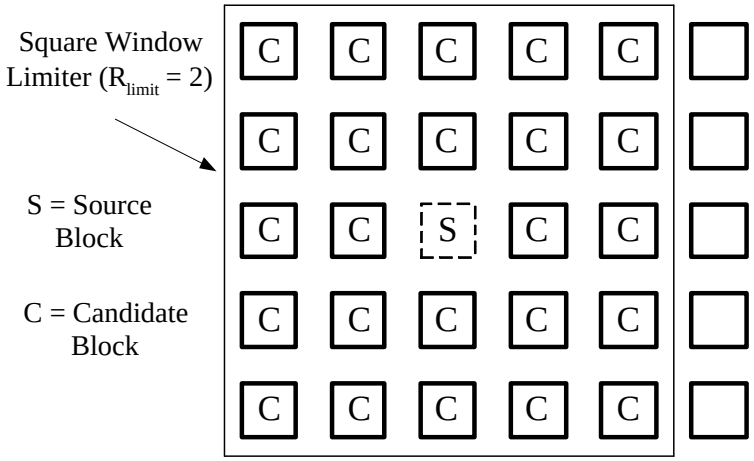

Figure 3.9: Window limiter example. The source block is in the center of a square limiter. Within the square, any other block could be a candidate to be picked to perform swap with the source. The size of the window limiter R_{limit} in this example is 2 (two logic block distance).

source block and the possible location of the target block.

In Timberwolf95 [Sun95], a fixed size window limiter (the value of R_{limit} is pre-defined) is used for all temperatures during the annealing process, which simplifies the determination of a suitable annealing schedule. However, VPlace [Betz99] [Betz97b] shows that altering the window size adaptively over the improvement procedure can increase the acceptance rate of attempted moves (described in Section 2.4.4) , resulting in finding high quality solutions in less CPU time.

In VPlace, at the beginning, the value of R_{limit} equals the whole span of the FPGA. This freedom is necessary for the placement tool to outline a good global placement. As the search progresses, the placement configuration gradually improves and the window limiter R_{limit} is also shrunk accordingly (in conjunction with temperature T), these reducing the scope of possible pairwise moves. Overall, it allows the whole placement procedure to progress from placing blocks in the

general area where they should be to tuning blocks in smaller and smaller regions.

In Section 3.7.2, we are going to show the effectiveness of implementing a range limiter to speed up the search convergence by comparing GSAs with and without this mechanism.

3.6.3 Algorithm with Fixed Parameters

"Fixed" update schemas usually work well within the narrow application range for which they are developed. However, they are not generalized enough to adapt to different problems [Huan86]. Nevertheless, they have the advantages of being easier to implement than adaptive schemas. We begin our approach by implementing a fixed parameter update schema. Figure 3.10[8] shows the behavior of GSA over a medium size circuit with a well-tuned fixed empirical update schema specific for this circuit. Figure 3.11 illustrates the behavior of GSA over a large size circuit with the same update schema. Notice that in Figure 3.10, the curve is fairly smooth whereas Figure 3.11 contains a number of "steps". These steps reveal that excessive evaluations are made, while obtaining little improvement. This is a result of the fact that parameters found to be acceptable for the medium size circuit are not suitable for the other circuit. Although only one example is used here, the results are typical of those obtained with other problems. Clearly, what is required is an adaptive update mechanism that would automatically change the values of the parameters based on the behavior of the search process.

[8] y-axis denotes the bounding box cost; x-axis denotes the number of attempted iterations, which is proportional to CPU time.

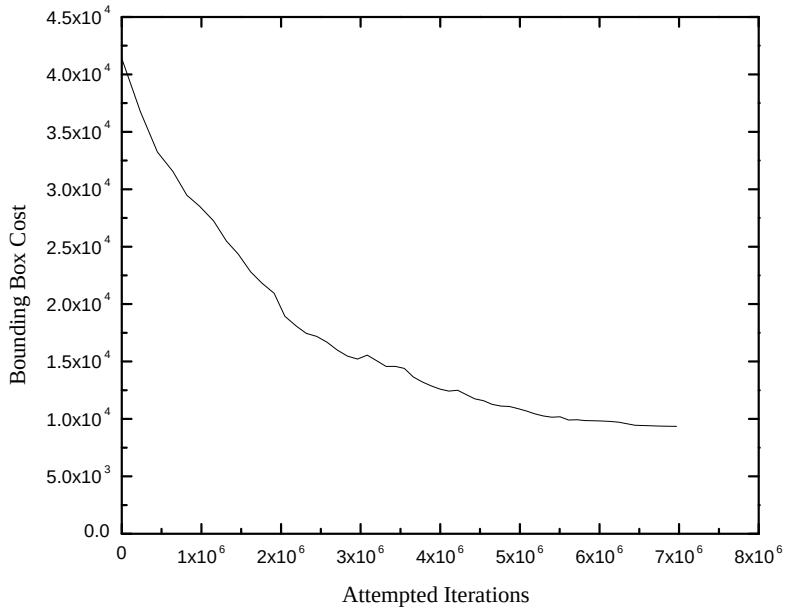

Figure 3.10: Search behavior of GSA with a fixed update scheme over a medium MCNC circuit "*tseng*" (1047 CLBs).

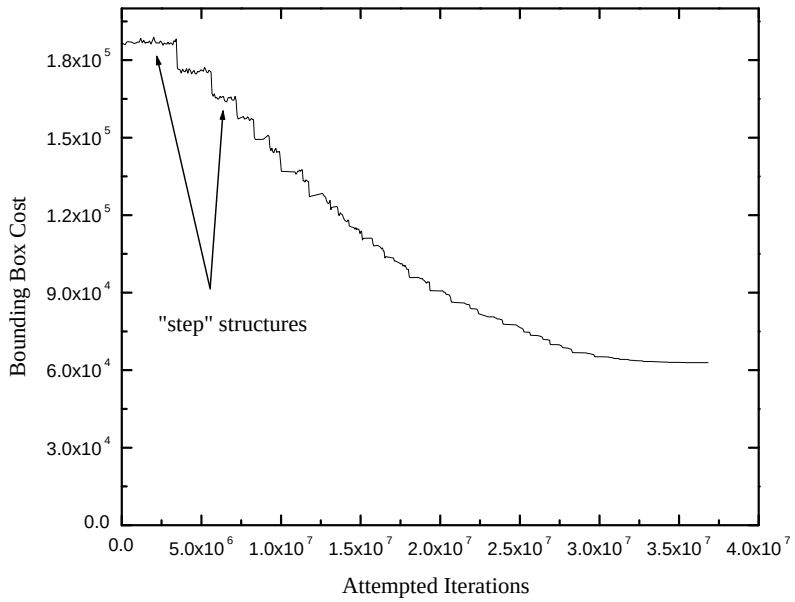

Figure 3.11: Search behavior of GSA with a fixed update scheme over a large MCNC circuit "*spla*" (3690 CLBs).

3.6.4 Adaptive Update Schema

Any FPGA placement algorithm must perform well over a wide range of circuits. As seen in the previous section, such consistency in performance is not possible with a fixed update schema. It is thus imperative to explore adaptive update strategies.

An adaptive annealing schedule in VPlace was described in Section 2.4.4, which determines the update of R_{limit} and temperature T by utilizing the statistics of rate of accepted moves. In our implementation, a parameter S_{best}, which stores the best solution obtained so far during the search procedure, is introduced for the same purpose. The value of S_{best} is set to the initial placement cost at the beginning. Then, a new placement cost $S_{current}$ is obtained after each $MovesPerD_{greedy}$ number of attempted moves that have been evaluated (*inner loop* lines 8-23 in pseudo-code). The difference in cost, $\Delta S = S_{current} - S_{best}$ is calculated. If ΔS is negative, it shows a new lower energy state placement has been found, indicates that the current values of D_{greedy} and R_{limit} are still fruitful. Therefore, no changes are made to the values of D_{greedy} and R_{limit}; these values are used to proceed with another round of search. Only the parameter S_{best} is updated in this case. Conversely, a zero or positive value of ΔS indicates that the current parameters are no longer yielding improving solutions and need to be updated.

In VPlace simulated-annealing, the acceptance rate of attempted moves, which is mostly determined by how the search progresses, serves as a guide to designate the right time to update parameters. In a similar manner, the time for GSA to "squeeze" (when little or no further improvement can be achieved, which is indicated by $\Delta S \geq 0$) one set of parameters dependent on the progress of the search as well. Depending on the value of ΔS, GSA can be made to adaptively

adjust parameters according to each circuit.

Similar to the relationship between temperature T and R_{limit} in VPlace (presented in Section 2.4.4), D_{greedy} and R_{limit} in GSA are two interactive parameters used to control the balance between exploration and exploitation during a search. Initially, D_{greedy}, which determines the aggressiveness of the search procedure, is set to 2 to ensure that the maximum (reasonable) number of attempted evaluations are accepted. At the same time, R_{limit}, which limits the movement of blocks within specific regions, is set to the whole span of the chip to allow maximum exploration to be performed. Whenever an update of D_{greedy} and R_{limit} is necessary ($\Delta S \geq 0$), a new value of D_{greedy} is computed as $D_{greedy\_new} = [\alpha * D_{greedy\_old}]$, where the value of α depends on:

$$\alpha = \begin{cases} 1.5, & \text{if } D_{greedy} \leq 10 \\ 1.3, & \text{else if } R_{limit} = 1 \\ 1.05, & \text{otherwise.} \end{cases} \tag{3.3}$$

At the same time, the value of the limiter R_{limit} is revised as follows: $R_{limit\_new} = [\ \beta\ *\ R_{limit\_old}\ ]$ and limited to the range $(1 \leq R_{limit} \leq \textit{maximum FPGA span})$. If the value of R_{limit} becomes less than 1, it is set to 1, otherwise, the value of β is determined as follows:

$$\beta = \begin{cases} 1, & \text{if } D_{greedy} \leq 10 \\ 0.9, & \text{otherwise.} \end{cases} \tag{3.4}$$

This strategy causes R_{limit} to be the size of the entire chip for the first part of the search, shrink gradually during the middle stages of the search, finally settling at "1" (logic block distance) during the latter part of the search.

When D_{greedy} is very small (≤ 10), non-improving moves are easily accepted. In this phase, the algorithm mainly explores the solution space, hoping a good global solution can be obtained. At the same time, R_{limit} is maximized to enable free block movements. During this exploration stage, have excessive numbers of swaps do not result in obtaining significant improvements. Conversely, at the end of placement stage, very few attempted moves can be accepted. This is because the value of D_{greedy} is very large (the process is very greedy, mostly only improving moves are allowed to be accepted) and the current placement is of fairly high quality. This is a pure exploitation stage and can be identified when R_{limit} equals "1" in the GSA. With this in mind, we create an update schedule to relatively increase the amount of CPU time (small α) spent in the more "productive" medium stage of the search, which represents a combination of exploration and exploitation.

The performance of GSA with different combinations of α and β is shown in Figure 3.12. In addition, Figures 3.13 and 3.14 illustrate the performance of GSA over a medium and a large sized circuit with the same combinations of α and β. The exact values of α and β listed here were found through experiments. Even though the values of α and β suggested by Figures 3.12 to 3.14 lead to best trade-offs between CPU time and placement quality, GSA is not overly sensitive to slight variations of them. As long as α and β have the right form based on our motivation, it performs reasonably well.

We terminate the placement process adaptively by making use of S_{best}. When the value of R_{limit} equals 1, the placement tool terminates if S_{best} cannot be updated in 5 consecutive $MovesPerD_{greedy}$ evaluation rounds. In this pure exploitation stage, little improvement can be achieved, and it is unlikely any further beneficial can be obtained after so many evaluations. Again, the performance for the

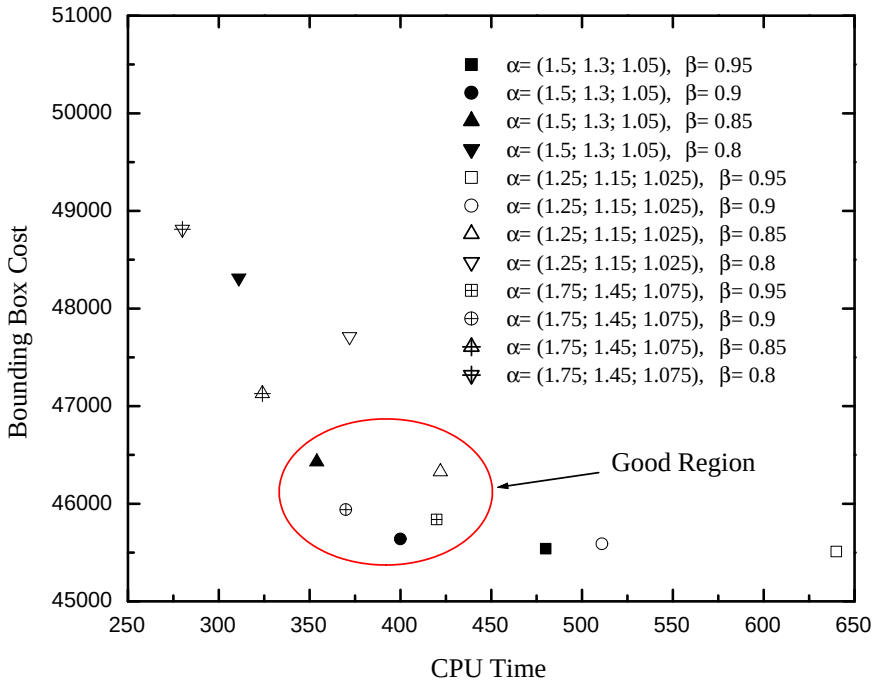

Figure 3.12: Quality-time plot of GSA (10 circuits average) with different α and β combinations ($innerNum = 5$).

placement tool is not too sensitive to the value of this termination number. Any value between 3 and 10 is reasonable, a larger value receiving slightly higher quality results at the expense of slightly increased CPU time.

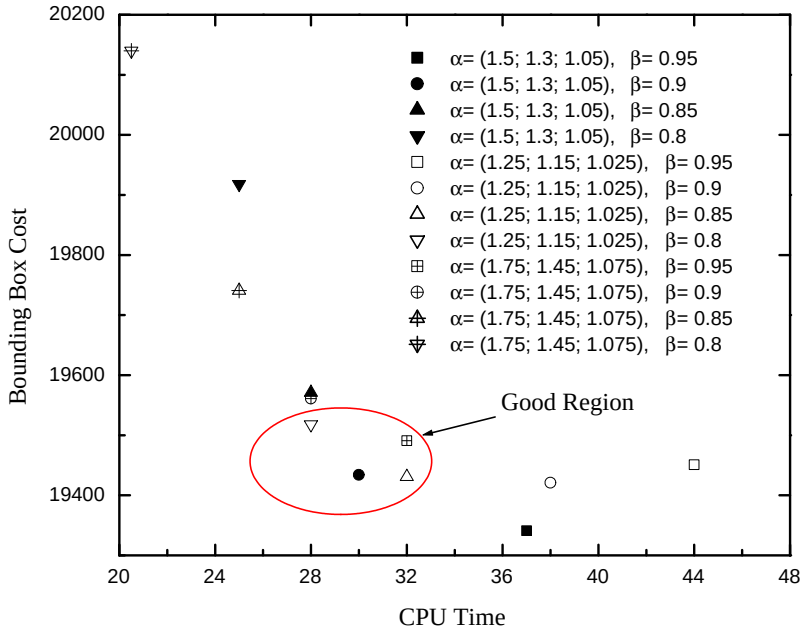

Figure 3.13: Quality-time plot of GSA over a medium circuit "*alu4*" (1522 CLBs) with different α and β combinations ($innerNum = 5$).

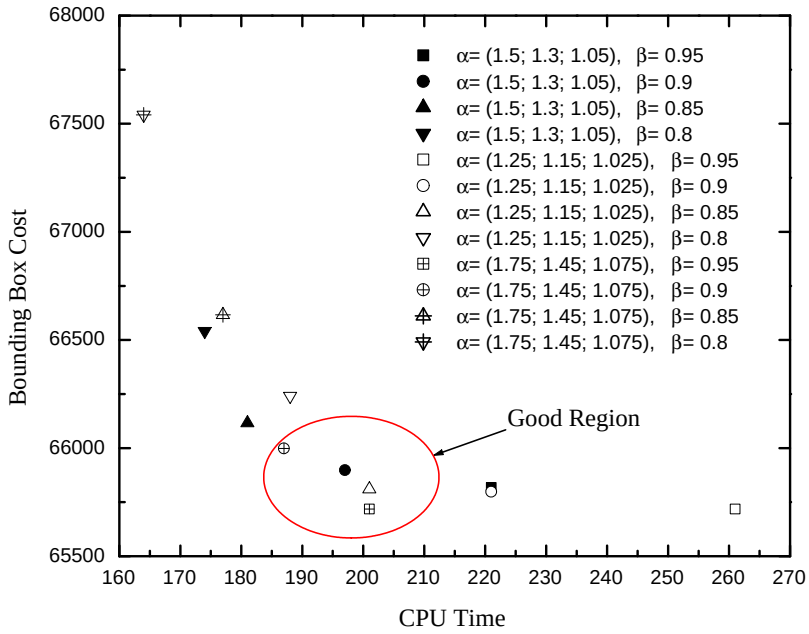

Figure 3.14: Quality-time plot of GSA over a large circuit "*ex1010*" (4598 CLBs) with different α and β combinations ($innerNum = 5$).

3.7 Performance and Conclusion of GSA

3.7.1 Search Behavior Comparison between VPlace and GSA

First, we consider the search behaviors of VPlace and GSA. For the purpose of comparison, we plot the search curves (placement quality vs. number of attempted evaluations) of VPlace and GSA for a medium size circuit in Figure 3.15 and a large circuit in Figure 3.16. Due to lack of runtime information output in VPlace, the number of attempted evaluations is employed instead [9].

Evaluating possible solutions consumes most of the computation time [Betz99] in both heuristics. Since both algorithms make use of the same "incremental net bounding box cost evaluation" technique (presented in Section 2.4.5), the evaluation time for each placement tool, on average, is the same. We have changed the default $innerNum$ of VPlace and GSA to enable both tools to terminate after almost the same number of attempted moves have been made. Hence, Figures 3.15 and 3.16 illustrate the general behavior of both algorithm.

Both Figures 3.15 and 3.16 shows better performance for both medium and large size circuits. The search curves of GSA converge much faster than those for VPlace, while the quality of final placement is very close to that obtained by VPlace. Quick convergence in the beginning and relatively long "flat" area at the end of the search period for GSA allow us to have a much faster placement tool at the cost of slightly increasing in placement cost. Although the figures are for two specific problem instances, they are typical of the behavior of both algorithms.

[9] All behavior illustrations of methods used in this thesis are plotted as placement quality vs. number of attempted evaluations.

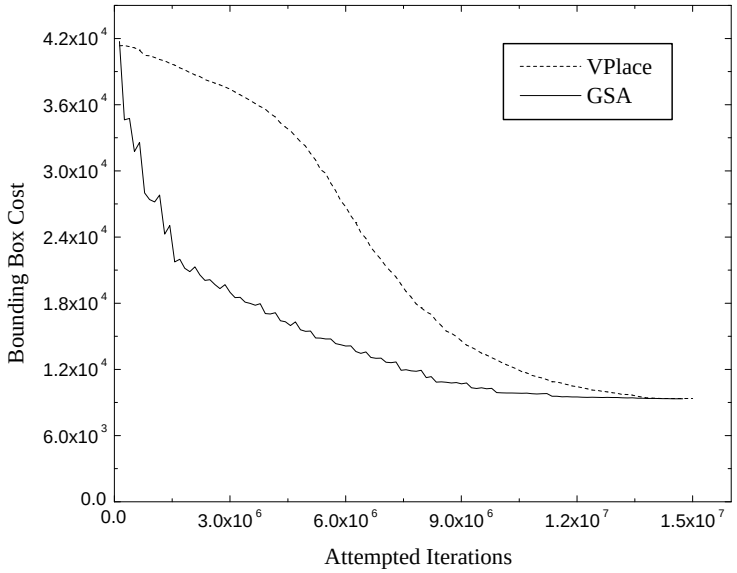

Figure 3.15: Search curve comparison between GSA and VPlace over a medium size MCNC circuit "*tseng*" (1047 CLBs).

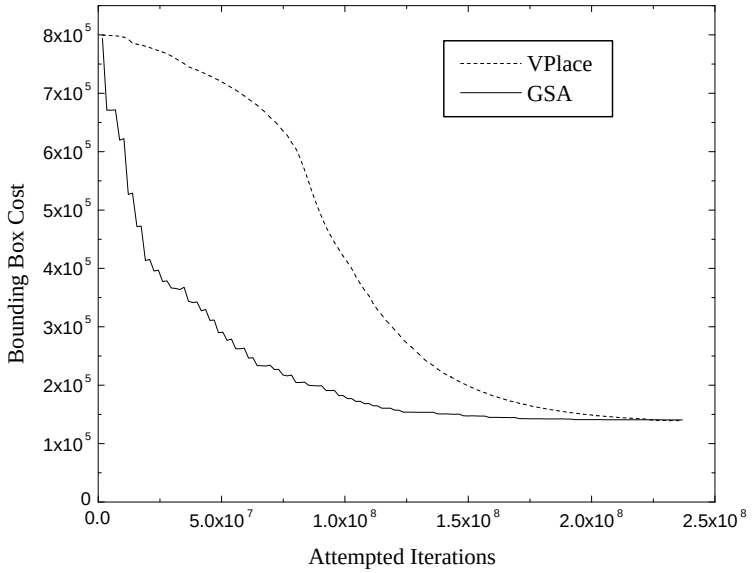

Figure 3.16: Search curve comparison between GSA and VPlace over a large size MCNC circuit "*clma*" (8381 CLBs).

3.7.2 Effectiveness of R_{limit}

As mentioned in Section 3.6.2, adaptively limiting the maximum distance (R_{limit}) between two swapping blocks can accelerate the convergence of search. For comparison with standard GSA, we implemented another version of GSA that does not have a window limiter. In the latter version, R_{limit} always equals the span of entire FPGA. Figures 3.17 and 3.18 show the search curves for the two versions of GSA, for two different sized circuits. The figures clearly illustrate the effectiveness of the move restriction mechanism in speeding up convergence in the medium and final stages.

3.7.3 Performance of GSA

Similar to VPlace simulated-annealing, the $innerNum$ (number of moves per D_{greedy}) of GSA can be overridden through the command line, allowing user-controlled trade-offs between placement quality and CPU time. Our suggested $innerNum$ for GSA is 5; for comparison we increase it to 10 (exhaustive version) to obtain better placement quality with longer runtime. Tables 3.7 and 3.9 list the results obtained by GSA with $innerNum$ set to "5" and "10", respectively. Tables 3.10 and 3.11 provide a comparison with those obtained by VPlace (described in Section 3.4.2). These tables show that GSA with $innerNum = 5$ and GSA with $innerNum = 10$ produced almost the same placement quality over the test suite, while the former is significantly faster.

In addition, Table 3.8 provides results obtained through 20 default GSA runs (more general). The results show little difference with those obtained through "5" runs. Figure 3.19 illustrates the normalized average placement cost and CPU

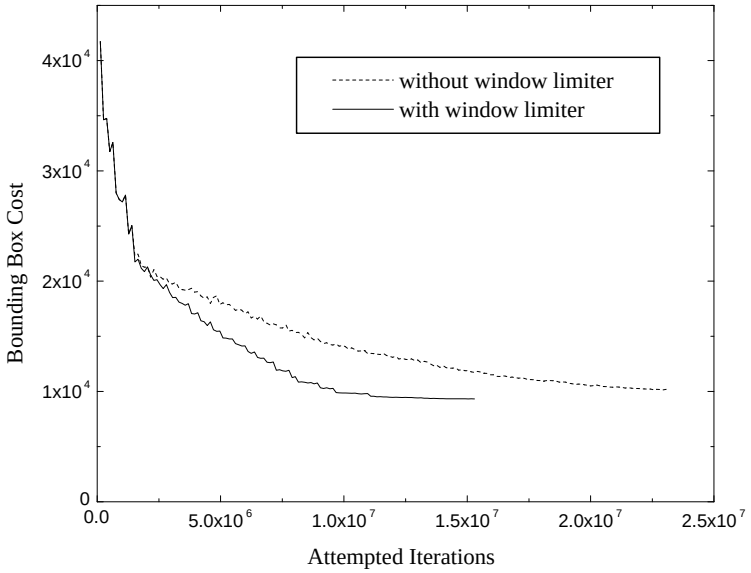

Figure 3.17: Search behavior of GSA with and without swap restriction over a medium size MCNC circuit "*tseng*" (1047 CLBs).

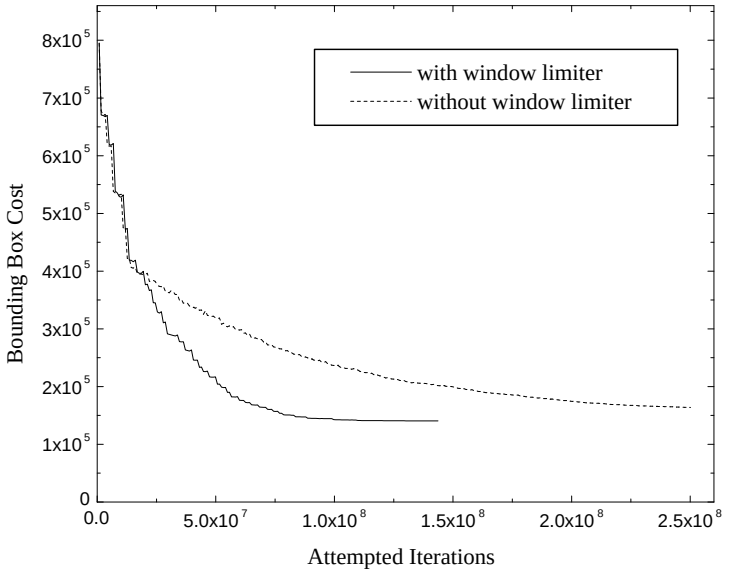

Figure 3.18: Search behavior of GSA with and without swap restriction over a large size MCNC circuit "*clma*" (8381 CLBs).

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**2872**	2880	2864	6	**3**	3	3	0
tseng	**9444**	9594	9263	125	**19**	20	19	1
ex5p	**16280**	16412	16213	76	**18**	19	17	1
alu4	**19434**	19637	19222	166	**30**	31	30	1
seq	**24866**	24961	24749	85	**39**	42	37	2
M.avg	**17506**	17651	17362	113	**27**	28	26	1
frisc	**52368**	52698	52100	256	**125**	135	116	7
spla	**60722**	61615	59638	787	**135**	141	130	4
ex1010	**65898**	66171	65634	245	**197**	214	188	10
s38584.1	**65674**	65910	65395	242	**325**	333	314	8
clma	**140878**	141933	140026	939	**574**	602	527	38
L.avg	**77108**	77665	76559	493	**271**	285	255	13
Avg	**45844**	46181	45510	292	**146**	154	138	7

Table 3.7: Performance of GSA with innerNum = 5 (default)

runtime statistics for the results obtained by GSA default, GSA exhaustive, and VPlace default based on grouped benchmark circuits.

The various tables and figures show GSA produces almost the same quality placement as VPlace in a significantly shorter amount of time. When $innerNum$ is set to 5, which is the default value in our implementation, on average the GSA achieves 69% reduction in CPU time (3.2x faster) compared with VPlace at the cost of only a slight (0.5%) decrease in the final placement quality. When $innerNum$ is increased to 10 to trade CPU time for better quality, GSA obtains a slight gain of 0.4% in placement quality with 41% reduction (1.7x faster) in CPU time, compared to VPlace. Furthermore, the performance of GSA is consistent for both medium and large circuits, which is neccesary for a robust adaptive heuristic [Betz99].

Both VPlace and GSA can generate placement output files that can be fed to VPR's routing tool to perform subsequent FPGA Routing. According to the

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**2870**	2903	2859	9	**3**	3	3	0
tseng	**9479**	9685	9287	96	**20**	22	19	1
ex5p	**16304**	16541	16187	83	**18**	19	17	1
alu4	**19435**	19621	19215	116	**30**	33	29	2
seq	**24871**	25241	24763	104	**40**	43	36	3
M.avg	**17522**	17772	17363	100	**27**	29	25	2
frisc	**52347**	53452	51694	352	**125**	138	115	7
spla	**60642**	62115	59617	755	**138**	151	130	7
ex1010	**65946**	67321	65441	380	**196**	216	189	10
s38584.1	**65787**	66506	64792	523	**329**	338	315	22
clma	**140872**	142433	139847	807	**579**	594	539	27
L.avg	**77119**	78365	76365	563	**273**	287	257	15
Avg	**45855**	46582	45441	322	**148**	154	139	8

Table 3.8: Performance of GSA (default) with 20 runs over each benchmark circuits

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**2861**	2877	2851	10	**5**	5	5	0
tseng	**9297**	9367	9232	49	**38**	39	37	0
ex5p	**16193**	16226	16144	31	**34**	38	30	3
alu4	**19257**	19376	19169	78	**60**	61	59	1
seq	**24745**	24776	24707	29	**76**	81	73	4
M.avg	**17373**	17437	17313	47	**52**	55	50	2
frisc	**51684**	52197	51277	409	**245**	268	227	15
spla	**60082**	61096	59373	704	**258**	266	245	8
ex1010	**65377**	65742	65141	237	**374**	398	360	17
s38584.1	**64870**	65747	64323	642	**585**	616	553	28
clma	**139709**	140663	139232	643	**1026**	1075	986	38
L.avg	**76344**	77089	75869	527	**498**	525	474	21
Avg	**45408**	45807	45145	283	**270**	285	257	11

Table 3.9: Performance of GSA with innerNum = 10

Circuit name	**Ave.cost impro.%**	Max.cost impro.%	Min.cost impro.%	**Ave.CPU t impro.%**	Max CPU t impro.%	Min CPU t impro.%
e64	**-0.49**	+0.07	-0.85	**+76.92**	+76.92	+76.92
tseng	**-0.53**	-1.64	+1.09	**+73.09**	+72.60	+72.46
ex5p	**-0.33**	-0.51	-0.51	**+74.21**	+73.24	+75.36
alu4	**-1.42**	-2.10	-0.82	**+71.21**	+71.03	+70.59
seq	**-0.52**	-0.82	-0.20	**+72.50**	+70.83	+72.99
M.avg	**-0.70**	-1.27	-0.11	**+72.75**	+71.93	+72.85
frisc	**-0.41**	+0.45	-1.06	**+68.14**	+66.42	+69.71
spla	**+0.53**	-0.14	+1.46	**+67.42**	+66.43	+67.82
ex1010	**-0.62**	-0.75	-0.42	**+64.43**	+62.26	+65.38
s38584.1	**-1.15**	-0.66	-1.58	**+64.09**	+63.92	+64.60
clma	**-0.53**	-0.04	-0.53	**+56.92**	+55.37	+59.37
L.avg	**-0.40**	-0.23	-0.43	**+64.20**	+62.88	+65.38
Avg	**-0.53**	-0.61	-0.34	**+69.28**	+67.90	+69.52

Table 3.10: Comparison between VPlace and GSA with innerNum = 5

Circuit name	**Ave.cost impro.%**	Max.cost impro.%	Min.cost impro.%	**Ave.CPU t impro.%**	Max CPU t impro.%	Min CPU t impro.%
e64	**-0.10**	+0.17	-0.39	**+61.54**	+61.54	+64.85
tseng	**+1.03**	+0.76	+1.42	**+46.18**	+46.58	+46.38
ex5p	**+0.21**	+0.63	-0.08	**+51.29**	+46.48	+56.52
alu4	**-0.50**	-0.76	-0.54	**+42.42**	+42.99	+42.16
seq	**-0.03**	-0.07	-0.03	**+46.40**	+43.75	+46.72
M.avg	**+0.18**	+0.14	+0.19	**+46.57**	+44.95	+47.94
frisc	**+0.90**	+1.40	+0.54	**+37.56**	+33.32	+40.73
spla	**+1.58**	+0.70	+1.90	**+37.74**	+36.67	+39.36
ex1010	**+0.18**	-0.10	+0.34	**+32.47**	+29.81	+33.70
s38584.1	**+0.09**	-0.41	+0.09	**+35.36**	+33.26	+37.66
clma	**+0.49**	+0.85	+0.04	**+23.00**	+20.31	+23.98
L.avg	**+0.65**	+0.49	+0.58	**+33.23**	+30.68	+35.08
Avg	**+0.38**	+0.32	+0.33	**+41.40**	+39.47	+43.20

Table 3.11: Comparison between VPlace and GSA with innerNum = 10

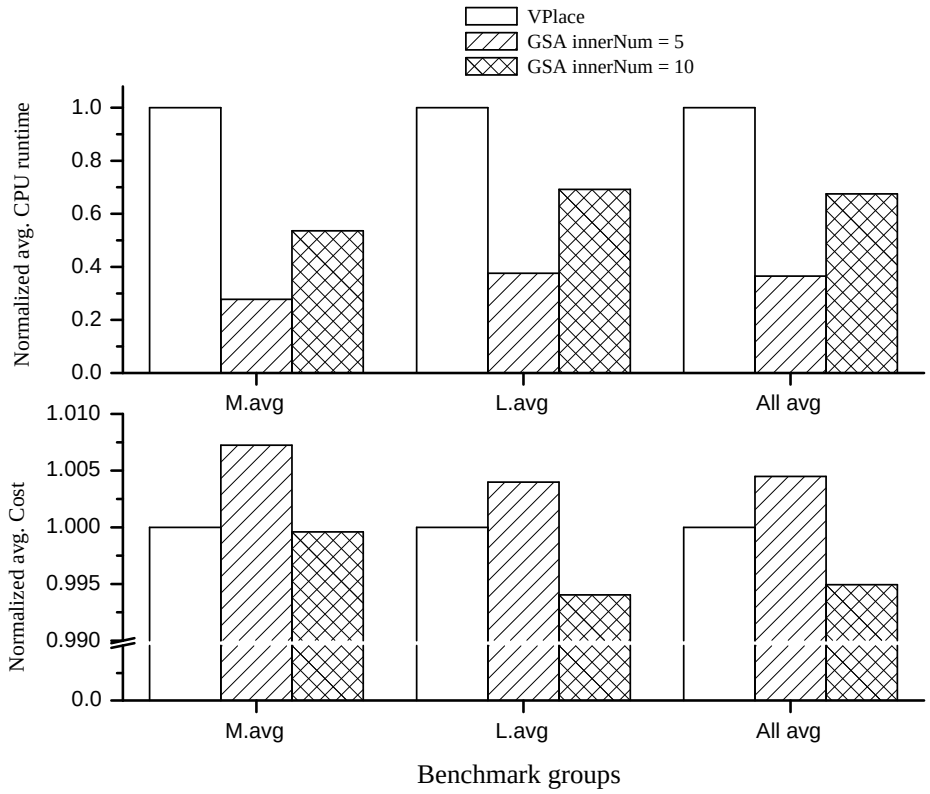

Figure 3.19: Normalized (with respect to the results obtained by VPlace) grouped benchmark performance comparison among VPlace and two versions of GSA.

output of VPR's routing tool, the routing quality[10] is not sensitive to the difference in placement quality of less than 1%. Consequently, 5 is chosen as the default $innerNum$ in GSA, this represents a good compromise between the placement quality and CPU time.

[10]The quality of FPGA routing in VPR is measured in terms of minimum number of tracks required to route the circuit on the target FPGA chip.

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**2881**	2896	2873	10	**3**	3	3	0
tseng	**9476**	9551	9372	69	**19**	20	18	1
ex5p	**16317**	16384	16246	55	**18**	19	17	1
alu4	**19520**	19659	19313	165	**30**	31	30	1
seq	**25165**	25607	24938	256	**39**	41	37	2
M.avg	**17620**	17800	17467	136	**27**	28	26	1
frisc	**52572**	53199	52139	461	**124**	133	115	4
spla	**60810**	61520	60423	428	**134**	139	128	6
ex1010	**65831**	66708	65378	522	**196**	211	187	9
s38584.1	**65669**	66007	65377	275	**315**	327	302	12
clma	**142191**	144464	140247	1302	**549**	587	520	29
L.avg	**77415**	78380	76713	598	**264**	279	250	12
Avg	**46043**	46600	45631	354	**143**	151	136	6

Table 3.12: Performance of GSA with large fanout nets removed

3.8 Large Fanout Nets Elimination

A possible enhancement that can speed up the placement is to ignore nets with large fanout [Sank99]. This is useful because a high-fanout net will likely cover much of the FPGA and so it is harder to reduce its wirelength. By ignoring nets above a certain fanout threshold, called $maxfanout$, the placement problem is simplified. However, if this threshold is set too low, there may lack sufficient information to create a good placement. In our implementation, the value of maxfanout is set to one tenth of the total number of CLBs and I/O pads in the circuit.

Initially, all nets with fanout greater than maxfanout are removed from the netlist. Next, GSA (default) was run on the simplified netlist. When GSA placement terminated, the ignored nets are reinstated. Table 3.12 shows the performance of GSA with large fanout nets removed. The placement cost as well as CPU time is compared to that obtained when the original circuit was placed (with no nets ignored). Table 3.13 shows the comparison results.

Circuit name	**Ave.cost impro.%**	Max.cost impro.%	Min.cost impro.%	**Ave.CPU t impro.%**	Max CPU t impro.%	Min CPU t impro.%
e64	**-0.31**	-0.56	-0.31	**0**	0	0
tseng	**-0.34**	+0.45	-1.18	**0**	0	+5.26
ex5p	**-0.23**	+0.17	-0.20	**0**	0	0
alu4	**-0.44**	-0.11	-0.47	**0**	0	0
seq	**-1.20**	-2.59	-0.76	**0**	+2.38	0
M.avg	**-0.55**	-0.52	-0.65	**0**	+0.60	+1.32
frisc	**-0.39**	-0.95	-0.07	**+0.80**	+1.48	+0.86
spla	**-0.14**	+0.15	-1.32	**+0.74**	+1.42	+1.54
ex1010	**+0.10**	-0.81	+0.39	**+0.51**	+1.40	+0.53
s38584.1	**+0.01**	-0.15	+0.03	**+3.08**	+1.80	+3.82
clma	**-0.93**	-1.78	-0.16	**+4.36**	+2.49	+1.33
L.avg	**-0.27**	-0.71	-0.23	**+1.90**	+1.72	+1.62
Avg	**-0.39**	-0.62	-0.41	**+0.95**	+1.10	+1.33

Table 3.13: Comparison between two GSA with and without large fanout nets elimination mechanism.

From Table 3.13, we can conclude that very little (less than 1%) CPU time is saved, while the quality of placement cost, on average, deteriorates by 0.4%. When compared with the conclusion of Section 3.7.3, where we doubled the runtime to obtain 1% gain of placement quality, the loss of cost (0.4%) in this method is expensive (as CPU time reduction is very small). In addition, the placement quality will be further deteriorated if we reduce the fanout threshold maxfanout (as more nets are removed during a placement, which prevents a good solution to be obtained). On the other hand, it seems that very little reduction of runtime can be achieved, if maxfanout is increased to trade CPU time for quality. Therefore, the large fanout nets elimination method is not implemented in our high-quality oriented placement tools.

3.9 Summary

In this chapter, simple local search methods are implemented and evaluated. Then a new algorithm named GSA for FPGA placement is described. The algorithm is capable of overcoming the problem of slow convergence in simulated-annealing. This novel heuristic incorporates some of the best features from existing methods. It operates in a greedy manner to accelerate the entire search procedure. A detailed methodology and the associated adaptive update schema, as well as the empirical parameters have been provided. Some placement-related issues are evaluated such as strategically changing the window limiter, which proves to be very effective speeding up the convergence. It is shown that compared to VPlace, our heuristic yields very promising final quality while drastically reduceing the CPU time.

In the next chapter, we investigate another method for tackling this complicated problem in FPGA physical design. A multi-level circuit clustering method is utilized to reduce the complexity of the placement task at each level. Various algorithms are employed to achieve improvement at each clustering level, while our GSA heuristic still plays an important role in the overall placement process.

Chapter 4

Hierarchical Approach

In this chapter, a hierarchical FPGA placement algorithm is implemented and evaluated. The detailed implementation strategy and methodology of the approach are described, including the intergration of techniques and parameter tuneup which lead to the best solution quality and CPU time trade-off.

The GSA based placement tool, which displayed robustness and speed in convergence, plays an important role in the hierarchical approach. Furthermore, a novel adaptive method for the determination of proper initial parameters for the simulated-annealing algorithm is provided to assist a smooth transition between the different algorithms at different hierarchical levels. Finally, the performance of the hierarchical placement tool is compared to both the "flat" GSA placement tool, as well as the existing known high-quality FPGA placement tool VPlace [Betz99] [Betz97b].

4.1 Overview of Hierarchical Approach

FPGA hierarchical placement, roughly, is a two step procedure: first proceeding bottom-up clustering, then top-down improvement.

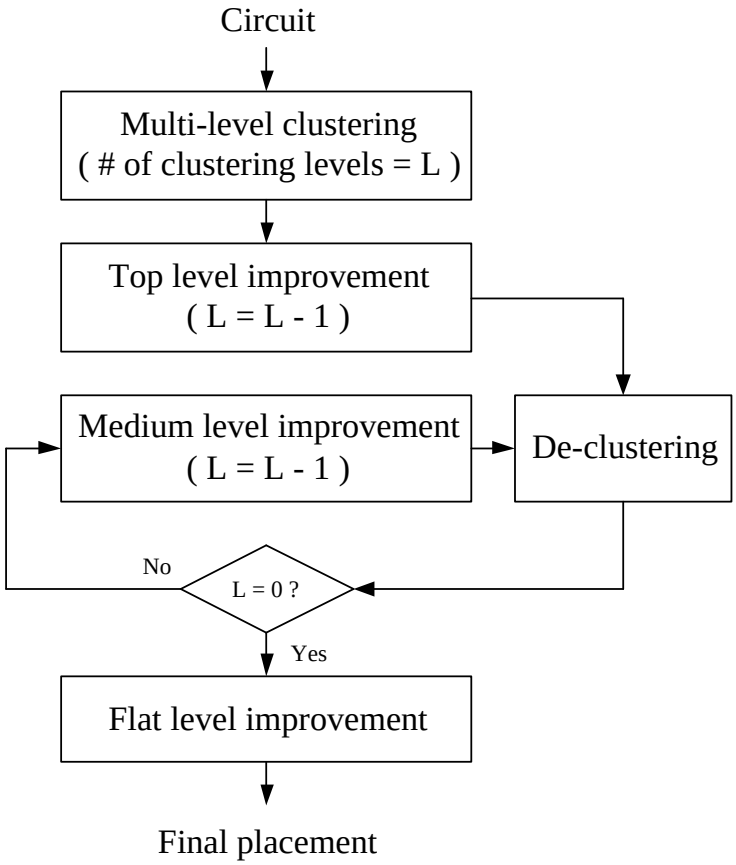

Figure 4.1: Framework of our hierarchical placement algorithm

Figure 4.1 illustrates the framework of our algorithm. The first stage is a multi-level bottom-up clustering of the logic blocks, which groups highly connected blocks into clusters; these clusters can be grouped again to create a second level of clustering, and so on. Once all the required clustering is done, placement will be performed at each level of the hierarchy to obtain further 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. As the most simplified clustering level, improvement of the top level is of great importance and is listed separately in Figure 4.1. The de-clustering or flattening process, which decomposes clusters into logic blocks or clusters of lower levels, will be followed when the top-down optimization at each level is completed. Finally, a localized heuristic, which only moves blocks in small regions, will be performed at the bottom (flat) clustering level to achieve a high quality final solution.

As we have described in Chapter 2, multi-level clustering has been used as an effective way to speed up the procedure of circuit placement and partitioning problems. Furthermore, as some FPGAs that have 40-million effective gates on the horizon, a hierarchical approach is probably the ultimate method that can greatly reduce placement time without compromising the quality of solution. These served as the major motivation for us to explore the performance of this approach.

4.2 Clustering-based FPGA Placement

4.2.1 Clustering Method

Clustering serves as the first processing stage in a hierarchical approach, which groups blocks together to reduce the complexity of the placement problem. A good clustering method should identify batches of blocks that will eventually end up together in the final placement stage. This can be difficult because clustering decisions are made prior to the start of the placement without a global view of the circuit structure. Therefore, a top-down global circuit partitioning methodology

has usually been preferred to a bottom-up clustering methodology. But the sizes of today's circuits are so large, for which the associated heavy computation makes any top-down partitioning scheme infeasible. Therefore, only an effective bottom-up clustering approach is the choice for a hierarchical approach [Sun95].

However, for clustering to be used as a practical bottom-up approach, there are two important concerns.

1. **The computation time used to generate clusters:** It is recommended that the clustering time be negligible in comparison to the time required to place the circuits.

2. **The quality of clusters (presented in Section 2.6.1):** It is measured in terms of the percentage of nets that would be completely *absorbed* by a clustering method resulting in a single level of clustering.

The main goal of [Sank99] is to provide an FPGA placement prediction tool that would give a solution ultra fast with affordable loss of the final placement quality. The major objective of our work in this thesis is to obtain high-quality placement within short amount of CPU time. Sankar's clustering method (described in Section 2.6.3) is quite attractive, which produces high quality clusters very quickly. Unlike the time-consuming simulated-annealing based optimization used in Timberwolf95, it is a greedy constructive method, which enables the clustering procedure to complete quickly with reasonable loss of clustering quality. Consequently, this part of our hierarchical approach is based on the method in [Sank99].

Figure 4.2 illustrates how clustering size influences the net absorption rate resulting from a single level of clustering. Note that I/O pads are not included in clusters so that any nets share connections with pads will not be absorbed. Fine

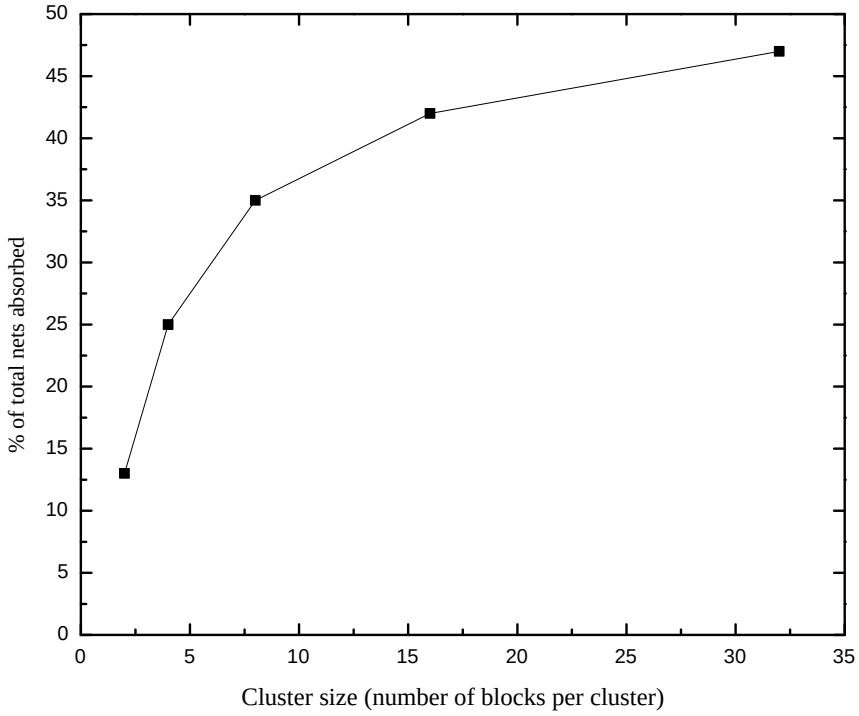

Figure 4.2: Cluster size vs. % of total nets absorbed of Sankar's [Sank99] clustering method resulting in one level of clustering (data are obtained averagely over 10 MCNC benchmark circuits).

grained clusters (between 2 and 16 logic blocks per cluster) are mainly used in this thesis. Even for these relative small size clusters, a significant proportion of flat nets are collapsed (13% to 47%). The runtime of this clustering method has to be compared with the whole placement procedure, which will be determined in later sections.

4.2.2 De-clustering and Legalization

De-clustering or *flattening* is a procedure used to restore those "collapsed" blocks in the higher hierarchical level as well as their interconnections.

Some deterioration of the solution is expected. It is partly because those nets that were "localized" (absorbed) to each cluster must be added back to a flattened circuit, which would increase the total wirelength. Another factor that contributes to the increase of wirelength is the approximation of cluster location. In our implementation, the coordinate of one of the blocks in each cluster is employed as the coordinate of the cluster, which is an estimate for the true localities in the eventual placement for other blocks in this cluster. When the clusters are broken down, this location difference is exposed, and the overall solution gets worse as well. This approximation error increases as the clustering size increases, for which the difference between the "real" block positions and the coordinate of the clusters are greater. Consequently, fine grained clusters are preferred in our work to obtain high quality placement.

Except for a few clusters, most clusters are created with an identical number of blocks. The de-clustering procedure does not change the original layout of the target FPGA chip. For those marginal irregular clusters, an additional legalization procedure, which automatically adjusts their locations to fit the chip, will be followed after de-clustering.

Some previous methods, like [Sun95], flatten a clustered circuit simply by placing the blocks randomly within the physical confinements of the clusters in the previous level. Recent research in [Sank99] and [Thom01] has shown that performing this flattening procedure by a constructive or greedy optimization method

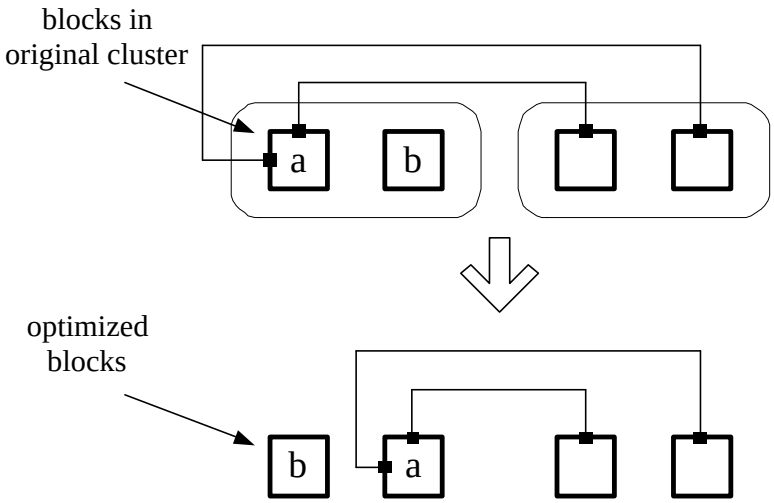

Figure 4.3: Blocks in original clusters are optimized to minimize wirelength during de-clustering.

would produce superior results. The benefit of this optimization is illustrated in Figure 4.3. Block a has more connections from the right side or its "center-of-gravity" is at that side. It is natural for block a to be placed at the right slot of the original cluster to reduce the wirelength. However, the "center-of-gravity" of each block depends on the locations of blocks in other clusters. Therefore, it only can be determined when the top-down optimization at this level is completed (when all locations of clusters have been determined).

In our implementation, a clustered circuit is initially flattened by placing the blocks randomly within the physical confinements of the clusters that host them in the previous level. Next, a simple optimization process is performed within the confinement of each cluster, such that the blocks which belong to the same cluster are allowed to be swapped. A simple deterministic local search heuristic, shown in Figure 4.4, is introduced to complete this part of de-clustering in our hierarchical approach. Generally, deterministic local search algorithms are not appropriate for

```
    /* assuming there are N clusters (from 1 to N) need to be optimized */
1.  S = InitPlacement();
2.  for(i = 1; i ≤ N; i++)
    /* blocks from m to n are included in cluster_i */
    /* cluster_i ∋ (block_m ~ block_n) */
3.  {       for(j = m; j ≤ n - 1; j++)
4.      {       for(k = m + 1; k ≤ n; k++)
5.          {       S_candidate = Swap(block_j, block_k);
                    /* find a candidate solution from current one */
                    /* by a determined swap between block_j and block_k */
6.                  ΔC = Cost(S_candidate) - Cost(S_current);
7.                  if(Δ C < 0)    /* only accept improving swaps */
8.                      S_current = S_candidate;  /* accept this candidate */
9.          }
10.     }  /* end of optimization for one cluster */
11. }  /* end of de-clustering optimization at this level */
    /* output optimized placement */
```

Figure 4.4: Pseudo-code for de-clustering optimization

problems that have a large number of neighbouring solutions, since the exhaustive evaluation would consume too much CPU time. However, as mentioned before, we are more interested in fine grained clusters, for which the number of neighbouring solutions is small. Thus this heuristic can be finished in short amount of time.

4.2.3 Evaluating the Effectiveness of Clustering, De-clustering and Simple Local Search in the Hierarchical Approach

The effectiveness of clustering and de-clustering optimization, as well as simple local search in the hierarchical approach is evaluated. The evaluation is based on the following : clustering level $L = 2$ and clustering size (blocks per cluster) at each level $S = 4$.

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**7452**	7984	7273	318	**0**	0	0	0
tseng	**41285**	44789	39271	573	**0.02**	0.02	0.02	0
ex5p	**42301**	45376	40589	623	**0.02**	0.02	0.02	0
alu4	**61504**	63891	60812	967	**0.08**	0.08	0.08	0
seq	**79903**	81045	78734	1463	**0.03**	0.03	0.03	0
M.avg	**56248**	58775	54852	906	**0.04**	0.04	0.04	0
frisc	**229152**	236421	225510	2355	**0.19**	0.20	0.19	0
spla	**236251**	243011	233721	1987	**0.11**	0.11	0.11	0
ex1010	**332664**	337701	329438	3101	**0.21**	0.23	0.20	0
s38584.1	**559870**	564933	552789	4754	**0.38**	0.40	0.37	0
clma	**796591**	807265	792277	10647	**0.52**	0.58	0.49	0
L.avg	**430906**	437866	426747	4569	**0.28**	0.30	0.27	0
Avg	**238697**	243242	236041	2679	**0.16**	0.17	0.15	0

Table 4.1: Random clustering and random de-clustering ($L = 2, S = 4$)

Results in Table 4.1 are obtained from randomly assigning blocks to cluster at each level to create the hierarchy. Following clustering a random de-clustering phase is performed, which places the blocks randomly within the physical confinements of the clusters. Table 4.2 shows the performance of optimized clustering only. Results are obtained by performing optimized clustering at each level followed by random de-clustering. Table 4.3 shows the performance of optimized clustering and optimized de-clustering at each hierarchical level. Table 4.4 shows the improvement of solution quality of optimized clustering itself, and optimized clustering and de-clustering in our hierarchical approach.

These results illustrate that the optimized clustering itself and optimized clustering with optimized de-clustering procedure improve the original random start placement, on average by 29% and 33% respectively. Furthermore, both the clustering and de-clustering process can be completed very quickly.

Table 4.5 shows the performance of simple non-deterministic local search (pre-

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**6271**	6441	6130	239	**0.01**	0.01	0.01	0
tseng	**25431**	26011	24891	326	**0.09**	0.09	0.08	0
ex5p	**32653**	34218	32186	479	**0.09**	0.09	0.09	0
alu4	**42550**	43072	41857	630	**0.22**	0.23	0.22	0
seq	**59268**	60027	58792	592	**0.13**	0.13	0.13	0
M.avg	**39976**	40832	39432	507	**0.13**	0.14	0.13	0
frisc	**152999**	154872	152185	1559	**0.58**	0.61	0.56	0.01
spla	**169741**	171133	168534	884	**0.35**	0.36	0.34	0
ex1010	**250873**	253964	249836	2174	**0.67**	0.74	0.63	0.02
s38584.1	**335429**	338347	333928	2392	**3.40**	3.71	3.20	0.11
clma	**548932**	551023	541938	9573	**2.79**	2.91	2.64	0.07
L.avg	**291595**	293868	289284	3316	**1.56**	1.67	1.47	0.04
Avg	**162415**	163911	161028	1885	**0.83**	0.89	0.79	0.02

Table 4.2: Optimized clustering and random de-clustering ($L = 2, S = 4$)

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**5743**	5932	5649	199	**0.01**	0.01	0.01	0.01
tseng	**23378**	23745	22849	312	**0.09**	0.09	0.08	0
ex5p	**29856**	30202	29710	328	**0.10**	0.10	0.09	0
alu4	**39624**	40246	39037	532	**0.23**	0.23	0.23	0
seq	**55571**	56028	55019	491	**0.14**	0.14	0.14	0
M.avg	**37107**	37530	36679	416	**0.14**	0.14	0.14	0
frisc	**144279**	145101	143729	1039	**0.61**	0.64	0.56	0.01
spla	**159422**	161138	158587	782	**0.39**	0.41	0.36	0
ex1010	**240335**	242021	238937	1077	**0.72**	0.74	0.68	0
s38584.1	**320052**	322398	318834	1590	**3.46**	3.73	3.38	0.09
clma	**528772**	532182	525182	4171	**2.87**	2.95	2.74	0.05
L.avg	**278572**	280568	277054	1732	**1.61**	1.69	1.54	0.04
Avg	**154703**	155889	153763	1052	**0.86**	0.90	0.83	0.02

Table 4.3: Optimized clustering and optimized de-clustering ($L = 2, S = 4$)

Circuit name	Random clustering & random de-clustering		Optimized clustering & random de-clustering			Optimized clustering & optimized de-clustering		
	Ave. cost	Ave. CPU t.	Ave. cost	Ave. CPU t.	Cost impro.%	Ave. cost	Ave. CPU t.	Cost impro. %
e64	7452	0	6271	0.01	15.85	5743	0.01	22.93
tseng	41285	0.02	25431	0.08	38.40	23378	0.09	43.37
ex5p	42301	0.02	32653	0.09	22.81	29856	0.09	29.42
alu4	61504	0.08	42550	0.22	30.82	39624	0.23	35.57
seq	79903	0.03	59268	0.13	25.83	55571	0.14	30.45
M.avg	56248	0.04	39975	0.14	29.46	37107	0.14	34.71
frisc	229152	0.19	152999	0.58	33.23	144279	0.61	37.04
spla	236251	0.11	169741	0.35	28.15	159422	0.39	32.52
ex1010	332664	0.21	250872	0.67	24.59	240335	0.72	27.75
s38584.1	559870	0.38	335429	3.40	40.09	320052	3.46	42.83
clma	796591	0.52	548932	2.79	31.09	528772	2.87	33.62
L.avg	430905	0.28	291594	1.56	31.43	278572	1.61	34.75
Avg	238697	0.15	162414	0.83	29.08	154703	0.86	33.55

Table 4.4: Solution improvement of optimized clustering and optimized de-clustering ($L = 2, S = 4$)

sented in Section 3.3). The local search technique is performed at each hierarchical level together with full-fledged optimized clustering and de-clustering. Table 4.6 shows the improvement in placement cost by performing optimized clustering, optimized de-clustering, and simple local search at each hierarchical level. The overall improvement of these three optimization methods, with respect to no optimization being made, is on average 61%.

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**3816**	3968	3673	118	**0.23**	0.23	0.22	0
tseng	**13938**	14277	13650	272	**0.73**	0.75	0.72	0
ex5p	**20713**	20992	20374	222	**0.71**	0.71	0.70	0
alu4	**26134**	26837	25595	483	**0.93**	0.95	0.93	0.01
seq	**36418**	37054	35889	474	**1.21**	1.21	1.20	0
M.avg	**24301**	24790	23877	363	**0.90**	0.91	0.89	0
frisc	**88342**	90513	85540	1865	**3.33**	3.36	3.30	0.02
spla	**96534**	97886	95457	984	**3.34**	3.42	3.23	0.07
ex1010	**99941**	103128	97717	2001	**4.59**	4.74	4.22	0.21
s38584.1	**142213**	145573	138637	2657	**10.19**	10.48	9.28	0.51
clma	**262927**	277606	252673	9637	**14.15**	14.97	13.62	0.88
L.avg	**137991**	142941	134005	3429	**7.12**	7.39	6.73	0.34
Avg	**79098**	81783	76921	1871	**3.94**	4.08	3.74	0.17

Table 4.5: Performance of simple local search in hierarchial approach ($L = 2, S = 4$)

4.3 Improvement Techniques Implemented at Each Hierarchical Level

4.3.1 Top Level Improvement

A large portion of iterative improvement should be done at the highest level of the hierarchy, taking advantage of the reduced solution space facilitated by clustering. Ideally, a clustering method will convert the original circuit into a similar but much smaller structure, with many entities combined into one large cluster. Blocks in a cluster (as they are decomposed during the de-clustering procedure), should be close to where they would be in a high-quality flat placement.

Nevertheless, the exact final location of each block is unknown at the clustering period, where an estimated value based on the connectivity is used instead. The potential of the improvement is over shadowed by this rough calculation. Additionally, the succeeding de-clustering procedure would partly "destroy" (described in

Circuit name	Random clustering & random de-clustering		Non-deterministic simple local search		
	Ave. cost	Ave. CPU t.	Ave. cost	Ave. CPU t.	Cost impro.%
e64	7452	0	3816	0.23	48.79
tseng	41285	0.02	13938	0.73	66.24
ex5p	42301	0.02	20713	0.71	51.03
alu4	61504	0.08	26134	0.93	57.51
seq	79903	0.03	36418	1.21	54.42
M.avg	56248	0.04	24301	0.90	57.30
frisc	229152	0.19	88342	3.33	61.45
spla	236251	0.11	96534	3.34	59.14
ex1010	332664	0.21	99941	4.59	69.96
s38584.1	559870	0.38	142213	10.19	74.60
clma	796591	0.52	262927	14.15	66.99
L.avg	430905	0.28	137991	7.12	66.43
Avg	238697	0.15	79098	3.94	61.01

Table 4.6: Solution improvement of simple local search with optimized clustering and optimized de-clustering ($L = 2, S = 4$)

Section 4.2.2) the solutions obtained from higher levels as well. An excellent solution in the high level is not guaranteed to be translated into a matching excellent solution in the flat level, even with well-optimized clustering and de-clustering.

The circuit at the top level of clustering is considered to be the least complex representation of the original circuit, and therefore a good heuristic technique such as GSA should be able to place clusters efficiently in a short period of time. Similar to a global placement tool in standard cell placement, the GSA placement tool behaves in a top-down manner on the top level, roughly determining the best position of each block. Medium and bottom level improvements would follow to find exactly where each block should be for the final high-quality placement.

4.3.2 Medium Level Improvement

When the total number of hierarchical levels is more than two, extra improvement should be performed at the intermedium levels. With proper top level placement, blocks should have been assigned their proper locations. However, the placement entities are still clusters in these medium levels, even though they are smaller than the clusters in the top level. The major objective in the medium level is to gradually improve the solution quality. A simple yet effective stochastic method, non-deterministic local search algorithm, is used here to complete this task. The pseudo-code as well as the detailed implementation and evaluation of this local search algorithm are provided in Section 3.3 and Section 3.5, respectively.

4.3.3 Bottom Level Improvement

Fine tuned search should be performed at the flat level of the hierarchy, since it is an accurate representation of the actual circuit (all absorbed nets are released and all locations of each block are accurate). Utilizing appropriate heuristic techniques at the top and medium levels, blocks in this level should be placed close to their final location; a localized placement tool is therefore preferred. Simple local search algorithms, as has been discussed earlier converge quickly. However, they can easily get trapped in local optimal, which can be too far from the optimal solution.

A low temperature VPlace simulated-annealing algorithm is therefore utilized on the flattened circuit to optimize the final local ordering of the logic blocks. With the help of the parameter R_{limit}, which shrinks when temperature T becomes low, VPlace behaves in a bottom-up manner by placing blocks in smaller and smaller regions. Furthermore, this dynamic well-tuned heuristic guarantees to find a high-

quality solution for the whole hierarchical approach.

4.3.4 Choice of Starting Parameters for Simulated-annealing

One crucial feature of any dynamic adaptive annealing schedule for a variety of circuits is the choice of starting parameters for a given good configuration. In our hierarchical approach, this good configuration is the placement result from the improvement in the higher levels. Parameter tuning, in VPlace simulated-annealing, involves the start temperature T_0 and the initial size of window limiter R_{limit}, which are crucial in controlling the behavior of a placement tool (described in section 2.4.4). If T_0 is set too high or R_{limit} is set too large, subsequent annealing will destroy the existing good placement structure, which makes any previous work toward placing the circuit useless. Conversely, if T_0 is set too low or R_{limit} is set too small, the annealer is unlikely to improve upon the existing placement significantly, as it will be unable to escape local minima.

In [Rose90], a concept, which is proposed to compute a good starting temperature T_0 for simulated-annealing placements, is introduced. The idea evolves around an initial temperature where the placement is in a state of "equilibrium". In this state, there is no expected net change in the cost function after a large number of moves, which implies that the expected change in placement cost is zero.

Unfortunately, the parameter R_{limit} is not considered in their work[1], which is so important and should not be neglected. We propose a new method to calculate this equilibrium state. It resembles the annealing schedule of VPlace with the exception of having less number of moves attempted at each temperature. The pseudo-code

[1]The work of [Sank99], which utilize the method of [Rose90], did not consider R_{limit} either.

for this method is shown in Figure 4.5.

Starting from an initial good placement, this method performs N evaluations at each temperature, none of which are actually permitted to change the placement. The criteria of accepting a move is based on the following: $r < e^{-\Delta C/t}$ where r is a random number between 0 and 1. If a move is "accepted", the change in cost associated with this move is recorded and accumulated in the parameter $V_{accumulate}$. Accepting a bad move increases the value of $V_{accumulate}$, while an improvement move, which is always "accepted", would reduce it. At the end of each N evaluations (inner loop in Figure 4.5), the value of $V_{accumulate}$ is assessed.

When temperature t is high enough, bad moves are accepted with high probability leading to a high value of $V_{accumulate}$. Temperature t and window limiter r_{limit} are updated with the same method as VPlace simulated-annealing algorithm (described in section 2.4.4) to ensure the best combination. As the value of t decreases, the heuristic becomes more and more greedy; favoring moves that would decrease the objective function. Once negative overall changes in placement cost $V_{accumulated}$ is reached, the present temperature t and r_{limit}, which would render the current placement in a state of equilibrium, are set to be the suitable value of initial T_0 and R_{limit}, respectively, for the following actual annealing.

It is important to ensure that enough number of moves per temperature are made to obtain an accurate probability distribution. In our implementation, the value of N is set equal to the value of N_{block}, which is the total number of logic blocks plus the number of I/O pads in a circuit.

```
1.  S_init = InitPlacement();
    /* here the initPlacement S_init is a result generated by */
    /* other algorithms and is better than just random start */
2.  t = InitTemperature();
3.  r_limit = Initr_limit();  /* r_limit is set to whole chip initially */
4.  V_accumulate = 0;  /* V_accumulate is the exit criterion */
5.  P_parameter(t, r_limit);
    /* P_parameter is a structure to store temperature t and r_limit */
6.  while(ExitCriterion() == false)    /* outer loop */
7.  {       while(InnerLoopCriterion() == false)    /* inner loop */
8.      {       S_candidate = GenerateMove(S_init, r_limit);
                /* create one candidate solution from current one by */
                /* a random pairwise moves within a window r_limit */
9.              ΔC = Cost(S_candidate) - Cost(S_init);
                /* calculate the change in cost */
10.             r = random(0, 1);
                /* create a random float number between (0, 1) */
11.             if(r < e^(-ΔC/t))
12.                 V_accumulate = V_accumulate + Δ C;
                /* add accepted candidate ΔC incrementally to V_accumulate */
13.     }   /* end of inner loop */
14.     if(V_accumulate ≤ 0)
15.     {       exit();  /* end of program */
                /* export t and r_limit that stored in P_parameter to the following */
                /* simulated annealing placer as the start T_0 and R_limit */
16.     }
17.     else
18.     {       Update(P_parameter(t, r_limit));
                /* record current t and r_limit in P_parameter */
19.             Update(t);
20.             Update(r_limit);
                /* the update of t and r_limit are the same as VPlace */
21.             V_accumulate = 0;  /* start a new round of evaluation */
22.     }
23. }   /* end of outer loop */
```

Figure 4.5: Pseudo-code for choosing start temperature T_0 and initial R_{limit} for simulated-annealing algorithm, which begins with a good initial placement

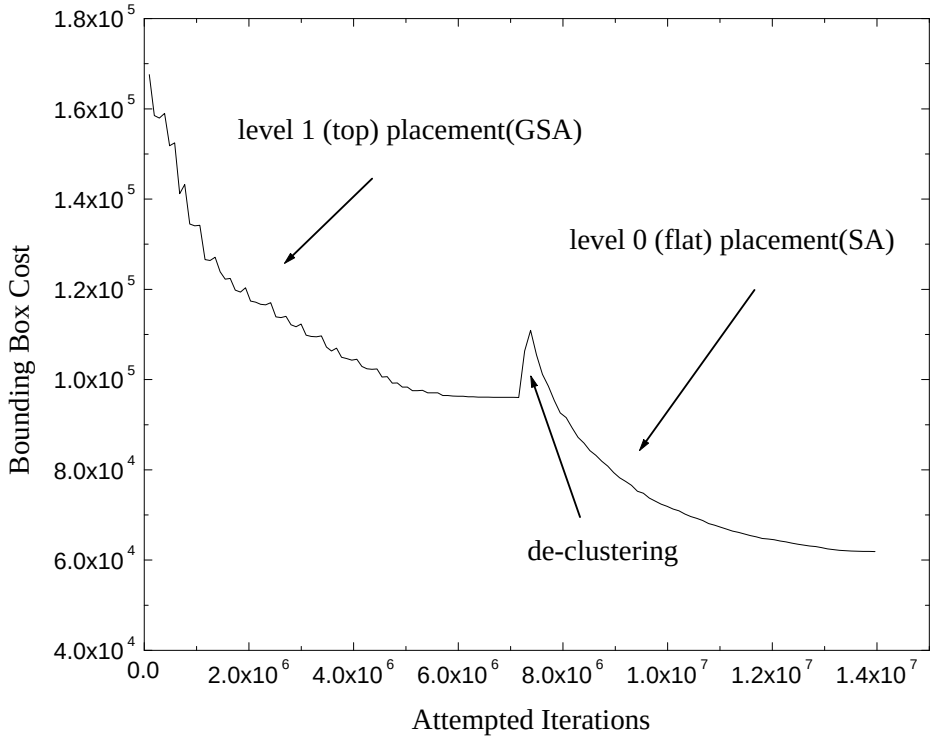

Figure 4.6: Hierarchical approach behavior over a large MCNC circuit "*spla*" (3690 CLBs).

4.3.5 Hierarchical Approach Behavior

Figure 4.6 illustrates the overall behavior of a two-level hierarchical placement tool over a large MCNC circuit. This example begins from a random placement where all blocks have already been collapsed into each cluster. The top level placement is completed by the GSA placement tool, which quickly discovers a good local optimum at this level, such that the total bounding box cost is almost halved. However, this bounding box cost is an approximation of the flat counterpart and they are

"equalized" by the following de-clustering procedure. As expected, the bounding box cost is increased (explained in Section 4.2.2), even with a local optimizer to place the flattened blocks close to their "center-of-gravity" to minimize the jump in cost. The VPlace simulated-annealing algorithm is employed at the flat level with proper initial parameters to ensure a "smooth" transfer between different heuristics.

4.4 Results

In this section, experiments used to identify the proper set of parameters for our placement tool are described first. There are two type of parameters: those that control the cluster formation such as clustering depth (total clustering level) and clustering size (blocks per cluster) and those that control the iterative improvement technique such as $innerNum$ in GSA and VPlace. Our goal is to determine these parameters that lead to the best quality and CPU time trade-off.

Next, the performance of our hierarchical placement tool will be compared with the GSA placement tool (provided in Chapter 3) and VPlace [Betz99] [Betz97b] based on the same MCNC benchmarks.

The details of the actual FPGA architecture and MCNC benchmarks as well as the configuration of our common workstation are given in Section 3.4. Furthermore, all of our results are obtained through 5 runs with different initial placements and random seeds which are more reliable than those based on a single run.

4.4.1 Iterative Improvement Parameter Setting

One of the parameters of the top-level improvement technique (GSA) that needs to be tuned is $innerNum$. With the collapse of blocks and the reduction of their

interconnections, the number of local minima are reduced [Ghen02]. The placement at this level is built on an approximation and would be partly "destroyed" by the following de-clustering. Consequently, a smaller value of $innerNum$ is needed at the top-level compared with a default GSA placement tool.

Several experiments were conducted where the GSA improver was performed at the top-level and then de-clustered to the flat level without any other further improvement. Figure 4.7 plots the average normalized bounding box cost versus different $innerNum$ with different clustering depth and clustering size, across our test benchmark suite. In addition, Figure 4.8 and Figure 4.9 show similar performance of GSA with the same settings for a medium/large MCNC circuit respectively.

It is obvious from Figures 4.7-4.9 that 1 is the right value for parameter $innerNum$ at the top-level GSA placement tool, which results in a slight loss of placement quality (less than 3%) than the default GSA ($innerNum = 5$), while offering better speedup.

A simple local search, which exactly follows the implementation of the non-deterministic local search in Section 3.3, will be performed in the medium levels.

The parameter $innerNum$ (which is the number of iterations performed per temperature T) in the simulated-annealing placement at the bottom level requires further tuning. GSA with the parameters that deduced from previous experiments, is employed at the top-level. Next, a simple non-deterministic local search placement tool will be followed if necessary (Clustering Level > 1). Finally, VPlace with properly determined inital parameters will complete the overall placement. Figure 4.10 plots the average final normalized bounding box cost versus different $innerNum$ with different clustering depth and clustering size, across the test benchmark suite. In addition, Figure 4.11 and Figure 4.12 show similar perfor-

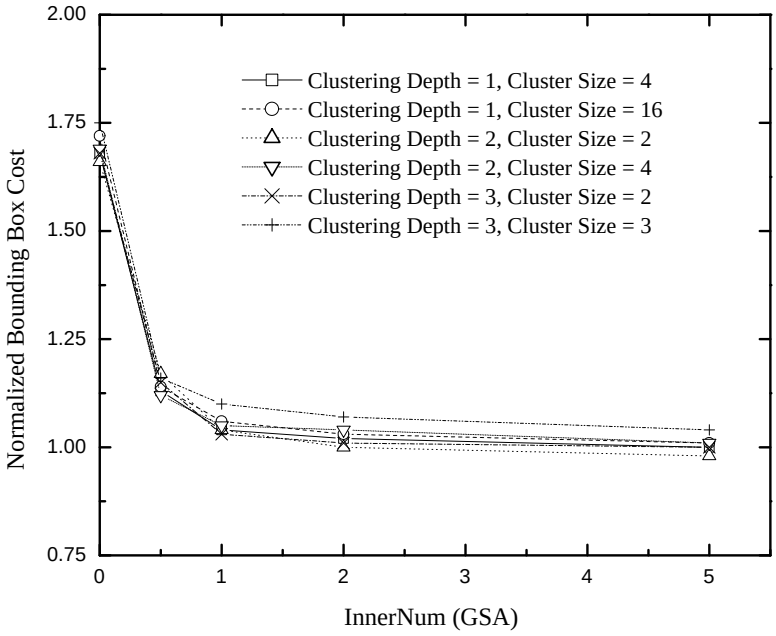

Figure 4.7: Average normalized bounding box cost (with respect to: Clustering Level = 1, Cluster Size = 4, $innerNum = 5$) over 10 MCNC circuits vs. different GSA $innerNum$ with different clustering depth and clustering size.

mance of VPlace with the same settings over a medium and a large MCNC circuit respectively.

Observing the curves in Figure (4.10-4.12), 5 is chosen for the value of $innerNum$ in VPlace which would produce very slight worse final cost (less than 1%), while speeding the bottom-level improvement up to two times [Betz99].

4.4.2 Clustering Parameters

Having determined the right placement tool with proper parameters in each level, the next step that needs to be explored is the clustering property, including clus-

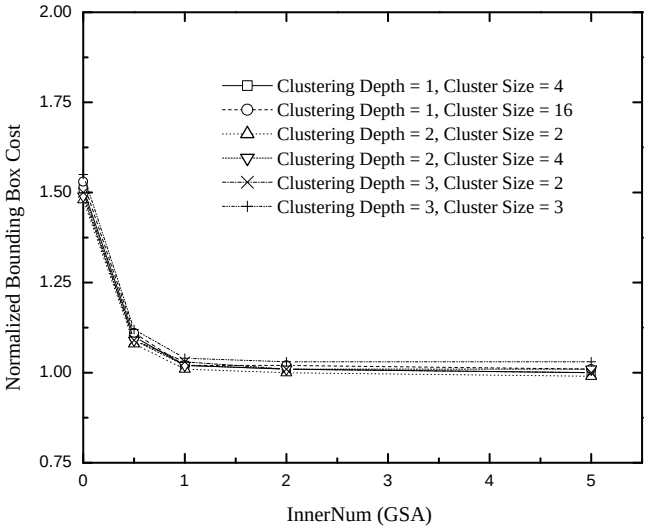

Figure 4.8: Normalized bounding box cost (with respect to: Clustering Level = 1, Cluster Size = 4, $innerNum = 5$) over a medium MCNC circuit "*tseng*" (1047 CLBs) vs. different GSA $innerNum$ with different clustering depth and clustering size.

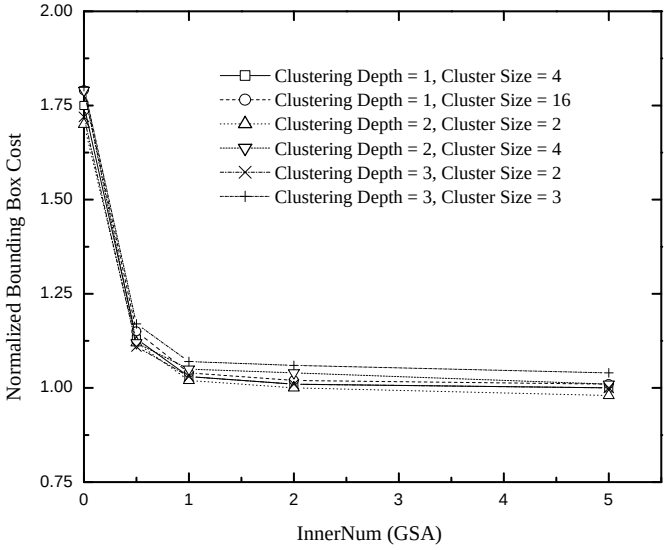

Figure 4.9: Normalized bounding box cost (with respect to: Clustering Level = 1, Cluster Size = 4, $innerNum = 5$) over a large MCNC circuit "*clma*" (8383 CLBs) vs. different GSA $innerNum$ with different clustering depth and clustering size.

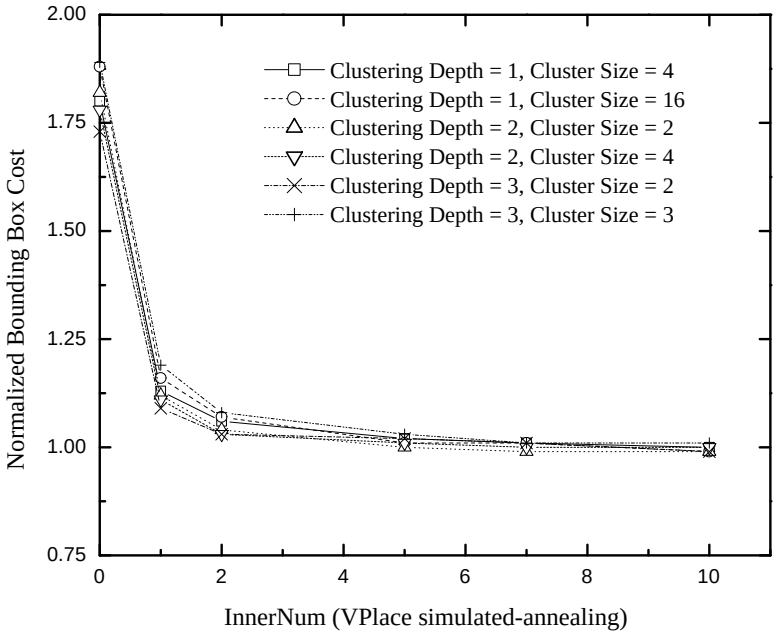

Figure 4.10: Average normalized bounding box cost (with respect to: Clustering Level = 1, Cluster Size = 4, $innerNum = 10$) over 10 MCNC circuits vs. different VPlace $innerNum$ with different clustering depth and clustering size.

tering size or blocks per cluster (S) and clustering depth(L). We performed experiments for $L = 1$, 2, 3 and 4 levels[2] of clustering with different clustering size S. Cluster size S denotes how many blocks per cluster, where these blocks could be CLBs for $L = 1$ or smaller clusters in higher levels when $L > 1$. Additionally, once S is set to n, then n would be the clustering size for every clustering level. The higher a cluster is, the more CLBs it can hold. For example, a cluster at the second level with the clustering formation, $S = 4$, can host 4 smaller clusters at the first level and these smaller clusters each can host 4 CLBs at the flat level.

[2] $L = 0$ denotes the flat level.

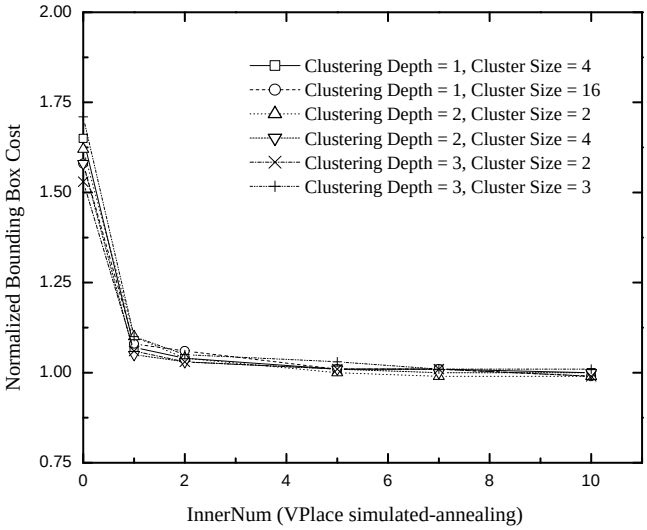

Figure 4.11: Normalized bounding box cost (with respect to: Clustering Level = 1, Cluster Size = 4, $innerNum = 10$) over a MCNC medium circuit "*tseng*" (1047 CLBs) vs. different VPlace $innerNum$ with different clustering depth and clustering size.

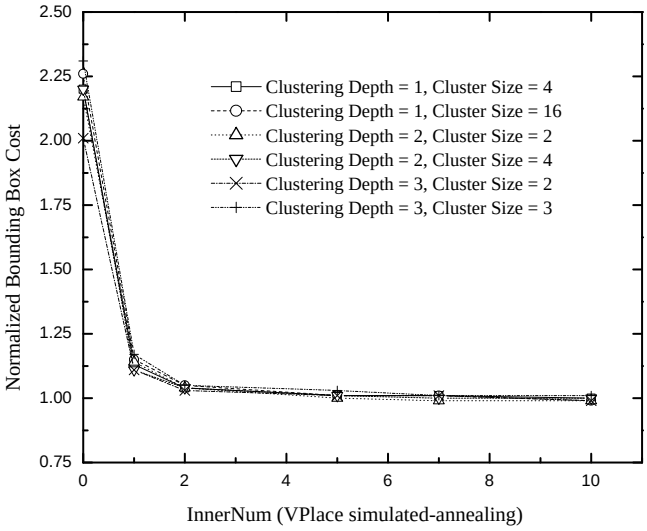

Figure 4.12: Normalized bounding box cost (with respect to: Clustering Level = 1, Cluster Size = 4, $innerNum = 10$) over a large MCNC circuit "*clma*" (8383 CLBs) vs. different VPlace $innerNum$ with different clustering depth and clustering size.

The curves of the average final placement cost as well as average CPU time over ten MCNC benchmark circuits versus clustering size and clustering depth are shown in Figure 4.13 and Figure 4.14 respectively.

We can conclude from Figure 4.13 that two schedules ($L = 2$ with $S = 4$) and($L = 3$ with $S = 2$) produce the best final quality. However, Figure 4.14 shows that the first schedule consumes less CPU time than the latter. Consequently, it is the best selection for our hierarchical approach. Nevertheless, the performance of the placement tool is not extremely sensitive to this specific schedule. As long as we keep the clustering depth L to 2 and S between 2 and 7, the placement quality and CPU time curve appear quite even. Figure 4.13 also shows that the average placement cost deteriorates when top clusters hold 64 CLBs or more.

Furthermore, Figure 4.15 and Figure 4.17 show the performance of the hierarchical placement tool over a medium MCNC circuit with different clustering depth and clustering size. Figure 4.16 and Figure 4.18 show similar results over a large MCNC circuit.

4.4.3 Performance of the New Hierarchical Placemenet Tool

Table 4.7 provides the results, including placement quality and CPU time, obtained by our new hierarchical placement tool with the following schedule: clustering level $L = 2$ and clustering size at each level $S = 4$, with GSA $innerNum = 1$ in the top level, VPlace $innerNum = 5$ in the bottom level, and non-deterministic local search in the medium level.

Table 4.8 shows a comparison between the runtime of the clustering procedure and the overall optimization procedure (i.e. clustering, de-clustering, top-down im-

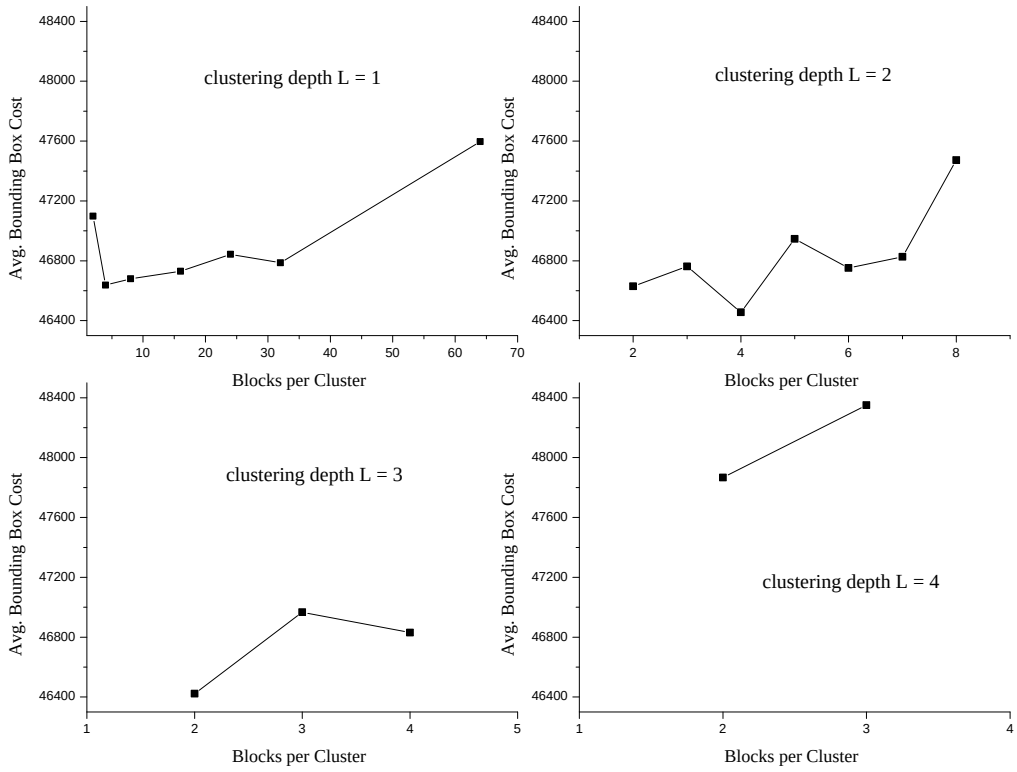

Figure 4.13: Average "Bounding Box Cost" over 10 MCNC circuits vs. clustering size with different clustering depth.

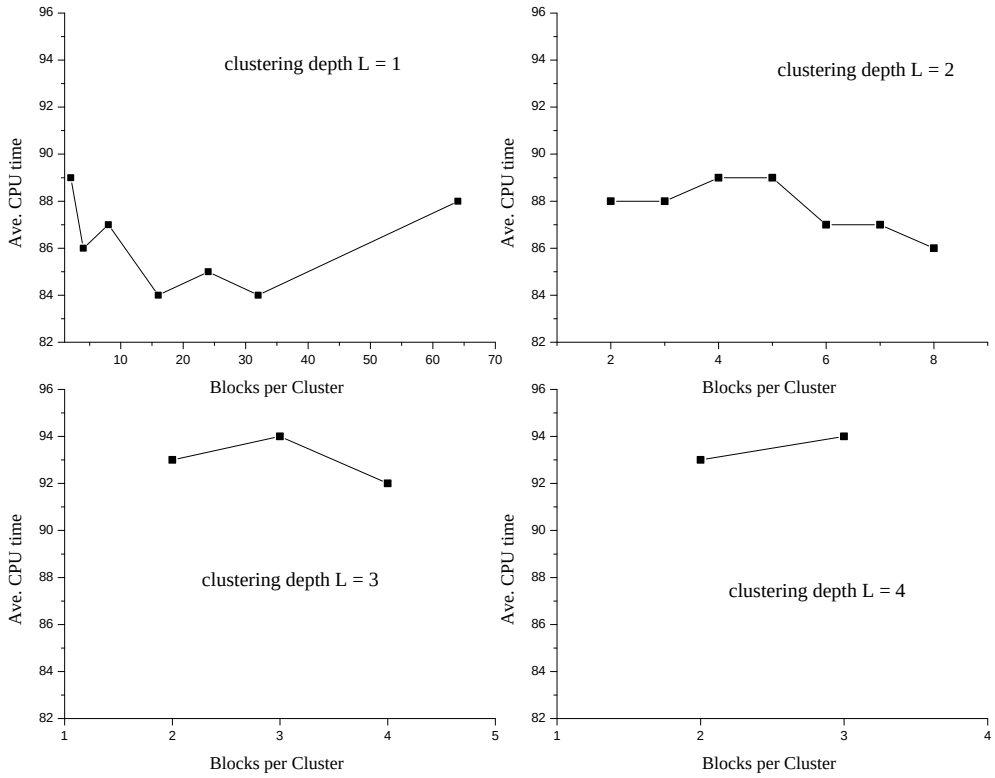

Figure 4.14: Average "CPU time" over 10 MCNC circuits vs. clustering size with different clustering depth.

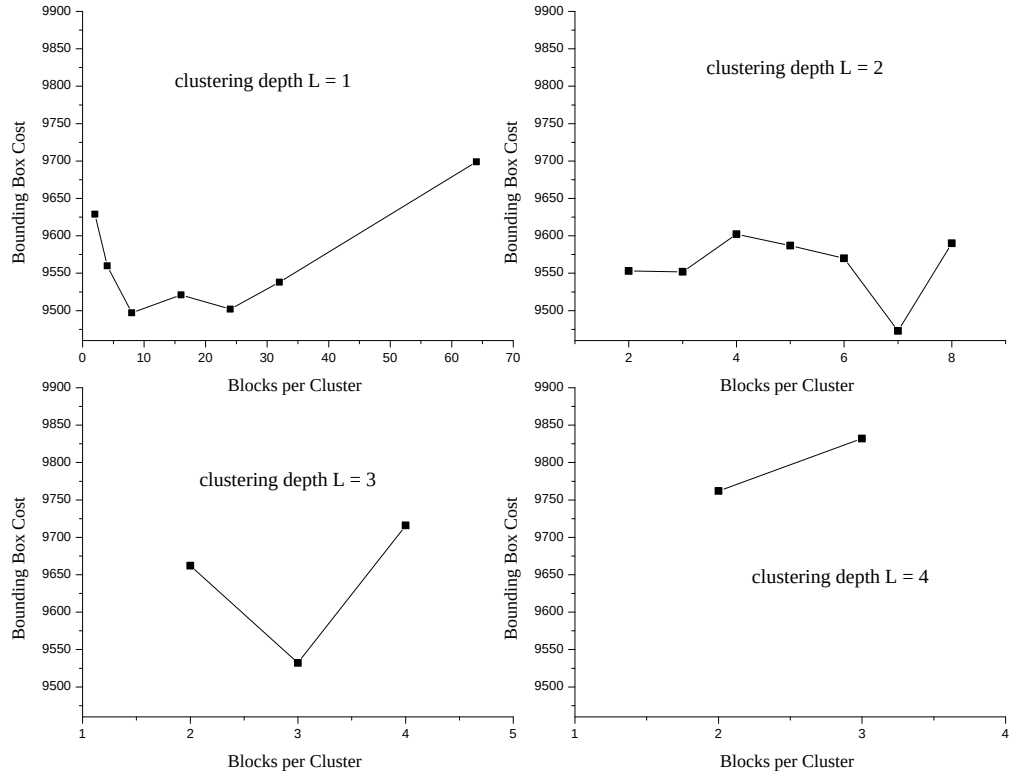

Figure 4.15: "Bounding Box Cost" over a medium MCNC circuit "*tseng*" (1047 CLBs) vs. clustering size with different clustering depth.

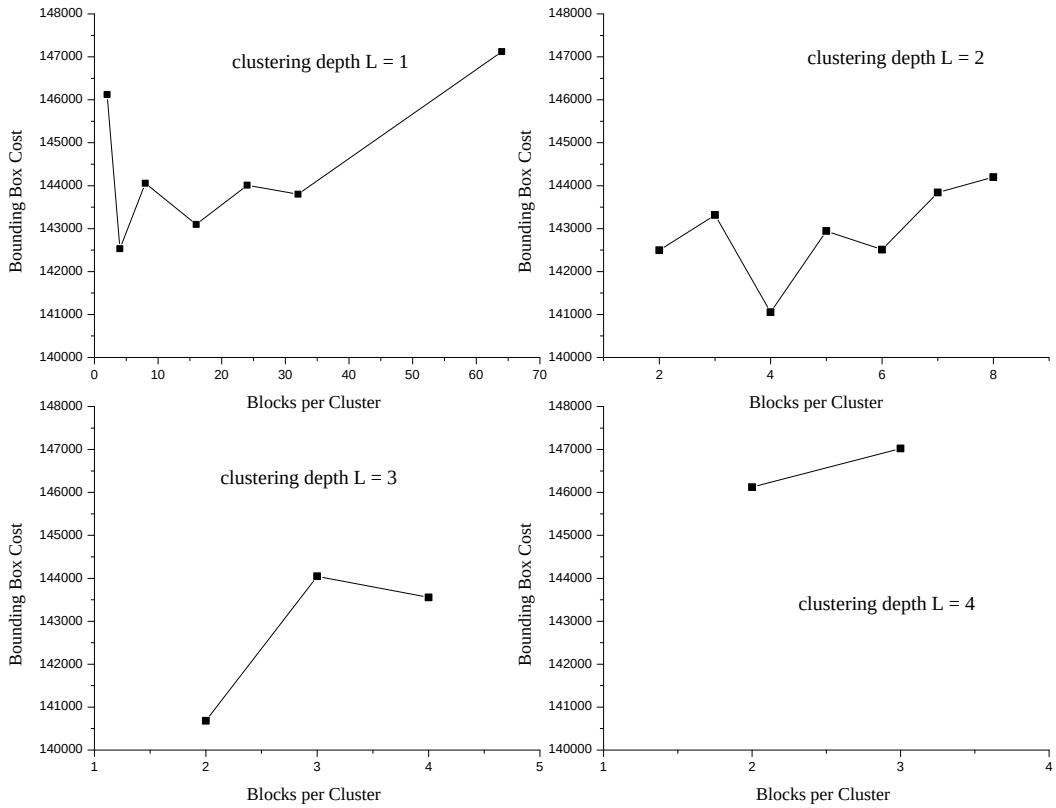

Figure 4.16: "Bounding Box Cost" over a medium MCNC circuit "*clma*" (8383 CLBs) vs. clustering size with different clustering depth.

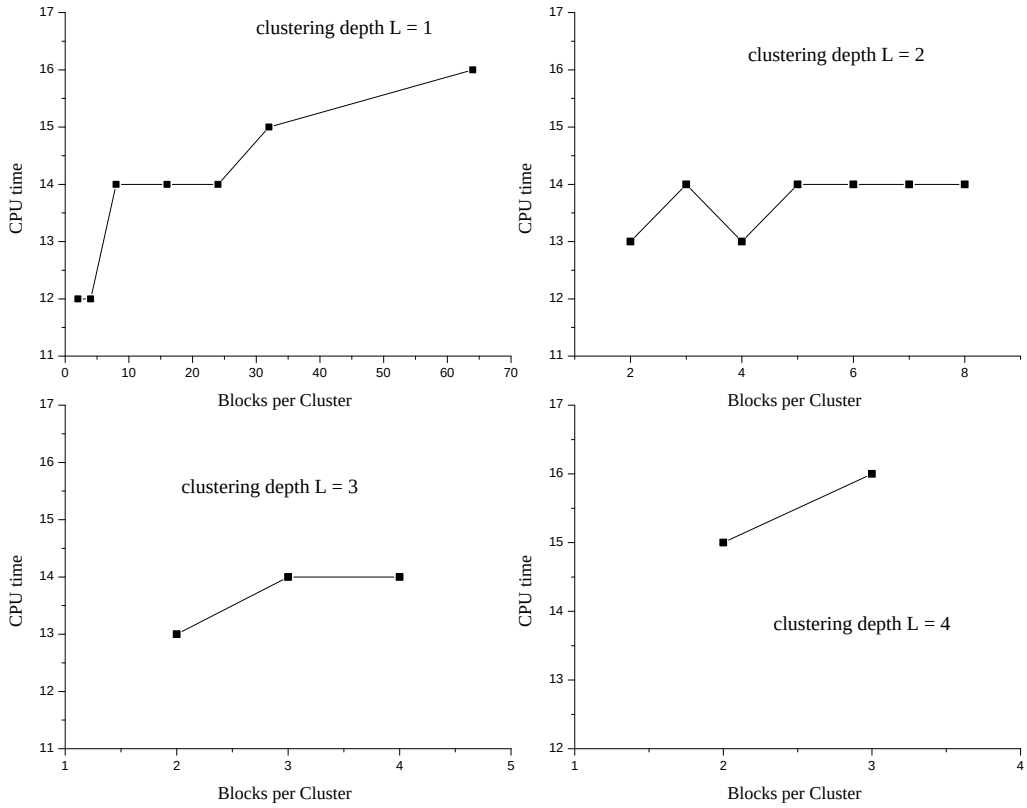

Figure 4.17: "CPU time" over a medium MCNC circuit "*tseng*" (1047 CLBs) vs. clustering size with different clustering depth.

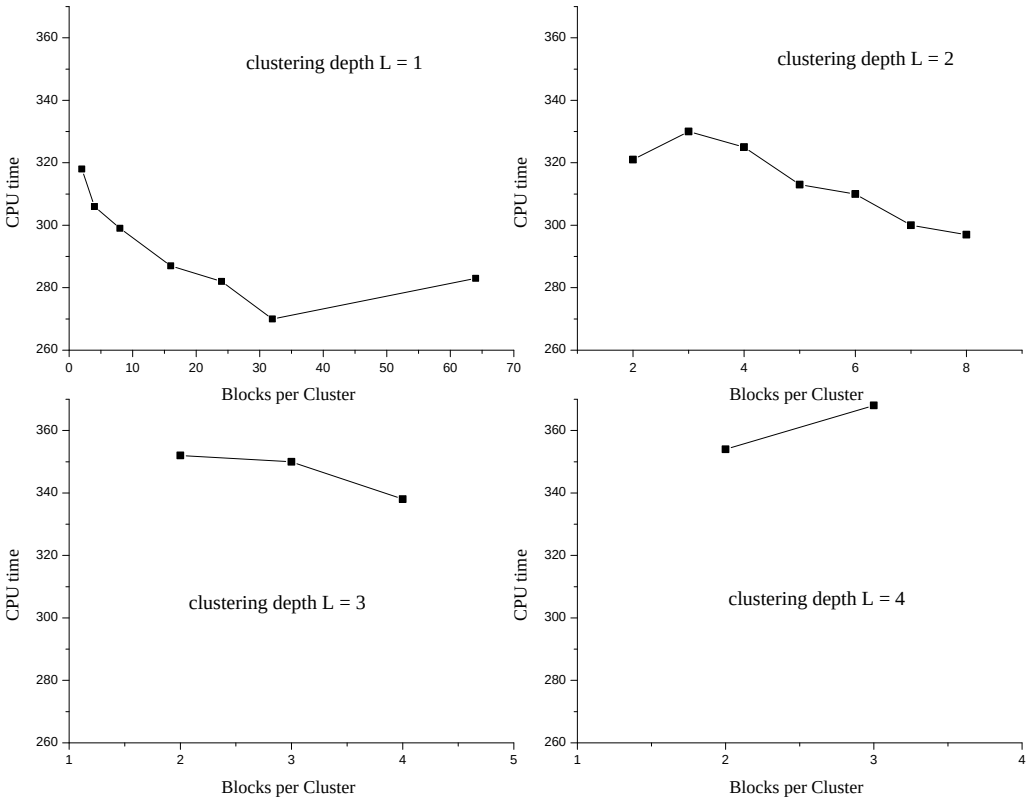

Figure 4.18: "CPU time" over a large MCNC circuit "*clma*" (8383 CLBs) vs. clustering size with different clustering depth.

Circuit name	**Ave. cost**	Max cost	Min cost	Cost STDEV	**Ave. CPU t.**	Max CPU t.	Min CPU t.	CPU t. STDEV
e64	**2877**	2897	2860	6	**3**	3	3	0
tseng	**9662**	9737	9555	77	**13**	14	13	1
ex5p	**16444**	16532	16365	74	**13**	14	12	1
alu4	**19590**	19754	19374	151	**18**	21	16	2
seq	**25008**	25426	24803	210	**27**	28	24	1
M.avg	**17676**	17862	17524	128	**18**	19	16	1
frisc	**53763**	54969	52190	1156	**74**	77	68	4
spla	**62005**	63852	59617	1736	**92**	102	81	8
ex1010	**66471**	67545	66020	620	**138**	152	123	12
s38584.1	**68199**	69430	66843	1218	**187**	194	172	11
clma	**140680**	142612	139339	1436	**326**	353	291	33
L.avg	**78224**	79682	76802	1233	**163**	176	147	14
Avg	**46470**	47275	45697	668	**89**	96	80	7

Table 4.7: Performance of hierarchical placer

provement at each level). The table clearly indicates that the clustering procedure, on average, consumes less than 1% of the overall CPU time.

In addition, Table 4.9 shows the improvement of simple local search and hierarchical placement tool with respect to a random placement. On average, hierarchical placement tool improved the quality of the original random placement by 73%.

4.4.4 Performance Comparison Between the New Hierarchical Placement Tool and GSA Based Placement tool as well as VPlace

Table 4.10 and 4.11 compare the performance [3] of our hierarchical placement tool with that of GSA (default) based placement tool and VPlace respectively.

From Tables 4.10 and 4.11, we can conclude that our hierarchical placement

[3]Comparison results are obtained according to equation: $(Value_{(GSA\ or\ VPlace)} - Value_{(hierarchical)})/Value_{(GSA\ or\ VPlace)} * 100\%$.

Circuit name	Clustering avg.CPU t	Overall avg.CPU t	CPU t %
e64	0.01	3	0.36
tseng	0.08	13	0.64
ex5p	0.09	13	0.70
alu4	0.22	18	1.18
seq	0.13	27	0.52
M.avg	0.13	18	0.74
frisc	0.58	74	0.76
spla	0.35	92	0.39
ex1010	0.67	138	0.49
s38584.1	3.40	187	1.76
clma	2.79	326	0.84
L.avg	1.52	163	0.93
Avg	0.81	89	0.91

Table 4.8: Runtime comparison between the multilevel clustering procedure and the overall hierarchical placement

Circuit name	Random clustering & random de-clustering		Non-deterministic simple local search			Hierarchical placement tool		
	Ave. cost	Ave. CPU t.	Ave. cost	Ave. CPU t.	Cost impro.%	Ave. cost	Ave. CPU t.	Cost impro. %
e64	7452	0	3816	0.23	48.79	2877	3	61.39
tseng	41285	0.02	13938	0.73	66.24	9662	13	76.60
ex5p	42301	0.02	20713	0.71	51.03	16444	13	61.13
alu4	61504	0.08	26134	0.93	57.51	19590	18	68.15
seq	79903	0.03	36418	1.21	54.42	25008	27	68.70
M.avg	56248	0.04	24301	0.90	57.30	17676	18	68.64
frisc	229152	0.19	88342	3.33	61.45	53763	74	76.54
spla	236251	0.11	96534	3.34	59.14	62005	92	73.75
ex1010	332664	0.21	99941	4.59	69.96	66471	138	80.02
s38584.1	559870	0.38	142213	10.19	74.60	68199	187	87.82
clma	796591	0.52	262927	14.15	66.99	140680	326	82.34
L.avg	430905	0.28	137991	7.12	66.43	78224	163	80.09
Avg	238697	0.15	79098	3.94	61.01	46470	89	73.64

Table 4.9: Solution improvement of simple local search and hierarchical placement tool ($L = 2, S = 4$)

Circuit name	**Ave.cost impro.%**	Max.cost impro.%	Min.cost impro.%	**Ave.CPU t impro.%**	Max CPU t impro.%	Min CPU t impro.%
e64	**-0.17**	-0.59	+0.14	**0**	0	0
tseng	**-2.31**	-1.49	-3.15	**+30.53**	+28.50	+32.63
ex5p	**-1.01**	-0.73	-0.94	**+28.89**	+27.37	+30.01
alu4	**-0.80**	-0.60	-0.79	**+39.33**	+32.90	+46.33
seq	**-0.57**	-1.86	-0.22	**+30.77**	+33.81	+34.32
M.avg	**-1.17**	-1.16	-1.27	**+32.38**	+30.65	+35.82
frisc	**-2.66**	-4.31	-0.17	**+40.80**	+42.96	+41.38
spla	**-2.11**	-3.64	+0.04	**+31.85**	+27.66	+37.69
ex1010	**-0.87**	-2.08	-0.59	**+30.10**	+28.97	+34.57
s38584.1	**-3.84**	-5.34	-2.21	**+42.46**	+41.47	+45.22
clma	**+0.14**	-0.48	+0.49	**+43.21**	+41.36	+44.78
L.avg	**-1.87**	-3.17	-0.49	**+37.68**	+36.54	+40.73
Avg	**-1.42**	-2.11	-0.74	**+30.59**	+30.19	+36.03

Table 4.10: Comparison between GSA placer and hierarchical placer

Circuit name	**Ave.cost impro.%**	Max.cost impro.%	Min.cost impro.%	**Ave.CPU t impro.%**	Max CPU t impro.%	Min CPU t impro.%
e64	**-0.66**	-0.52	-0.70	**+78.46**	+76.15	+80.00
tseng	**-2.85**	-3.16	-2.03	**+81.30**	+80.41	+81.45
ex5p	**-1.34**	-1.24	-1.45	**+81.66**	+80.56	+82.75
alu4	**-2.24**	-2.71	-1.62	**+82.53**	+80.59	+84.22
seq	**-1.10**	-2.70	-0.42	**+80.96**	+80.69	+82.29
M.avg	**-1.88**	-2.45	-1.38	**+81.61**	+80.56	+82.67
frisc	**-3.08**	-3.84	-1.23	**+81.14**	+80.85	+82.25
spla	**-1.57**	-3.78	+1.50	**+77.80**	+75.71	+79.95
ex1010	**-1.49**	-2.84	-1.01	**+75.14**	+73.19	+77.35
s38584.1	**-5.04**	-6.04	-3.84	**+79.34**	+78.98	+64.60
clma	**-0.21**	-0.52	-0.04	**+75.53**	+73.86	+77.56
L.avg	**-2.28**	-3.40	-0.92	**+77.79**	+76.51	+79.54
Avg	**-1.96**	-2.73	-1.08	**+79.36**	+78.09	+80.84

Table 4.11: Comparison between VPlace and hierarchical placer

tool, on average, achieves 30% reduction in CPU time (1.4x faster) compared to GSA (default) placement tool at the cost of a slight (less than 1.5%) increase of the final placement cost. When compared with VPlace, on average, it achieves 79% reduction in CPU time (4.8x faster) at the cost of a slight (less than 2%) increase of the final placement cost.

Compared with the most recent FPGA chips that include over 10 million effective gates, our test circuits are quite small, even through they are the largest and the most frequently cited benchmarks in the literature. For these undersized circuits, flat iterative algorithms, like GSA and VPlace simulated-annealing, perform well within reasonable amount of CPU time. Furthermore, based on the experiments conducted in Section 3.4.2, the solutions obtained by VPlace are close to optimal. This prevents our hierarchical placement tool producing striking results, similar to Timberwolf95, which yields total wirelength reductions of up to 9% while consuming up to 7.5 times less CPU time in comparison to its flat counterpart.

4.5 Summary

In this chapter, a simple yet effective clustering method similar to the approach used in [Sank99] was utilized and embedded into our implementation. The overall approach strategy and their key tunable parameters of both the clustering formation (clustering depth, clustering size) and placement algorithm ($innerNum$) used at each level were presented. A novel method, which automatically calculates the start parameters (start temperature T_0, initial R_{limit}) for VPlace simulated-annealing algorithm in the bottom level, is provided. The search for the set of parameters that provide the best quality and CPU time trade-off for our hierarchical placement

tool was described as well. In addition, some hierarchical placement related issues are evaluated such as the performance of a clustering, de-clustering and simple local search.

Placement results obtained by VPlace (described in section 3.4.2) and GSA (described in section 3.7.3) were used as the metric of comparison for our placement tool for both quality and runtime. The comparison results show that the new hierarchical placement tool yields promising final quality, while consuming, on average, 1.4x less time measured with GSA placement tool and 4.7x less CPU time measured with VPlace.

Chapter 5

Conclusions and Future Directions

5.1 Conclusion

In this thesis, two new approaches, which deal with the FPGA placement problem, have been presented and investigated. A novel greedy iterative heuristic and a new hierarchical heuristic were shown to provide excellent results. Both methods greatly reduce the execution time of FPGA placement, while achieving the same quality of placement, when compared to the placement quality record holder, VPlace.

In the first part of the thesis, we restricted our focus to an enhancement of simulated-annealing algorithm (GSA), which attempts to solve the problem in a top-down manner by considering all (flat) blocks simultaneously. By utilizing part of the short term search history, GSA appears more greedy than simulated-annealing used in VPlace, thus it effectively accelerates the convergence of a search. On average, GSA with default parameter is 3.2x faster compared with VPlace at the cost of only slightly (0.5%) deterioration of the final placement quality over ten MCNC benchmark circuits. When we increase the runtime for GSA to obtain a

better solution, solution quality improves slightly (0.4%) with 1.7x speedup compared with VPlace over the same suite of benchmarks. Furthermore, adaptively altering the maximum distance between two swapping blocks has shown speeding up the convergence of our heuristic significantly.

The size of FPGA designs is increasing at a substantial rate, and new effective methods are needed. One approach, such as GSA, is utilized to speed up the convergence of traditional time-consuming heuristics, e.g. simulated-annealing. Another approach is to reduce the complexity of the problem itself, by restructuring the original placement into a simplified form through multi-level *clustering*.

In our hierarchical approach, a simple yet effective clustering method in [Sank99], which generates high-quality clusters in negiligible (less than 1%) CPU time compared with the overall placement, is selected to complete the multi-level clustering. The objective of this approach is to reduce the number of entities that need to be considered and the number of interconnections between them, at each level of the hierarchy. The smaller problem, consequently, reduces the degrees of freedom for block moves, making advanced time-consuming methods feasible for today's largest FPGAs.

There are two type of schemes necessary to be investigated in the hierarchical approach: those that control the clusters such as clustering depth and clustering size and those that determine which available improvement method should be implemented at each specific level. We explored various combinations of these two schemes to find the best quality and time trade-off envelope. Results obtained show that our hierarchical placement tool yields, on average, less than 2% worse final placement quality and is 4.7x faster than VPlace over our test benchmark circuit suite. In addition, a novel adaptive technique, which automatically determines

the suitable initial parameters of VPlace simulated-annealing, is provided to effect a smooth transition between different algorithms at different hierarchical levels.

5.2 Future Work

Our work has provided some research in the domain of FPGA fast placement. However, there are many areas within this topic necessary to be explored more thoroughly.

1. It would be interesting and beneficial to evaluate the performance of *Genetic Algorithm* (GA) for FPGA placement, especially in the hierarchical approach. GA has been proven to be effective in solving many search and optimization problems including numerical function optimization, machine learning, combinatorial optimization tasks and so on. Iterative heuristics, like the simulated-annealing algorithm, only look at one candidate solutions at one time, so they do not build up an overall picture of the search space. In contrast, GA explores a problem in a parallel manner with a large set of *population*. In many *generations*, it would give an exhaustive view of the whole search space.

2. Another interesting field worth investigation to speed up FPGA placement is the parallel approach. Taking advantage of multi-processor computers, a few small jobs that are divided from a large task can be executed at the same time.

3. A constructive technique similar to GRASP or based on analytical mathematical technique can be used to create good initial starting point for the local

search at the highest level of the hierarchy, thus speeding the convergence of the algorithm.

Appendix A

MCNC Benchmarks

Table A.1 shows the MCNC [Yang91] benchmark test suite used to measure the performance of the heuristics in this thesis. This test suite consists of ten circuits ranging from a few hundred CLBs to nearly ten thousand CLBs which are listed in three groups according to the size. The small circuit is "$e64$", the medium circuits are "$tseng$", "$ex5p$", "$alu4$" and "seq", and the large circuits are "$frisc$", "$spla$", "$ex1010$", "$s38584.1$" and "$clma$".

The circuits included in the large and medium groups are all from the largest 20 MCNC logic benchmark circuits indicated in the work of Betz et al. [Betz97a]. They are the largest benchmark circuits we can obtain from the literature, even though they are a little bit dated from today's deep-submicron view. These circuits are the primary focus of this thesis and have been widely used in many FPGA physical design publications [Mulp01] [Betz97a] [Betz99] [Betz97b] [Sank99] [Part01]. While the small circuit used here is for completeness, it is not as valuable as the others for drawing conclusions.

Circuit name	FPGA matrix	Number of CLBs	Number of I/O Pads	Number of Nets	Average Fanout	Maximum Fanout
e64	17x17	274	130	290	3.94	22
tseng	33x33	1047	174	1099	4.28	389
ex5p	33x33	1064	71	1072	4.73	324
alu4	40x40	1522	22	1536	4.52	250
seq	42x42	1750	76	1791	4.46	234
frisc	60x60	3556	136	3576	4.48	774
spla	61x61	3690	62	3706	4.73	215
ex1010	68x68	4598	20	4608	4.49	303
s38584.1	81x81	6447	342	6485	4.18	2742
clma	92x92	8383	144	8445	4.61	1170

Table A.1: MCNC Benchmark circuit suite used as test cases

Appendix B

FPGA Placement Problem Illustration

Figure B.1 and Figure B.2 are output of VPR package [Betz00] which can give us a vivid demonstration of the FPGA placement problem. The benchmark circuit "*e*64" and target FPGA architecture (9x9 matrix) are all the same in these two figures, while the physical location of CLBs and I/O pads are changed. Figure B.1 shows the initial configuration whose bounding box cost equals to 3262.69 and figure B.2 shows the final configuration after simulated-annealing improvement which has a much reduced bounding box cost 1905.76. The total wirelength is also reduced that can be easily discovered from the reduction of the density of wire tracks. Furthermore, the I/O pad pitch-to-logic block ratio, which is how many pads available at each marginal block location, is set to 4.

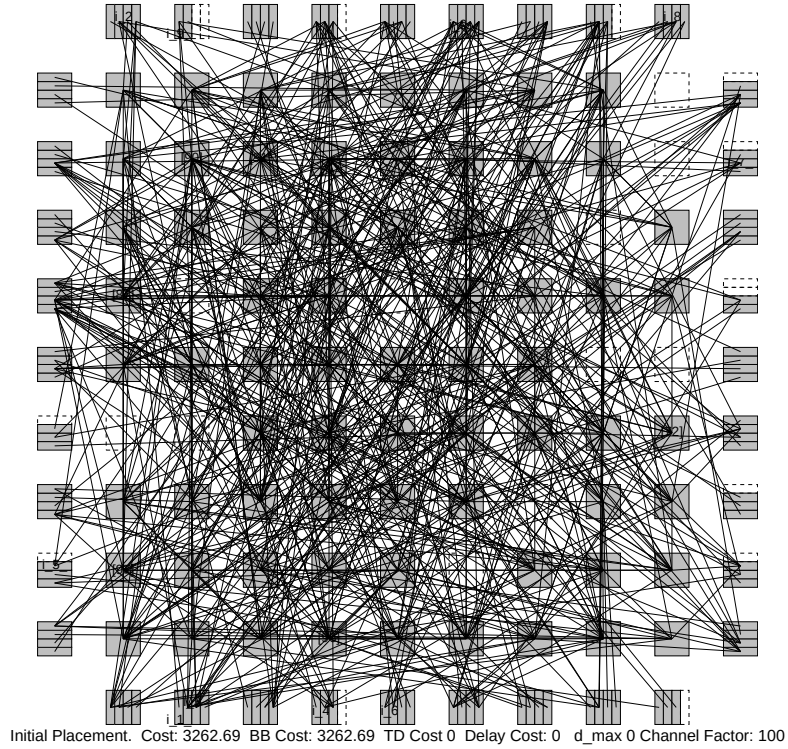

Figure B.1: Initial (random start) placement configuration of MCNC circuit "*e*64" implemeting on a 9x9 FPGA. (courtesy of Jonathan Rose)

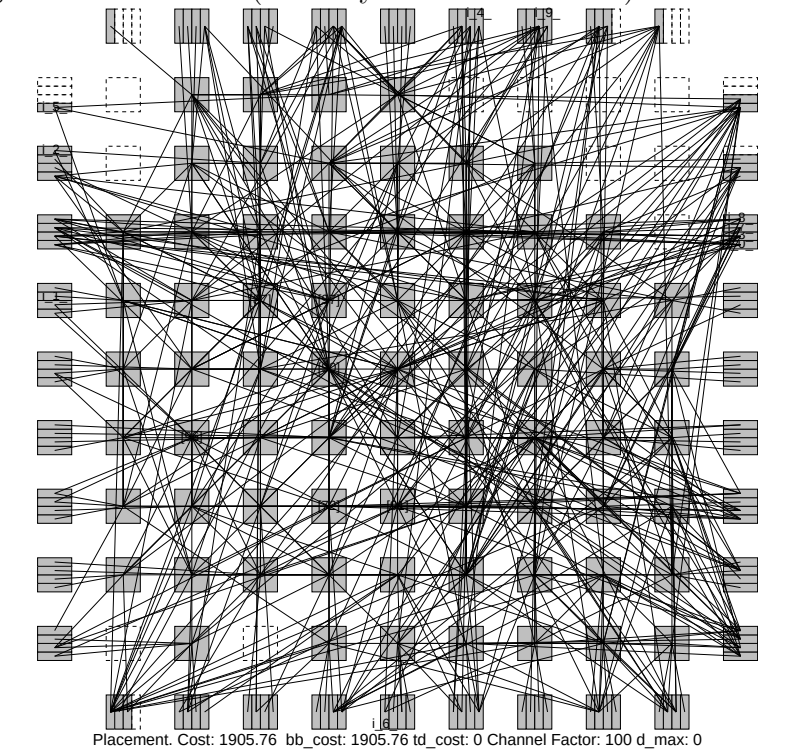

Figure B.2: Final placement configuration of MCNC circuit "*e*64" implemeting on a 9x9 FPGA. (courtesy of Jonathan Rose)

Appendix C

Acronym Glossary

ASIC:	Application-Specific Integrated Circuit.
CAD:	Computer-Aided Desigh.
CLB:	Configurable Logic Block.
CPLD:	Complex PLD.
FPD:	Field-Programmable Device.
FPGA:	Field-Programmable Gate Array.
GSA:	Greedy Simulated-Annealing algorithm.
HDL:	Hardware Description Language.
MCNC:	Microelectronics Corporation of North Carolina.
MPGA:	Mask-Programmable Gate Array.
NRE:	non-recurring engineering
PLD:	Programmable Logic Device.
SPLD:	Simple PLD.
VLSI:	Very Large Scale Integration.
VPR:	Versatile Placement and Routing tool for FPGAs.

Bibliography

[Aart03] E. Aarts and J. K. Lenstra, "Local Search in Combinatorial Optimization," Princeton Universtiy Press, 2003.

[Alpe97a] C. J. Alpert, T. Chan, D. Huang, I. Markov, and K. Yan, "Quadratic Placement Revisited," *ACM/IEEE Design Automation Conference*, pp. 752–757, 1997.

[Alpe97b] C. J. Alpert, J. H. Huang, and A. B. Kahng, "Multilevel Circuit Partitioning," *Proc. ACM/IEEE Design Automation coference*, pp. 530–533, 1997.

[Arei01a] S. Areibi, M. Thompson, and A. Vannelli, "A Clustering Utility Based Approach for ASIC Design," *14th Annual IEEE International ASIC/SOC Conference, Washington, DC*, pp. 12–15, September 2001.

[Arei01b] S. Areibi, M. Xie, and A. Vannelli, "An Efficient Rectilinear Steiner Tree Algorithm for VLSI Global Routing," *Canadian Conference on Electrical and Computer Engineering*, May 2001.

[Betz00] V. Betz and J. Rose, "VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs Package ver 4.30," Available from http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html, 2000.

[Betz97a] V. Betz and J. Rose, "The FPGA Place-and-Routing Challenge," Available from http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html, 1997.

[Betz97b] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *Proc. Intel. Workshop on Field Programmable Logic and Applications*, pp. 213–222, 1997.

[Betz99] V. Betz, J. Rose, and A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAs," Kluwer Academic Publishers, 1999.

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

[Brow96] S. Brown, M. Khellah, and G. Lemieux, "Segmented Routing for Speed-Performance and Routability in Field-Programmable Gate Arrays," *Journal of VLSI Design*, vol. 4, No. 4, pp. 275–291, 1996.

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

[Chen94] C. Cheng, "A accurate and Efficient Placement Routability Modeling," *ICCAD*, pp. 690–695, 1994.

[Coho86] J. P. Cohoon and W. D. Paris, "Genetic Placement," *Proc. IEEE Intl. Conference on Computer-Aided Design*, pp. 422–425, 1986.

[Cong94] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Based FPGA Designs," *IEEE transactions on Computer-Aided Design*, pp. 1–13, January 1994.

[Fidu84] C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," *Design Automation Conference*, pp. 175–181, 1984.

[Ganl95] J. L. Ganley, "Geometric Interconnection and Placement Algorithms," *Ph.D Thesis, University of Virginia, School of Engineering and Applied Science*, 1995.

[Gare79] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," San Francisco, CA: Freeman, 1979.

[Ghen02] H. Ghenniwa and S. Areibi, "Agent-Orientation for Evolutionary Computation," *Proc. of the Genetic and Evolutionary Computation Conference*, pp. 57–65, July 2002.

[Hage92] L. Hagen and A. B. Kahng, "A New Approach to Effective Circuit Clustering," *IEEE International Conference on CAD*, pp. 422–427, 1992.

[Huan86] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated annealing," *ICCAD*, pp. 381–384, 1986.

[Karg86] P. G. Karger and B. T. Preas, "Automatic Placement: A Review of Current Techniques," *IEEE Proc. of The 23rd DAC, Las Vegas, Nevada*, pp. 622–629, 1986.

[Kary97] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partition: Applications in VLSI Domain," *Proc. ACM/IEEE Design Automation coference*, pp. 526–529, 1997.

[Kern70] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, pp. 49(2):291–307, 1970.

[Kirk83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671–680, 1983.

[Klei91] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, "GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization," *IEEE transactions on Computer-Aided Design*, vol. 10, No. 3, pp. 356–365, March 1991.

[Krus56] J.B. Kruskal, "On the shortest spanning tree of a graph and the traveling salesman problem," *Proc. of The American Mathematical society 7*, pp. 48–50, 1956.

[Lam88] J. Lam, J. Delosme, and C. Sechen, "Performance of a New Annealing Schedule," *Proc. 25th DAC coference*, pp. 306–311, 1988.

[Mall89] S. Mallela and L. K. Grover, "Clustering based Simulated Annealing for Standard Cell Placement," *Proc. Design Automation Conference*, pp. 312–317, 1989.

[Marq00] A. Marquardt, V. Betz, and J. Rose, "Timing-Driven Placement for FPGAs," *FPGA 2000, ACM Symposium on FPGAs*, pp. 203–213, February 2000.

[McMu88] L. McMurchie and C. Ebeling, "PathFinder: A Negotiation Based Performance Driven Router for FPGAs," *Proc. of the International Symposium on FPGAs*, pp. 111–117, February 1988.

[Mitr86] D. Mitra, R. Romeo, and A. Sangiovanni-Vincentelli, "Convergence and Finite-Time Behavior of Simulated Annealing," *Advances in Applied Probability*, vol. 18, No. 3, pp. 747–771, 1986.

[Mulp01] C. Mulpuri and S. Hauck, "Runtime and Quality Tradeoffs in FPGA Placement and Routing," *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 29–36, 2001.

[Part01] G. Parthasarathy, M. Marek-Sadowska, A. Mukherjee, and A. Singh, "Interconnect Complexity-aware FPGA Placement Using Rent's Rule," *Proc. of System Level Interconnect Prediction*, March 2001.

[Prim57] R. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, pp. 36:1389–1401, 1957.

[Rose90] J. Rose, W. Klebsch, and J. Wolf, "Temperature Measurement and Equilibrium Dynamics of Simulated Annealing Placements," *IEEE transactions on Computer-Aided Design*, vol. 9, No. 3, pp. 253–259, March 1990.

[Sait95] S. M. Sait and H. Youssef, "VLSI Physical Design Automation," IEEE Press, New Jersey, 1995.

[Sank99] Y. Sankar and J. Rose, "Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs," *Proc. of The 7th ACM/SIGDA Intl. Symposium on FPGAs*, pp. 157–166, 1999.

[Schu72] D. M. Schuler and E. Ulrich, "Clustering and Linear Placement," *Proc. Design Automation Workshop*, pp. 50–56, 1972.

[Sech85] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE Journal of Solid-State Circuits*, vol. 20, No. 2, pp. 510–522, April 1985.

[Shah91] K. Shahookar and P. Mazumder, "VLSI Cell Placement Techniques," *ACM Computing Surveys*, vol. 23, No. 2, pp. 143–220, June 1991.

[Shin93] H. Shin and C. Kim, "A Simple Yet Effective Technique for Partitioning," *IEEE transactions on VLSI Systems*, vol. 1, No. 3, pp. 380–386, September 1993.

[Song92] L. Y. Song and A. Vannelli, "A VLSI Placement Method Using Tabu Search," *Microelectronics Journal*, vol. 11, No. 7, pp. 167–172, July 1992.

[Sun95] W. Sun and C. Sechen, "Efficient and Effective Placement for Very Large circuits," *IEEE transactions on Computer-Aided Design Automation Conference*, vol. 14, No. 3, pp. 349–359, March 1995.

[Swar90] W. Swartz and C. Sechen, "New Algorithms for the Placement and Routing of Macro Cells," *ICCAD*, pp. 336–339, 1990.

[Swar95] W. Swartz and C. Sechen, "Timing-Driven Placement for Large Standard Cell Circuits," *DAC*, pp. 211–215, February 1995.

[Thom01] M. Thompson, "A Clustering Utility-Based Approach for ASIC Design," *Master Thesis, Dept. of Electrical Engineering, University of Waterloo, Waterloo, Ontario, Canada*, 2001.

[Wong88] D. F. Wong, H. W. Leong, and C. L. Liu, "Simulated Annealing for VLSI Design," Kluwer Academic Publishers, 1988.

[Yang91] S. Yang, "Logic Synthesis and Optimization Benchmarks," *Tech. Report, Microelectronics Center of North Carolina*, 1991.