ANALYTIC METHODS FOR

THE FPGA PLACEMENT PROBLEM

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

MING XU

In partial fulfilment of requirements

for the degree of

Doctor of Philosophy

May, 2009

Abstract

Within the last 20 years, the use of *Field Programmable Gate Arrays* (*FPGAs*) to implement digital systems has grown significantly because of their flexibility, dramatic reduction in turn-around time, and start-up costs compared with traditional *Application Specific Integrated Circuits* (*ASICs*). Moreover, FPGAs themselves have experienced an exponential growth in size, complexity, and performance. *Computer-Aided Design* (*CAD*) plays a critical role in optimizing high-performance design solutions using these high-end FPGAs. However, compilation times for designs, which are dominated by placement and routing times, are growing much more rapidly than the available computation power. While current CAD algorithms provide quality solutions, they often require significant amounts of CPU time. For many circuits, the compile time can be on the order of tens of CPU hours, which adversely impacts the use of FPGAs by hardware designs. This provides compelling motivation to explore new methods for fast compilation of designs.

In this thesis, we focus on the placement phase of the FPGA design process. Given a circuit represented as a connection of logic blocks, the placement problem can be stated as that of assigning each logic block to a unique physical resource on the FPGA while achieving a given overall performance. Placement is an NP-complete problem [10] and one of the most time-consuming tasks in the automation of FPGA design.

We present a new "near-linear" model for estimating wirelength, called Star+. The model is similar to the traditional star model [21], but is strictly differentiable, making it suitable for use with analytic placement methods. Most importantly, the time to compute the change in cost resulting from the swap of two blocks always runs in $O(1)$ time. We also present two analytic placement methods based on *Conjugate Gradient* (*CG*) [18] and *Successive Over-Relaxation* (*SOR*) [33], respectively. Both analytic methods seek to minimize total wirelength using the Star+ model as an estimate of wirelength. The novelty of the CG method lies in the fact that this method avoids computing the Hessian matrix on each iteration, thus reducing the (traditional) cost of computing the inner loop

from $O(n^2)$ to $O(n)$. The SOR method, on the other hand, has the same runtime complexity as CG, but by properly arranging the sequence in which equations in the non-linear equation system are processed, SOR placement runs faster by a constant amount (approximately 7x). We also present a novel pre-placement method for pre-assigning certain blocks on the FPGA which runs in $O(n \log n)$ time. Finally, we develop and present a timing-driven placement algorithm by adding timing-driven parameters into the original (Star+) objective function used by both CG and SOR.

Compared with *Versatile Place and Route* (*VPR*) – the state-of-the-art academic placement tool, with our methods we are able to achieve solutions 4 to 40 times faster and with 1 to 8.8% less critical-path delay.

Acknowledgements

First, I would like to acknowledge my sincere gratitude to my advisor, Dr. Gary Grewal, for his continuous and wholehearted support in the Ph.D. program. He was always willing to listen and gave many valuable suggestions. He not only taught me skills and approaches to solve problems, but also helped and encouraged me when I was in difficulty. His broad knowledge and rich experience were essential to the accomplishment of my thesis. To me, he is a responsible supervisor, as well as a reliable mentor.

I would also like to thank Dr. Dilip Banerji, for his generous help and expert advice. I am indebted to him for his high requirements and thoughtful recommendations.

I would like to thank Dr. Kenneth Kent from University of New Brunswick, for his willingness to read my thesis and precious feedback.

I really appreciate the help of Dr. Tom Wilson for his enlightening suggestion and feedback on my thesis.

Many thanks to Dr. Charlie Obimbo, Dr. Shawki Areibi, Dr. Judi McCuaig, Dr. William Gardener, Sheryl Beauchamp, Debra Byart, Pam Varga, and others for all their help.

Last, but not least, I thank my parents, Junyu Xu and Wenruo Li. I thank my wife, Xiaoyan, and my son, Jason, for their persistent support and understanding.

Contents

1 Introduction 1
1.1 Motivation and Technology Trends 1
1.2 Overview of the Placement Problem 3
1.3 FPGA Placement Methods 7
1.4 Contributions 12
1.5 Thesis Organization 13
2 Relevant Background 15
2.1 FPGA Architecture(s) 15
2.2 FPGA Design Procedure 17
2.3 Placement Methods 22
2.3.1 Partition-Based Methods 22
2.3.2 Simulated Annealing 23
2.3.3 Analytic Methods 28
2.3.4 Multilevel Clustering 31
2.3.5 Other Approaches to Placement 33
2.4 Wire-Length Models 34
2.5 Summary 42
2.6 Benchmarks 43
3 The Star+ Model 45
3.1 Wire-estimation based on the Star+ model 46
3.2 Constant-time update of cost 48
3.3 Star+ Model Evaluation 50
3.3.1 Routability 51
3.3.2 Critical Path Delay 55
3.3.3 CPU Running Time 62
3.3.4 Wirelength 65
3.4 Parameter Tuning 67

3.5 Limitations of the Star+ Model 72
3.6 Summary 73
4 Modifying Conjugate Gradient for Placement 75
4.1 Conjugate Gradient Method 76
4.1.1 Standard Conjugate Gradient Algorithm 77
4.2 Conjugate Gradient Placement 82
4.2.1 Objective Function $f(x)$ 82
4.2.2 Gradient $f'(x)$ 84
4.2.3 Hessian Matrix $f''(x)$ 86
4.3 Conclusion 91
5 Pre-Placement and Legalization Methods 93
5.1 I/O Pad Pre-placement 94
5.1.1 Terminology 96
5.1.2 Shrubbery Example 98
5.1.3 Shrubbery Algorithm 103
5.1.4 Implementation and Time Complexity 106
5.2 Legalizing Solutions using Recursive Bi-partitioning 106
5.3 The CG Placement Algorithm 107
5.4 Convergence of CG 109
5.5 Experimental Results 111
5.5.1 Shrubbery versus Random Pre-Placement 111
5.5.2 CG versus VPR 113
5.6 Summary 117
6 Successive Over-Relaxation Placement 119
6.1 Background 120
6.1.1 Jacobi Method 120
6.1.2 Gauss-Seidel Method 121
6.1.3 Successive Over-Relaxation 124
6.2 SOR Placement 125
6.3 Improving SOR for FPGA Placement 128
6.3.1 Ordering Heuristic 129

6.3.2 Effect of Ordering Heuristic 135
6.3.3 Choosing the Value of Relaxation Factor ω 136
6.4 Experimental Results 138
6.4.1 SOR versus CG 138
6.4.2 SOR versus VPR 140
6.5 Convergence of SOR 144
6.6 Hybrid Approach 145
6.7 Conclusion 149
7 Conclusions and Future Work 151
7.1 Contributions 151
7.2 Future Work 154
7.2.1 Multilevel Optimization 154
7.2.2 FPGA Routing 155
7.2.3 Algorithm Acceleration via Multi-Core and/or Re-configurable Computing 156
7.2.4 Timing and Congestion 156
7.2.5 Modern FPGA Architectures 157
A Timing-Driven Placement 158
A.1 Background 158
A.1.1 Timing Analysis 159
A.1.2 Criticality and Cost 161
A.2 SOR Timing-driven Placement 162
Bibliography 171

List of Tables

2.1: Weight of net with cardinality less than or equal to 50. 37
2.2: 20 MCNC benchmarks. 44
3.1: Channel Width and Routing (`breadth_first` and `inner_num`=1). 52
3.2: Channel Width and Routing (`breadth_first` and `inner_num`=10). 53
3.3: Channel Width and Routing (`timing_driven` and `inner_num`=1) 54
3.4: Channel Width and Routing (`timing_driven` and `inner_num`=10) 55
3.5: Summary of Minimum Routable Channel Widths 56
3.6: Critical Path Delay 57
3.7: Results of Student T-test 59
3.8: CPU Running Time 64
3.9: Re-computing time for HPWL and Star+ 65
3.10: The number of Wire Segments Needed for Successful Routing 66
3.11: Routing Results for Different Values of β (between 0.5 and 0.7) 68
3.12: Routing Results for Different Values of β (between 0.8 and 1.1) 69
3.13: Routing Results for Different Values of β (between 1.2 and 1.5) 70
3.14: Routing Results for Different Values of β (Summary) 70
3.15: Experimental Results of Different α Values 71
5.1: Shrubbery pre-placement vs. random pre-placement 113
5.2: Running time of CG and VPR in Seconds 114
5.3: Critical-path delays (CG vs. VPR) 115
5.4: Wirelength (CG vs. VPR) 116
6.1: With ordering vs. without ordering 136
6.2: Comparisons between SOR and CG 139
6.3: Running time of SOR and VPR in Seconds 141
6.4: Critical path delays (SOR vs. VPR) 142
6.5: Wirelength (SOR vs. VPR) 143
6.6: Critical path delays (hybrid) 147

6.7: Wirelength (hybrid) 148
7.1: Summary of contributions 152

List of Figures

1.1: A “good” placement. 6
1.2: A “bad” placement. 6
2.1: Architecture of Island Style FPGA. 16
2.2: Typical FPGA design flow. 19
2.3: Pseudo-code for simulated annealing. 26
2.4: Multi-level clustering. 32
2.5: Steiner tree example with a 3-block net. 36
2.6: HPWL model for 3-block net. 37
2.7: Two nets with the same bounding-box size. 38
2.8: The clique model of the net in Figure 2.6 (*x*-dimension). 40
2.9: The star model of the net in Figure 2.6 (*x*-dimension). 41
3.1: A star model of a 4-pin net 48
3.2: The placement of Net clma:661 obtained using bounding box 61
3.3: The placement of Net clma:661 obtained using Star+ 61
3.4: Pseudo-code of incremental bounding box evaluation 62
3.5: Pseudo-code for re-computing the Star+ model of net *l* 63
3.6: A hard logic within the Star+ model of a net 72
4.1: The graph of a positive-definite function $f(x)$ 78
4.2: The contours of $f(x)$ 79
4.3: The gradient $f'(x)$ of $f(x)$ 80
4.4: The Conjugate Gradient method 81
4.5: Pseudo-code of CG placement algorithm 92
5.1: The pre-placement of I/O blocks 95
5.2: Illustration of shrub, hedge, and grove 97
5.3: An arbitrary circuit 99
5.4: The corresponding graph 100
5.5: The shrubs when distance is 1 101

5.6: The shrubs when distance is 2 101
5.7: The shrubs when distance is 3 102
5.8: The final tree 103
5.9: Shrubbery algorithm 104
5.10: Placement of I/O pads shown in Fig 5.8 105
5.11: Pseudo-code of bi-partitioning algorithm 108
5.12: Entire CG placement algorithm 109
5.13: Wirelength with different reduction rates of the iteration number 110
5.14: CPU running time with different reduce rates of the iteration number 111
6.1: Jacobi method 122
6.2: Gauss-Seidel method 123
6.3: Successive Over-Relaxation method 125
6.4: The corresponding graph 131
6.5: The graph after a, e, c, d, f are in the source pool 131
6.6: Sort the equations 132
6.7: Ordering heuristic 133
6.8: Pseudo-code of SOR placement algorithm 134
6.9: Wirelength with different values of ω 137
6.10: Convergence of SOR and VPR 145
A.1: Timing analysis graph 160
A.2: Pseudo-code of SOR timing-driven placement 170

Chapter 1

Introduction

1.1 Motivation and Technology Trends

Field-Programmable Gate Arrays (FPGAs) represent a major manifestation of microelectronics as a key enabling technology. Since their inception in 1985, FPGA use has grown almost exponentially because they dramatically reduce design turnaround time and manufacturing costs for prototype circuits and small to medium volume electronic products. This is due to the fact that the logic and interconnect components in an FPGA are reconfigurable, making design debugging and modifications as easy as downloading another file to the chip. It has been estimated that more than 80% of design starts rely on the use of FPGAs, because they do represent an ideal means of establishing proof-of-concept, prototyping, and use in low and medium volume products. Due to the prohibitive cost of using the services of silicon foundries, many companies have turned away from designing and fabricating *Application Specific Integrated Circuits (ASICs)*, instead relying on FPGAs even for high-volume products. A major part of the impetus for this switch comes from the need to upgrade or modify the product functionality even after

releasing it in the market. The change is simply affected by downloading new configuration information into the FPGA. This makes it extremely cost-effective to make product/service changes in the field. This factor has become very important in today's highly competitive marketplace, where time to market and time to change or modify functionality in response to customer needs can make or break a company. From all indications, FPGAs have assumed a central role in digital system design. In fact, FPGA revenues are expected to grow from just under 4 billion in 2008 to just under 6 billion by 2011 [110].

Of course, technology must constantly change to meet the needs of the marketplace. When FPGAs first debuted in the mid-1980s, the Xilinx XC2064 (Xilinx, San Jose) FPGA had only 64 *Lookup Tables (LUTs)* and was used as simple glue logic. Today, Altera's Stratix IV (Altera, San Jose) and Xilinx's Virtex-6 both offer over 680,000 logic cells, plus a large number of hard-wired macro blocks such as embedded memories, DSP blocks, embedded processors, high-speed IOs, and clock synchronization circuits, representing more than a 10,000 times increase in logic capacity [110]. These modern FPGA devices are being used in entertainment, navigation, information, communication, and safety systems [1-5], including highly-complex *System-on-Chip (SoC)* components that contain both hardware and software elements.

Computer-Aided Design (CAD) plays a critical role in optimizing high-performance, high-density, and low-power design solutions using these high-end FPGAs. However, compilation times for designs are dominated by *placement* and *routing* time[†]. FPGA placement usually begins with a netlist of logic blocks and their interconnections. The result of placement is the physical assignment of all blocks on the target FPGA in a way that minimizes one or more specific objective cost functions (e.g., wirelength, delay, power dissipation, etc.). FPGA placement is similar to the more general ASIC placement problem in the sense that all blocks must be arranged inside a prescribed region on the chip such that no two blocks overlap and the estimated wirelength needed to implement the connections is minimized. However, it differs from the ASIC problem in that both the

[†] A detailed overview of the FPGA design flow is given in Chapter 2

size of the logic blocks, the type of logic blocks available, and the location that the blocks can occupy on the chip is fixed. Thus, FPGA placement can be viewed as a *more constrained* version of the general ASIC placement problem. In practice, placement has a significant impact on the performance and routability of circuit design, especially in nanometer designs because a placement solution, to a large extent, defines the amount of interconnect in the design, which now becomes the bottleneck of circuit performance.

FPGA routing is similar to the general ASIC routing problem in that all nets (wires that must be connected) must be successfully routed subject to timing constraints. However, FPGA routing is *more constrained* in the sense that it can use only the prefabricated routing resources on the FPGA, including available wire segments, programmable switches, and multiplexers. Therefore, achieving 100% routability is more challenging than ASIC routing. Moreover, a poor placement cannot be improved later by a high-quality routing.

For today's modern designs, FPGA placement and routing times are growing much more rapidly than the available computation power. While current CAD algorithms provide high-quality solutions, they often require great amounts of CPU time. For many circuits, this compile time can be in the order of tens of CPU hours, which adversely impacts the use of FPGAs by hardware designers. This provides compelling motivation to explore new methods for fast compilation of designs.

The focus of this thesis is on the *FPGA placement* problem. The *objective* is to develop efficient and effective placement algorithms. The *strategy* is based on developing novel *analytic models* and *solution methods*.

1.2 Overview of the Placement Problem

The FPGA placement problem usually begins with a netlist of logic blocks and their interconnections. The result of placement is the physical assignment of all blocks on the

target FPGA, which minimizes one or more specific cost functions. A *formal* description of the FPGA placement problem follows:

Given a set of blocks $\boldsymbol{B} = \{b_1, b_2, \ldots b_m\}$, a set of signals $\boldsymbol{S} = \{s_1, s_2, \ldots s_n\}$, and a set of locations on the field-programmable gate-array $\boldsymbol{L} = \{l_1, l_2, \ldots l_p\}$, where $p \geq |\boldsymbol{B}|$. $\forall$ $b_i \in \boldsymbol{B}$, there is a set of signals $\boldsymbol{S}_{bi} \subseteq \boldsymbol{S}$, and $\forall$ $s_i \in \boldsymbol{S}$, there is a set of blocks $\boldsymbol{B}_{si}$, $\boldsymbol{B}_{si} = \{b_j \mid s_i \in \boldsymbol{S}_{bj} \}$. $\boldsymbol{B}_{si}$ only contains all the blocks that send or receive signal s_i, and $\boldsymbol{S}_{bj}$ only contains all the signals that are sent or received by block b_j . The goal is to assign each block $b_i \in \boldsymbol{B}$ to a location $l_j \in \boldsymbol{L}$ such that the chosen objective function is optimized[†].

In practice, $\boldsymbol{B}_{si}$ is said to be a "signal net," and each such net specifies the connectivity of the original circuit. Typically, each block belongs to *several* nets. Locations on the FPGA typically correspond to *Configurable Logic Blocks* (*CLBs*) or *I/O pads*. CLBs are used to implement logic, while the pads are used for input/output to and from the FPGA. These blocks have distinct connection points on their boundary (called pins), which are used to provide each net a unique connection point.

When performing placement, the most basic objective is to minimize the *wirelength* required to complete the routing. Routing cost is used because reducing it reduces a number of associated design parameters. By reducing the routing length, the routing resources required by all interconnections are reduced. This results in an increase in circuit speed due to the reduction in connection capacitance and resistance. Power consumption, which is another important parameter to measure the quality of an FPGA implementation, is reduced too [12]. If the objective of the placement tool is to minimize the routing cost, the process is known as *wirelength driven placement*. There are other objective terms that can be added to the original cost function to directly optimize various design goals. For example, placement can be performed to minimize the length of the critical path to meet timing constraints, referred to as *timing-driven placement*. Circuits implemented on an FPGA are synchronous and, therefore, are driven by a clock.

[†] For simplicity, we leave off any discussion regarding the need to deal with timing constraints on signal nets

Minimizing the length of the critical path to meet timing constraints has the effect of maximizing the speed at which the circuit can be clocked.

In this thesis, the focus is primarily on using a near-linear wirelength objective in the analytical placement algorithms that we present. However, we also present a novel multi-objective analytical model that seeks to simultaneously optimize wirelength and critical-path delay.

A tiny problem is given in the (simplified) illustration in Fig. 1.1. (The illustration is simplified in that, for the sake of clarity, all of the programmable routing resources on the FPGA have been omitted. A much more detailed illustration is given in Chapter 2 which shows the typical routing resources available on an FPGA.) I/O pads are shown as shaded squares; CLBs are shown as non-shaded squares; and signal nets are shown as connections between the I/O pads and CLBs. Fig. 1.2 shows a "bad" placement that does not minimize total wirelength. As discussed, minimizing wirelength is important because excess wirelength degrades the performance of the final circuit. Moreover, excessive wirelength may lead to congestion in different parts of the chip making routing impossible (due to the limited and fixed number of routing resources available on the FPGA). Figure 1.1 shows a much better placement, from a wirelength perspective.

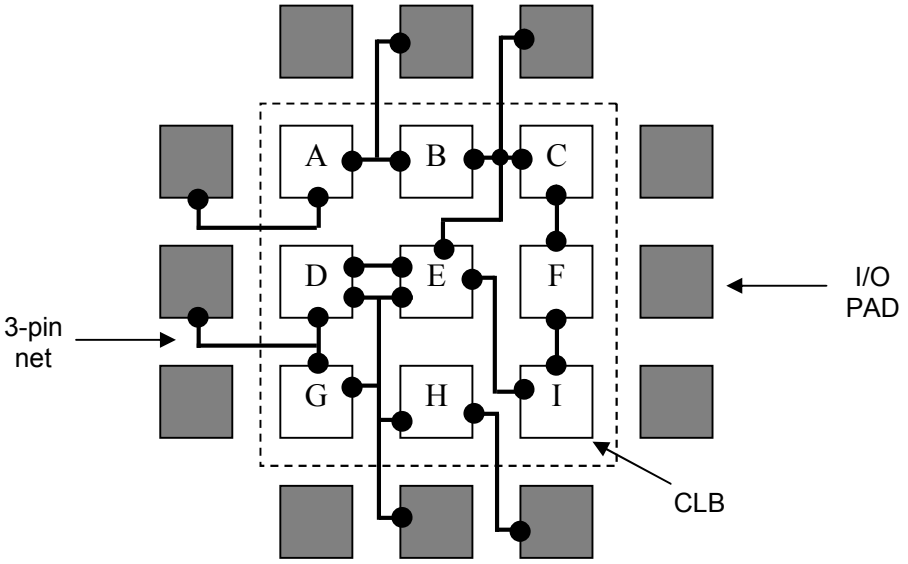


Figure 1.1: A "good" placement.

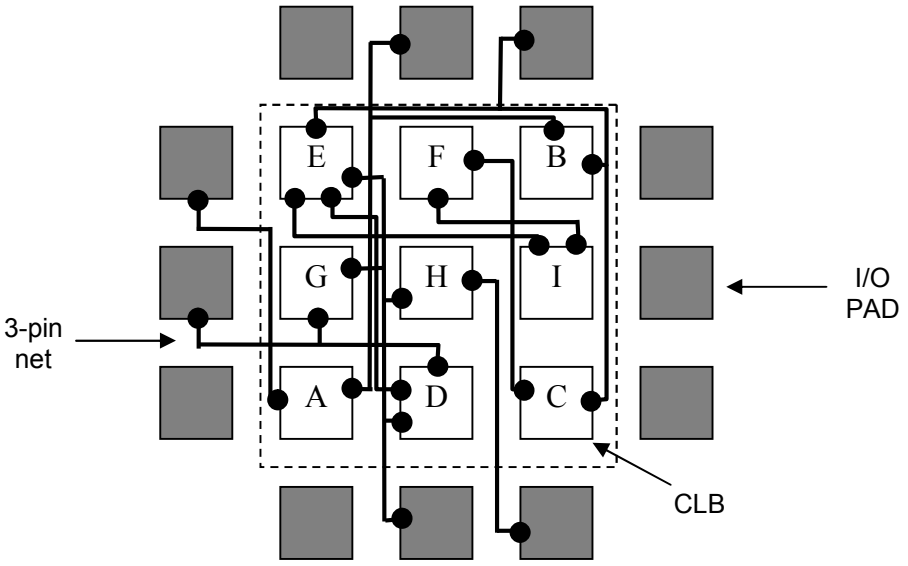


Figure 1.2: A "bad" placement.

1.3 FPGA Placement Methods

Over the last few years, many FPGA placement algorithms have been proposed to handle the objective of wirelength minimization. However, as the placement problem is NP-hard [10], no polynomial-time algorithm is known to produce an exact solution. Therefore, most algorithms are heuristic, seeking to find "good" solutions in "reasonable" amounts of time. Historically, these algorithms have been divided into three classes: partitioning-based placement [13][14], iterative improvement [15], and analytical-based placement [16-26].

In *partitioning-based placement*, a circuit is recursively bisected, minimizing the number of cuts of 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 highly connected blocks in one partition. These kinds of methods are good from a "global" perspective, but they do not *directly* attempt to optimize wirelength, timing, or routability. Therefore, the solutions obtained are inferior compared with other placement methods. However, partitioning methods run fast, and are normally used in conjunction with other search techniques, such as local search [15] for further quality improvement or quadratic programming [17].

Iterative methods, on the other hand, start with an initial placement and seek improvements by searching for small perturbations in the neighbourhood of the placement that result in better solutions. For FPGA placement, perturbations are location swaps (pair-wise moves) between blocks or moves (in the case where one of the "swapped blocks" is an empty location on the FPGA). The well-known *Versatile Placement and Routing* (*VPR*) [27][28] package for FPGA placement and routing uses the *Simulated-Annealing* (*SA*) method as its optimization engine for placement.

The simulated annealing algorithm simulates the annealing process that is used to temper metals. Given an initial placement configuration, a change to that configuration is made by either swapping the positions of two blocks, or moving a single block to an

unoccupied location on the FPGA. In simulated annealing, all swaps (or moves) that result in a decrease in cost are accepted. Swaps (or moves) that result in an increase in cost are accepted with a probability that decreases over the iterations. More specifically, moves and swaps that deteriorate the solution are accepted in SA with a probability of $e^{-\frac{\Delta C}{T}}$, where ΔC is the change in cost, and T is analogous to temperature in the metal-crystallization process. The change of T is referred to as an *annealing schedule*. Initially, T is set to a high value such that most inferior solutions can be accepted. This helps the search escape the many local optima it is likely to encounter as it begins to explore the problem's search space. As the annealing process continues, T gradually decreases (cools), reducing the probability of accepting poor solutions. This causes the search to slowly turn its focus away from exploring the search space in a global fashion to exploiting the current region of the search space. In the final state, T usually is only a small fraction of its original value, and almost only improving solutions are allowed; thus, the primary focus of the search becomes one of pure exploitation.

In the context of FPGA placement, it is too expensive to determine the exact configuration of routing resources needed to realize physical connections between the blocks, which is another NP-hard problem [10]. Besides distance (wirelength) between connections, there may be constraints on the number of wires sharing a channel, the allowed length or number of turns for certain wires, the availability of routing junctions, etc. For this reason, the routing cost is *approximated* during placement. The speed and accuracy of routing-cost estimation has a significant effect on the overall performance of any placement method. For example, VPR [27] (and many similar methods) employ the well known *Half-Perimeter Wirelength (HPWL)* model [30][31] to estimate the wirelength of a net. The wirelength is approximated by half the perimeter of the smallest bounding rectangle that encloses all terminals in the net. For a net with two or three terminals, the routing cost is accurate. However, when there are more than three terminals, a factor [27] can be introduced to compensate for the fact that HPWL underestimates the wire length required to connect all blocks. Using the HPWL model, VPR has achieved similar or higher quality solutions, compared with other types of

placement methods. However, because SA-based approaches must test an enormous number of possible swaps and moves, and because the annealing schedule required to find high-quality solutions is typically slow, the improvement over other placement methods comes at the cost of significantly longer run times.

In order to improve the runtime of SA-based approaches, some researchers apply multilevel techniques [66]. Multilevel optimization starts with multilevel clustering [82]. It requires cluster sizes at each level to be the same to facilitate pair-wise exchange at each level later on by simulated annealing. The clustering begins with a cluster with a random seed occupying an arbitrary slot in the cluster. Then, it grows the cluster by adding a logic block with the highest connectivity, measured by the summation of the shared nets between the block and the cluster. If all blocks on a net belong to the cluster, that net is absorbed. When the cluster is full, a new cluster is started with a random seed. This process is repeated until all blocks are clustered. The result is a clustered netlist with the absorbed nets removed. Then, it proceeds to create the next level of the clustering hierarchy. After the clustering hierarchy is created, low-temperature simulated annealing is performed at each level of the clustering hierarchy. During de-clustering from a coarser level to a finer level, the position of each cluster (or logic block) in the finer level is determined by the mean of the positions of the I/O pads and the parent clusters that are connected to it. Multilevel techniques speed up SA-based approaches at the cost of quality.

The last category of FPGA placement algorithms, and the focus of this thesis, is *analytical* placement methods. Rather than evaluate many small-scale provisional modifications (like iterative improvement methods), analytical placement methods typically tackle the problem from the top down by considering global (block and I/O pad) connectivity. They include both force-directed [16] and quadratic-programming [17][18][95] methods. The force-directed method introduces attracting, repelling, and other additional forces, and then solves a linear equation system using these forces. In contrast, the quadratic programming (QP) method solves a sequence of quadratic programming problems derived from the circuit connectivity information. On each

iteration, additional constraints are added to restrict the movement of blocks in order to gradually reduce the amount of block overlap.

Analytical methods are widely used for ASIC placement, but not as widely used for FPGA placement. Unlike ASICs, FPGAs are pre-fabricated before logic designs. Consequently, the placement solutions (in the form of the x- and y-coordinates of all logic blocks and I/O blocks) must be integers. However, the solutions obtained by solving equation systems are non-integers, which are acceptable in ASIC placement but are illegal in FPGA situation. In FPGA placement, the solutions directly obtained by solving equation systems must be legalized. This legalization procedure is usually performed at a cost of sacrificing placement quality.

Recently, graph-based approaches have been combined with analytic methods to better capture the true cost of using routing resources on the FPGA [95]. The algorithm in [95] views the placement task as an embedding of a graph (representing the netlist) into a chosen metric space. It first defines an analytic metric of "distance" in terms of the total delay through switches on the FPGA routing architectures, and then uses it to construct a metric space that captures FPGA performance. Next, the netlist graph is embedded into the metric space based on a binary quadratic assignment formulation, which is solved with a heuristic technique based on matrix projections followed by online bipartite graph matching. At last, the resulting solution is improved using low-temperature simulated-annealing method for local optimization.

In general, analytic placement is very promising as both the force-directed and quadratic-programming methods, if implemented correctly, have the potential to produce good solutions in small amounts of time. However, one of the primary considerations when implementing an analytic method is the form of wirelength model to use. Although HPWL [31] is widely used by iterative-improvement based methods, the fact that it is not continuously differentiable makes it difficult to employ in analytic methods that rely on the presence of first- and second-order partial derivatives. Moreover, HPWL ignores the relative positions of blocks inside the bounding box, despite the fact that the position of

these blocks has a direct affect on the total wirelength required. Due to these limitations, HPWL is rarely used by analytic methods directly. Analytic methods typically begin by transforming a hypergraph representation of the original circuit into a graph, where each (hyper) edge is modeled as a star [21] or a clique [21]. The actual effect of these models depends on the type of objective used. As discussed above, analytic FPGA placement algorithms commonly utilize a squared (quadratic) wirelength objective, as this allows efficient quadratic programming techniques to be applied [18] – something that is very important from a performance perspective. However, the quadratic wirelength objective has the effect of overemphasizing the optimization of longer nets at the expense of shorter nets. To compensate, some analytic methods have tried using a *regularized* linear wire-length estimate [17][21] in the context of ASIC placement. However, minimizing regularized linear wire-length is computationally more difficult than minimizing squared wire-length. Moreover, regularized linear wire-length results in lower-quality solutions compared with HPWL, again in the context of ASIC placement.

In this thesis, we propose to employ a new, *near-linear* wirelength model that is both differentiable (and hence suitable for use with the analytic methods we present later in the thesis) and accurate (does not overemphasize the optimization of longer nets). The employment of a near-linear objective into an analytic placement method is not new. For example, in the context of ASIC placement, the Gordian-L [17] uses iteratively refined piece-wise "linear" approximation of wire length. Compared with Gordian, which uses a traditional quadratic objective, Gordian-L is generally observed to lead to higher-quality placements. Moreover, the term-wise scaling approached used in Gordian-L does not change the properties of the underlying wirelength matrix. Hence, the same fast numerical techniques can be used to solve sequences of linear systems of equations arising in both formulations.

The proposed wire-length model, which we call *Star+*, is a variant of the well-known star model [21] (which was originally proposed for estimating wirelength in the context of ASIC placement) but with some key differences[1]. By using a near-linear wire-

[1] Differences between *Star+* and existing models are described in detail in Chapter 3

length model in our analytic optimization engine, wirelength can be more accurately estimated compared to the quadratic wirelength objective. This provides another avenue for analytic methods to follow. However, there is an important caveat. By employing a non-linear objective the resulting system of equations that must be solved is no longer linear, but non-linear; thus, making the resulting optimization problem (equation system) harder to solve. To compensate, we propose both theoretical as well as heuristic modifications to standard methods (conjugate gradient [32] and successive over-relaxation [33]) for solving systems of non-linear equations to improve their runtime performance. The goal is to develop analytic methods that are both effective and efficient.

1.4 Contributions

The contributions that this thesis makes are summarized as follows:

- We present a new near-linear model for estimating wirelength, called *Star+*. The model is based on the star model [21], but unlike HPWL, is directly differentiable, making it suitable for use with analytical methods. Another feature of the Star+ model is that the computation of ΔC caused by the swap of two blocks always runs in $O(1)$ time. This feature makes it suitable for SA-based methods, too. The Star+ model is also accurate. Our results show that when the Star+ model replaces the HPWL model in VPR [27][28], the quality of placements are similar with respect to total wire length and channel capacity, but Star+ produces placements with, on average, 6-9 percent smaller critical-path delay with no additional effort.

- We propose a novel non-linear *Conjugate Gradient (CG)* placement algorithm, which generates placement solutions by minimizing an objective function based on the Star+ model. The proposed method is more efficient than traditional conjugate gradient method [19] for FPGA placement. When using the traditional CG method, the Hessian matrix of the objective function has to be calculated on each iteration. This is not a problem when the Hessian matrix is sparse. But in the case of the

FPGA placement, the Hessian matrix is dense and, therefore, the computation of the Hessian is $O(n^2)$. In the non-linear CG method proposed here, we avoid computing the Hessian directly, and reduce the time complexity of an iteration to $O(n)$.

- We introduce a second analytic placement algorithm based on the *Successive Over Relaxation (SOR)* method [33]. SOR is also based on the Star+ model and has the same runtime complexity as CG placement. However, by properly arranging the sequence for calculating each equation of the nonlinear equation system, SOR-based placement runs faster by a constant amount (approximately 6.9 times as fast).

- In order to obtain non-trivial solutions of a nonlinear equation system, some logic and/or I/O blocks must temporarily be assigned to fixed locations on the FPGA chip. This assignment is *called pre-placement*. We present a novel pre-placement algorithm (called *Shrubbery*) that pre-places certain blocks using a method, which runs in $O(n \log n)$ time.

- We develop an analytical timing-driven placement algorithm by adding timing-driven factors into the original objective function used by both non-linear CG and SOR. The timing-driven model that is employed is the same as that used in other timing-driven placement methods, including VPR [27][28].

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces the FPGA architecture that we target throughout the remainder of this thesis, and gives an overview of previous works including a detailed comparison of different wirelength estimation and timing models, and a brief analysis of current placement algorithms. Chapter 3 describes the accuracy, computational complexity, and the differentiability of the Star+ model. Chapter 4 presents the non-linear CG placement algorithm based on the Star+ model.

Chapter 5 introduces the pre-placement algorithm – Shrubbery. Chapter 6 gives the SOR placement algorithm. Chapter 7 summarizes our research results and gives suggestions for future work. Finally, Appendix A describes the timing-driven placement.

Chapter 2

Relevant Background

In this chapter, we provide the necessary background material. In Section 2.1 we describe the basic island-style FPGA architecture that we assume throughout the remainder of this thesis. Section 2.2 describes the typical FPGA design flow. In Section 2.3, we discuss previous relevant work related to the FPGA-placement problem, while in Section 2.5 we describe the main wirelength estimation models employed by these placement methodologies. In Section 2.5 we provide a brief summary that seeks to place the work that we are proposing in this thesis in proper relation to the previous work. Finally, in Section 2.6 we introduce the benchmarks that will be used to validate the effectiveness of our placement algorithms.

2.1 FPGA Architecture(s)

There are various types of FPGA architectures available from different vendors including Xilinx, Altera, Actel, Lucent, and QuickLogic. Although the exact structure of these FPGAs varies from each other, all FPGAs consist of three fundamental components (as seen in Fig. 2.1):

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; and,
3. fixed, as well as programmable, routing resources used to realize all required interconnections between the blocks.

Based on their routing architectures, current commercial FPGAs can be classified into three groups: *island-style* FPGAs, *row-based* FPGAs and *hierarchical* FPGAs. As the placement methods mentioned in this thesis are aimed mainly at island-style FPGAs, we give a brief introduction of this FPGA architecture next.

Employed by many vendors, the island-style FPGA architecture is characterized by its two-dimensional symmetry. The architecture contains a square array of logic blocks surrounded by routing resources (wire segments and programmable switches). Logic blocks in this architecture are referred to as *Configurable Logic Blocks* (*CLBs*) and are arranged as a symmetrical array. Routing tracks have a *Manhattan* geometry; that is, they are either horizontal or vertical. Figure 2.1 shows a generic model of this kind of FPGA, an architecture that we assume throughout the rest of this thesis.

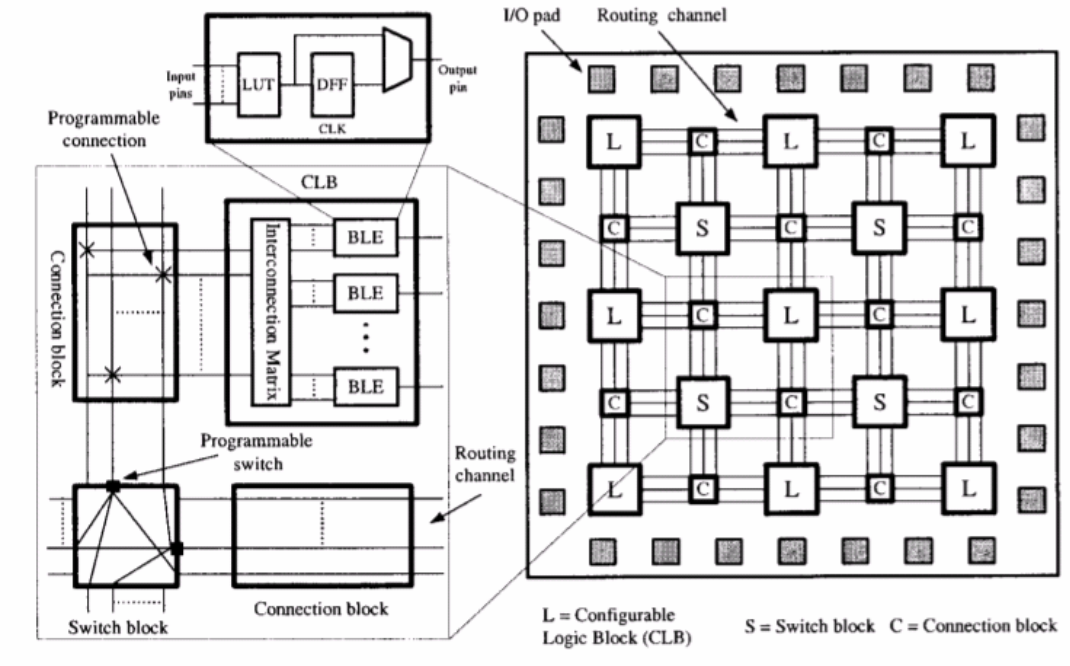


Figure 2.1: Architecture of Island Style FPGA.

The detailed routing structure consists of three components: *connection blocks*, *switch blocks*, and *routing channels*. A connection block is used to connect a CLB to the routing channels via programmable connections. The pins of each CLB pass uninterrupted through the connection block and have the option of "fusing" to some channel segments. The switch block is a switch matrix that is used to connect wires in one channel segment to other wires. Depending on the topology, each wiring segment on one side of a switch block 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. A CLB in most commercial FPGAs consists of one or more *Basic Logic Elements (BLE)*. Each BLE usually consists of a *Look Up Table (LUT)* and a register, as shown in Fig. 2.1. The underlying concept behind a LUT is relatively simple. A group of input signals is used as an index (pointer) to a lookup table. The contents of this table are arranged such that the cell pointed to by each input combination contains the desired value. In general, an *n*-input LUT can implement any possible *n*-input combinational circuit.

2.2 FPGA Design Procedure

As implementing a circuit using FPGAs involves the configuration of millions of programmable gates and switches, it is impractical for designers to specify all of the states for these components. Rather than setting the gates and switches to their proper states directly, designers describe the circuit to be implemented on the FPGA at a high-level of abstraction, typically using a Hardware Description Language (HDL) (or, in rare cases, using schematic entry). Then, *Computer-Aided Design* (*CAD*) tools convert this high-level description into a configuration file that specifies the states of all the programmable resources on the FPGA. Figure 2.2 shows a typical design flow.

Inputs to the design flow typically include the HDL specification of the design, design constraints, and a specification of the target FPGA. Each of these inputs is described below:

- **Circuit description**

  During the 1980s, schematic capture programs allowed engineers to create circuit (schematic) diagrams interactively. However, towards the end of the 1980s, as designs grew in size and complexity, schematic-capture tools began to run out of steam. Today, most FPGA designers use design tools and flows based on the use of HDLs. The most widely used design specification languages are Verilog [37] and VHDL [36], which are used at the *Register-Transfer Level (RTL)* to specify the operations in each clock cycle. Recently, there has been a trend toward moving to specification at a higher level of abstraction, using languages like System-C [89] or Handel-C [90], or domain specific languages, such as MatLab [91] or Simulink [92]. These languages allow a designer to focus on the algorithm/behavior that is to be implemented on the FPGA, rather than having to focus on the cycle-accurate description of the design.

- **Design Constraints**

  Design constraints typically include the desired operating frequencies of different clocks employed in the design, bounds on path delays from input pads to output pads, from input pads to registers (setup times), and from registers to output pads (hold times), or delays between specific pairs of registers. Moreover, the user (or synthesis tool) may specify constraints requiring that certain elements or blocks be placed at certain physical locations on the FPGA.

- **Target FPGA**

  The third input is the type of target FPGA to be used. Most FPGA vendors provide a wide variety of FPGA architectures that differ with respect to size, performance, power, and cost. Typically, a designer will start with a small (low capacity) FPGA with nominal speed-grade. However, if the synthesis effort fails to map the design onto the FPGA or fails to meet performance requirements, the user will have to upgrade to a larger (higher capacity and/or higher speed grade), but more expensive device. This fact clearly underscores the need to have better synthesis tools, as their quality directly impacts the performance and cost of FPGA designs.

We now briefly describe the FPGA design flow in Fig. 2.2. Given a design (described in a suitable HDL), set of design constraints, and a target FPGA device, the overall FPGA synthesis goes through the following steps.

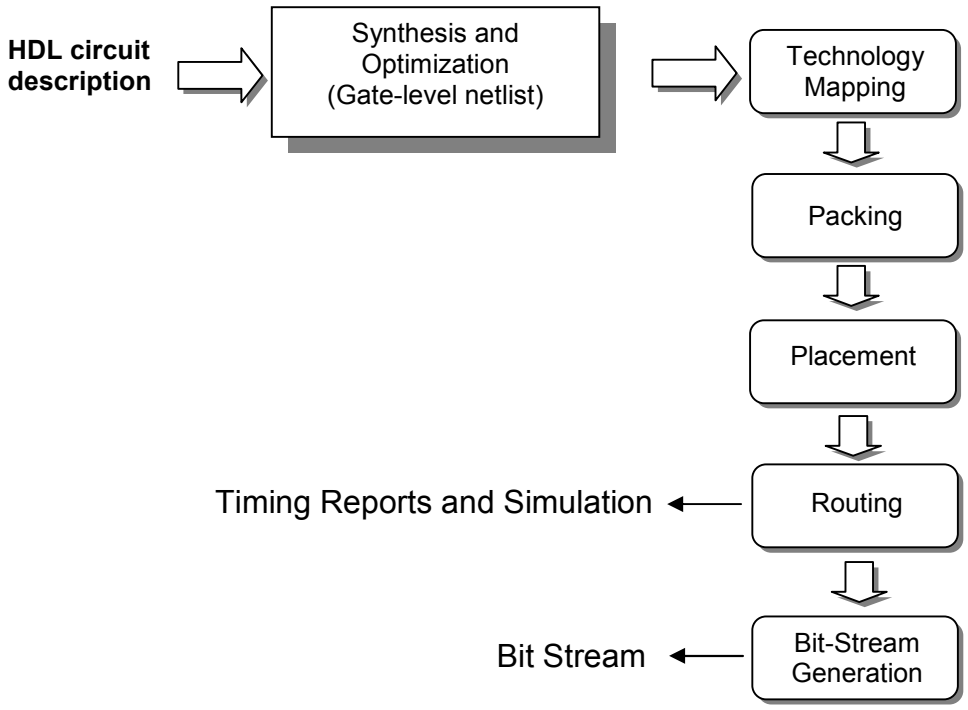


Figure 2.2: Typical FPGA design flow

- **Synthesis and logic optimization**

  This step involves synthesizing the designer's original hardware description into a logic design, using CAD tools. This involves identifying both *datapath* operations and *control* logic. Identification of the latter is important, as modern FPGAs often have architectural support such as embedded multipliers and adders with fast carry chains. Next, complex logic is broken down into simple logic expressions, which are converted into a netlist of basic gates. This netlist of basic gates is then transformed into a netlist of FPGA logic blocks. During this stage, technology-independent logic optimization is often involved to remove any redundant logic and simplify logic wherever possible [38][39].

- **Technology mapping**

  Once optimized, the netlist of logic gates has to be mapped into *Look-Up Tables (LUTs)*, which will be packed into FPGA logic blocks later on. In this context, mapping refers to the process of associating entities such as gate-level functions in the net-list with the LUT-level functions available on the FPGA. This is not a one-for-one mapping because each LUT can be used to represent a number of logic gates. In practice, mapping is a nontrivial problem because there are a large number of ways in which the logic gates forming a netlist can be partitioned into the smaller groups to be mapped into LUTs.

- **Logic block packing**

  Following the mapping phase, the next step is *packing*, in which the LUTs and registers are packed into the CLBs. A CLB usually contains more than one look-up table and flip-flop. Logic block packing groups several look-up tables and flip-flops into each CLB with the objective to minimize the interconnections between CLBs. During packing, constraints such as the maximum number of inputs per CLB have to be taken into account. In practice, packing is also a nontrivial problem because there are myriad potential combinations and permutations.

- **Placement**

  Following packing, we move to *placement*. After logic block packing, the circuit has been transformed into a list of blocks (CLBs) and nets (pins that must be connected) that specify the connections between these blocks. Placement algorithms now assign these logic blocks to physical locations on an FPGA with optimization goals to minimize the required wiring (wirelength-driven placement) [20][21], balance the wiring density (routability-driven placement), and/or to maximize circuit speed (timing-driven placement) [28]. In general, placement is an NP-hard problem and represents one of the main bottlenecks in the FPGA design flow, as FPGAs can contain hundreds of thousands of CLBs. In fact, placement times for industrial-strength applications are often so long that most designers today would be very

happy if these times could be reduced to allow for one full-compile (from HDL to FPGA) per day[†].

- **Routing**

  Following placement, routing algorithms (global and detailed) identify which specific wire segments should be used and which programmable switches should be turned on to connect all the nets specified in the netlist file. The optimization goals are typically to reduce the amount of routing resources required to connect all the nets, and/or minimize the delay.

- **Simulation**:

  Following place-and-route, we have a fully routed physical (CLB-level) netlist. At this point, a *static timing analysis* utility is run to calculate all of the input-to-output and internal path delays and also to check for any timing violations (setup, hold, etc.) associated with any of the internal registers. Interestingly, if the designer wishes to re-simulate their design with accurate (post place-and-route) timing information, they have to use the FPGA tool suite to generate a new gate-level netlist along with associated timing information in the form of an industry-standard file format, called *standard delay format*. The main reason for generating this new gate-level netlist is that once the original netlist has been coerced into its CLB-level equivalent – it simply isn't possible to relate the timings associated with this new representation back into the original gate-level incarnation.

- **Bit-stream file**:

  The last process in the FPGA-design flow inputs the mapped, placed, and routed design and generates a bit-stream file that can be downloaded to the target FPGA chip. This bit-stream file stores the configuration of programmable blocks and routing resources.

[†] informed through private communication with Altera's Vaughn Betz

As the focus of this thesis is on developing fast analytic methods for performing placement, we now turn our attention to the literature and existing placement methods.

2.3 Placement Methods

In the last two decades, abundant placement approaches have been proposed to deal with the broadly used objective of wirelength minimization. These methods can be roughly divided into 5 categories: (i) partition-based placement; (ii) simulated-annealing based placement; (iii) multi-level based placement; (iv) analytic placement; and (v) other placement methods, including hybrid and parallel methods.

2.3.1 Partition-Based Methods

Partitioning-based placement methods [40][41] are also referred to as *min-cut* methods. The basic idea is to use a graph-partitioning algorithm to divide a region of the FPGA into two halves. A circuit partitioning algorithm is then applied to determine which logic block goes to which half with the goals of minimizing the number of cuts in the nets across the boundary between two partitions, and placing highly-connected blocks in the same partition. These procedures are recursively repeated until each partition contains only a few blocks. The advantage of partitioning-based placement algorithms is that they run very fast. As they use a divide-and-conquer strategy, where large problems are divided into small sub-problems, partitioning-based methods significantly reduce the problem search space. However, since the cut size is not an exact function of wirelength, timing or routability, the quality is not as good as other placement strategies.

A notable (and recent) contribution is PFFF [93], which uses the start-of-the-art multi-level (see Section 2.3.4) partitioner hMETIS [94] as its partitioning engine. The experimental results in [93] report a slight degradation in solution quality, with a 3-4 times improvement in runtime, compared with the state-of-the-art academic placement tool, VPR [27][28].

2.3.2 Simulated Annealing

Simulated Annealing (*SA*) is widely used for solving combinatorial optimization problems and has been applied to circuit placement (in the context of ASIC design) successfully [27][54]. As the name suggests, this method mimics the process used to gradually cool 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 combinatorial-optimization problem. SA-based placement algorithms are readily adapted to handle any known form of constraint and optimization goals [55]. In addition to their hill-climbing property, their ability to accept non-improving moves enables them to escape local optima.

Simulated annealing belongs to a class of *stochastic* search algorithms that accept any randomly encountered solution within the neighborhood of solutions being currently considered with a defined probability. In practice, new neighbouring solutions are created incrementally from the current solution. If the cost of the new solution (derived using an appropriate objective function for the problem) is reduced, the new solution is accepted. However, if the new solution is found to have inferior cost, the new non-improving 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. The parameter T is used to control the probability of accepting non-improving moves. In general, a high value of T causes the search to become random, while a low value of T causes the stochastic algorithm to revert to an ordinary hill-climber. Thus, an appropriate value of the parameter T must be found (throughout the search) for the particular problem being solved.

The rate of change of T is referred to as the *annealing schedule*, and 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 non-improving solutions can be accepted. However, as the process continues, T is gradually decreased (simulating cooling), reducing the probability of accepting poor solutions. In the final states of the search, 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 to escape local optima, which often traps other search procedures.

Like all search methods, simulated annealing has both advantages and disadvantages. One advantage is that theoretical analysis [56] 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 SA compared with most other search procedures. However, there is precious little information on how to set the proper parameters for a particular implementation. Moreover, the runtime to find the globally optimum solution can become extremely large. Consequently, most current applications of simulated annealing employ simple, yet effective, approaches to obtain good, sub-optimal solutions.

2.3.2.1 Versatile Placement and Routing (VPR)

SA-based placement methods for FPGAs have been well studied [29][50][56][58][60]. In [103], Chen and Cong use a SA-based algorithm, called *SCPlace*, which performs clustering and placement simultaneously. There are two types of moves in their approach. The first type of move is the *block level move*, in which an entire *Clustered-based Logic Block* (*CLB*) is moved to a new location and swapped with another CLB if necessary. The second type of move is the *fragment level move*, in which only a *Basic Logic Element* (*BLE*) is moved to a new CLB and swapped with another BLE if necessary. After each move, the cost function is updated to decide whether to keep the move or not. In [27][28], Betz et. al. present the current state-of-the-art academic place-and-route tool, *Versatile Placement and Routing* (*VPR*). In addition to a mapping and routing tool, *VPR* contains an extremely effective SA-based placement tool, called *VPlace*. The VPlace tool follows the basic template of simulated annealing (see previous Section), but with several placement-specific enhancements like: (i) a new temperature updating scheme, which decreases the temperature faster when the move acceptance rate is very high or very low,

so that the annealing process spends more time at the most productive temperature regions (when a significant number of moves are being accepted); (ii) a limitation on the range of cell exchanges so that the move acceptance rate is as close to 0.44 as possible and for as long as possible; (iii) a linear congestion model that can be used when the channel capacity is non-uniform in the FPGA; (iv) and a faster method for incremental net bounding box updating. Overall, the VPR placement tool provides very good results and is widely used in the FPGA research community.

Figure 2.3 shows a pseudo-code description of the *VPlace* algorithm. As Fig. 2.3 indicates, *VPlace* first creates an initial solution by placing CLBs and I/O pads randomly into locations throughout the target FPGA (line 1). Some CLBs and I/O pads may remain unused; these blocks are marked as void blocks. Based on earlier work by Huang et. al. [58], the initial temperature T (line 2) is set to 20 times the standard deviation in cost after a set of N_{blocks} pairwise swaps (moves) have been attempted. (N_{blocks} is the total number of CLBs and I/O pads in the circuit.) The number of new configurations evaluated at this temperature is set to:

$$MovesPerT = innerNum * (N_{blocks})^{4/3} \qquad [59]$$

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

In [59][60] it is shown 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. [60].

$$R_{limit}^{new} = R_{limit}^{old} * (1 - 0.44 + \alpha) \qquad where$$
$$R_{limit} \in [1, maximum\ FPGA\ dimension]$$

Any attempted swap of blocks is allowed only within a square window, where the

length of each size of the window equals R_{limit}. A small value of R_{limit} ensures that only blocks close together are considered for swapping. These "local" swaps tend to result in an increase in the move being accepted. In practice, R_{limit} initially spans the entire FPGA, shrinks gradually as the search progresses and blocks find themselves settling in the correct regions, and finally reduces to 1 during the latter part of the search where only local refinement is necessary.

```
[1]     S = InitPlacement();
[2]     T = InitTemperature();
[3]     Rlimit = InitR_limit();   //set to whole chip initially
[4]     while( ExitCriterion() == false )  //outer loop
        {
[5]             while( InnerLoopCriterion == false )  //inner loop
                {

                //create a candidate solution from the current solution by
                //performing a random pair-wise move within the window
                //specified by R_limit
[6]             S_candiate = GenerateMove(S_current, R_limit);

                //Calculate change in cost
[7]             ΔC = Cost(S_candidate) – Cost(S_current)

[8]             r = random(0,1) // compute random number between 0 and 1

                //if ΔC ≤ 0, accept move; otherwise accept the move
                //with probability  e^(-ΔC/T)

[9]             if(ΔC ≤ 0 | | r  < e^(-ΔC/T))
[10]                    S_current = S_candidate;

                } //end of inner loop

[11]            Update(T);  //T_new = α * T_old
[12]            Update(R_limit);

[13]    } // end of outer loop

//return final placement solution S
```

Figure 2.3: Pseudo-code for simulated annealing [28]

The placement is improved by iteratively selecting random blocks and swapping their locations. The effect of each potential swap, on total wirelength, is calculated using the HPWL wirelength model described in Section 2.4.

Clearly, a robust FPGA placement tool must be able to effectively handle a wide variety of circuits with different sizes. Consequently, as the core of any SA-based implementation, the annealing schedule must automatically adapt to different circuits. The *VPlace* annealing schedule is based on the following observations and methodology: At the outset of the search when the temperature T is so high that almost every swap is accepted, the FPGA configurations randomly move from one configuration to another with no appreciable improvement in quality. Conversely, at the end of the search, when very few swaps are accepted due to the extremely low temperature T and (hopefully) high quality of the current placement, very little improvement in quality is obtained. Therefore, *VPlace* searches the problem space efficiently by increasing the amount of time spent on exploring the problem space in the middle part of the search, where more productive swaps are likely to be found and made. The exact update schedule for T in *VPlace* is as follows:

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

Finally, *VPlace* terminates when the temperature T falls below a certain fraction of the average cost per net (set of pins that must be connected). This makes the acceptance of any cost-increasing move almost impossible:

$$T_{end} = 0.005 * \frac{\textit{bounding box cost (HPWL)}}{\textit{total number of nets}}$$

With regards to computational complexity, the timing analysis for *VPlace* is performed once per temperature change, which is an $O(n)$ operation. At each temperature,

the inner loop of the placer is executed $O(n^{4/3})$ times; i.e., $O(n^{4/3})$ swaps are performed. In the inner loop is an incremental-bounding-box-update operation (see Section 2.4) that is worst case $O(k_{max})$, where k_{max} is the fanout of the largest net in the circuit. The average complexity of the bounding box update is $O(1)$ [28][61]. The overall result is that *VPlace* has an average runtime complexity, per temperature change, of $O(n^{4/3})$.

Currently, amongst academic tools, *VPlace* is considered to be the best – producing high-quality placements in reasonable amounts of time when tested with the MCNC [62] benchmark suite. Therefore, it has become the standard by which all other placement tools presented in the literature are compared. In this thesis, we will also be using *VPR* (placement and routing) as a baseline for comparison with our analytic technique. However, it should be noted that as FPGAs continue to grow in size, and the problem instances mapped to these FPGAs increase in size, SA-based placers, like *VPlace*, may fail to scale. Therefore, we plan to explore fast, analytic placement techniques.

2.3.3 Analytic Methods

Analytic algorithms are among the most promising methods for performing fast placements. These algorithms tackle the problem from the top down by considering global connectivity rather than evaluating many small-scale provisional modifications. They include both force-directed [16][22][105][106] and quadratic programming [17] [18][25][43][95][104][107] methods. The force-directed method introduces attracting, repelling, and other additional forces and then solves a linear equation system using these forces. In [16] and [22], Eisenmann et al. introduce additional forces to each cell based on cell distribution to pull cells away from dense regions. Xu and Khalid [104] use quadratic programming technique to minimize the squared distance, and then use low temperature Simulated Annealing to refine the placement. Etawil et al. [105] add repelling forces for cells sharing a net to maintain a target distance between them and attractive forces by fixed dummy cells to pull cells from dense to sparse regions. Hu et al. [106] introduce the idea of a fixed-point as a more general way to add forces for cell spreading.

The quadratic programming method solves the placement problem by solving a sequence of quadratic programming problems derived from the circuit connectivity information. This type of method maintains a whole view of the placement problem, and hence is often used as a global optimization method [17][25][104][107]. In general, quadratic programming methods take a hypergraph netlist as their input, and then seek to minimize the total squared wire length. The objective function follows:

$$\Phi(x,y) = \frac{1}{2}\sum_{i,j} W_{ij}[(x_i - x_j)^2 + (y_i - y_j)^2]$$

where *x, y* are the coordinates of a logic block. W_{ij} is the weight of the edge that connects block (x_i, y_i) and block (x_j, y_j). Since two blocks may be connected by more than one net, the hypergraph first needs to be converted into a weighted graph. Two models can be used for this conversion: *clique* and *star*. A *clique* model introduces $k(k\text{-}1)/2$ edges with each edge connecting each pair of blocks incident to a k-pin net, while a *star* model creates a new node at the center of gravity of the net and introduces k edges with each edge connecting a block and the center.

The previous objective function is often written in matrix form as shown below:

$$\Phi(x,y) = \frac{1}{2}(x^T Ax + y^T Ay)$$

where A is an $n \times n$ symmetric matrix, called the *Hessian* matrix, and n is the number of blocks. In order to obtain non-trivial solutions, some of the variables (x_i's, x_j's, y_i's and y_j's) must be fixed. Therefore, the above objective function can be rewritten as:

$$\Phi(x,y) = \frac{1}{2}x^T Ax + d_x^T x + \frac{1}{2}y^T Ay + d_y^T y$$

where d_x^T and d_y^T are *n*-dimensional vectors representing various constraints and fixed *x* and *y* values. As all *x* variables are independent of *y* variables and vice versa, this objective function can be separated into two functions with the same form. For the sake of simplicity, we only discuss the function in the *x*-dimension (the function in the *y*-dimension can be dealt with in a similar way): $\Phi(x) = \frac{1}{2} x^T Ax + d_x^T x$. This function is strictly convex and definitely positive and, therefore, its minimum is the point where (the gradient) $Ax + d_x^T = 0$. This linear equation system can be solved efficiently by a variety of standard techniques, including *Conjugate Gradient (CG)* [32][33] and *Successive Overrelaxation (SOR)* [33] methods.

A major concern with the quadratic programming is that it results in a placement with a large amount of overlap among blocks. It is reported in [16] that quadratic methods produce placements where 85% to 98% of the blocks overlap. Thus, to legalize the placement, researchers must apply various techniques that have the potential of degrading the quality of the original (infeasible) placement. Kleinhans et al. [17] and Kernighan et al. [53] use a bisection technique to recursively divide the circuit into two partitions until each partition has only one block and one CLB. Vygen [107] uses a quadrisection instead. Eisenmann and Johannes [22] use additional forces from higher density regions to coerce blocks to move into lower density regions. Mo et al. [51] introduce repulsive forces for overlapping cells and filling forces for lower density regions. Vorwerk and Kennings [16] apply min-cut partitioning before quadratic placement.

Another issue with quadratic programming is that the placement quality is sub-optimal since its objective function uses *squared* wirelength. To improve placement quality, subsequent refinement techniques are used after. For example, Viswanathan et al. [25] use a cell shifting technique for local refinement; Xu et al. [104] apply low temperature simulated annealing to improve the placement. Some other researchers use an approximate linear objective instead of quadratic objective. In particular, Kennings et al. [21][108][109] apply *regularization* techniques on linear wirelength (in the context of

ASIC placement). The objective function after regularization is:

$$\Phi(x) = \sum_{i>j} \alpha_{ij} \sqrt{(x_i - x_j)^2 + \beta} : Hx = b$$

where x_i and x_j are the x-coordinates of block i and block j; H represents various linear constraints. When $\beta \to 0$, $\Phi(x) \to \sum_{i>j} \alpha_{ij} \mid x_i - x_j \mid$. The regularized linear wirelength represents a better estimate of routing length compared with quadratic distance.

In summary, the primary advantage of analytic methods is their potential for short run times. However, the quality of analytic techniques is typically not as good as SA-based placers. Therefore, in this thesis we plan to develop placement algorithms, based on analytic methods that are not only fast, but also able to produce high-quality solutions.

2.3.4 Multilevel Clustering

As FPGAs continue to increase in logic capacity and functionality, so do the designs mapped to them. A recent approach to reducing the complexity of placing large designs (circuits) is to employ what is known as a *multilevel* strategy. Multilevel strategies construct a hierarchy of successively coarser problems from the bottom by recursive aggregation. They employ iterative improvement at each of the resulting levels, transfer these improvements up and down the hierarchy, and eventually terminate with a solution at the original, finest level.

The multilevel approach is illustrated in Fig. 2.4. As discussed, the procedure is essentially 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 of the clusters. The simplified problem makes the use of a traditionally time-consuming method, like simulated annealing, more feasible. A *declustering* process proceeds to restore the

original FPGA layout according to the previous placement result of the clusters. In this procedure, the flattened blocks should be as close as possible to their center-of-gravity [64][65] as possible. Finally, a localized improvement heuristic is executed to move blocks in small regions to achieve the final placement.

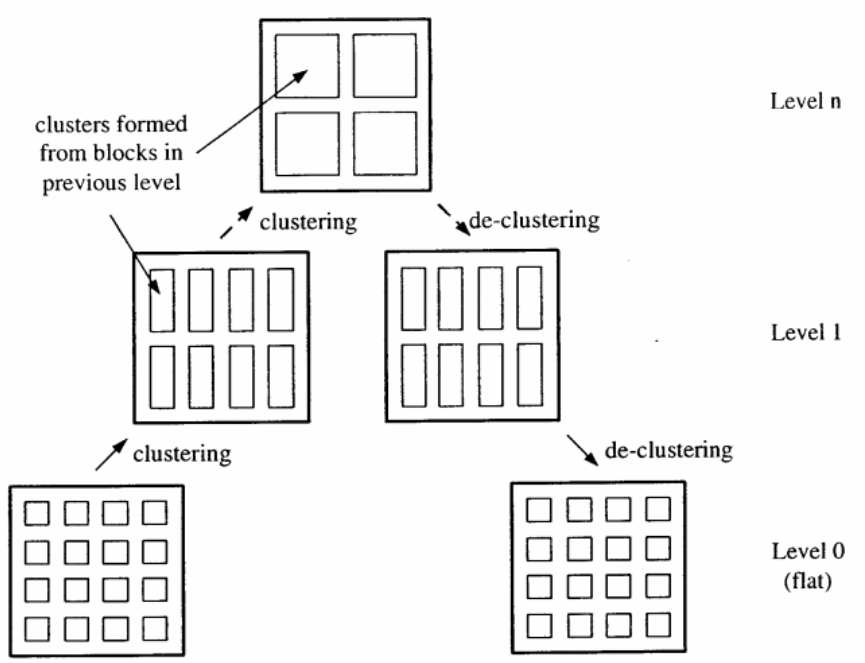


Figure 2.4: Multi-level clustering

Although multilevel strategies have the potential to improve the runtime of existing FPGA placement algorithms, only recently has the multilevel approach been applied, and then only in a limited way [66][93][96]. To the best of our knowledge, the first multilevel placement method described in the literature was Ultra-Fast Placement (UFP) [66]. The aim of UFP was to significantly reduce the runtime of VPlace. (Note that VPlace produces good results, but these results are not scalable as problem sizes increase due to the nature of the simulated annealing optimization engine employed by VPlace.) The approach described in [66] has the caveat that the size of the clusters at each level must be the same. In fact, they must be powers of 2 (e.g., 4, 8, 16, 32, …) to facilitate

pair-wise exchanges at each level using simulated annealing. The experimental results reported in [66] show a smooth runtime and quality trade-off. At one extreme, UFP achieves a 50 times speed-up over VPlace, but with 33% wire length overhead. The work in [66] uses an extremely effective simulated-annealing based optimization engine in a multilevel framework similar to that used by UFP. The results in [96] show that a 79% reduction in CPU time (compared with VPlace) can be achieved, with only a slight (less than 2%) reduction in solution quality. Recently, multilevel placement has become a very active research topic, with several high-quality multilevel placement methods being developed for standard cell designs [55][97][98]. It is likely that the multilevel placement techniques in these works can also be used to further enhance the quality of the work in [66][96]. However, we do not employ these multilevel strategies in the work in this thesis, but rather leave it for future work.

2.3.5 Other Approaches to Placement

In recent years there have been several novel placement algorithms that employ multiple placement techniques. For example, *Mongrel* [68] adopts a middle-down methodology in which a global placement solution is obtained by placing logic cells into coarse bins. During the placement phase, a *Relaxation-Based* local search methodology is applied to generate global complex modifications to the current placement. A novel *ripple* move [68] based legalization procedure is also presented. After the global placement is completed, a detailed placement is obtained by applying the optimal interleaving [68] technique. *Dragon2000* [69] uses a top-down hierarchical approach, and integrates the partitioning-based cut size minimization techniques and simulated-annealing-based wirelength minimization techniques. *mPL* [70] and *mPG* [71] are based on the multi-level framework to improve both runtime and quality of placement.

Several non-traditional approaches have also been tried for accelerating placement. For example, techniques for parallelizing simulated annealing have been used to accelerate *VPR* on expensive shared-memory machines (SGI Origin) or specialized

distributed memory multiprocessors (IBM-SP2) [47]. FPGA-based computing platforms to accelerate placement and routing have also been proposed in [48][49][50]. These methods often reduce the runtime of placement by orders of magnitude compared with a sequential algorithm. However, the quality of results that they produce is significantly worse than that obtained with other methods. In [72], a placement algorithm, called *NAP*, which runs in a ubiquitous network environment, is presented. The algorithm obtains speedups of 2-3 using a small number of machines connected on a local network. However, while readily available, this environment still requires the user to have access to a network and multiple machines. In [51], the previous placement algorithm is implemented using both multi-core and SIMD units resulting in speedups of 1.34 compared when implemented as a traditional serial algorithm. Although the speedups are modest, and lag well behind those of previously reported methods, they are immediate and more widely available. Very recently, strategies have been presented for parallelizing move-based heuristics on processors with multiple cores [99]. The experimental results show speed up of 1.3 times on 2 cores and 2.2 times on 4 cores.

Given that it is hard to achieve 100% routability, especially for the earlier generation of FPGAs (1990s), several attempts were made to combine placement and routing, so that the placement solution is assured to be routable [100][101]. However, in general, these approaches have not shown results that demonstrate the superiority of the combined approach. Moreover, given that modern FPGAs have much higher logic capacity and richer routing resources, one may question if it is feasible to compute or even necessary to carry out simultaneous placement and routing.

2.4 Wirelength Models

As the precise wire length for a given placement can only be known after routing, accurate and fast to compute wirelength estimates are required by FPGA placement algorithms. The main models include minimum Steiner-tree, half-perimeter wire length, clique, and star. A brief description of these models is given below.

- **Minimum Steiner Tree Model**:

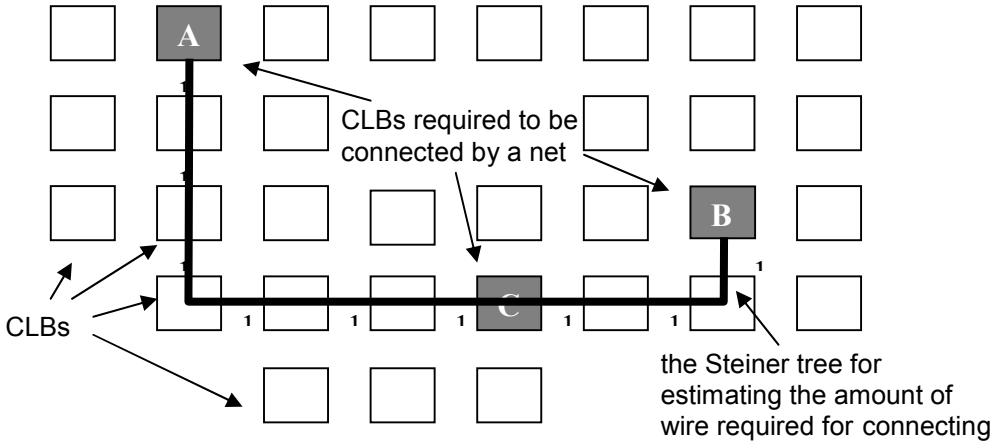
  A minimum Steiner tree [74] is a minimum-cost tree that spans a set of terminals. In a minimum Steiner tree net model, each input or output pin of a CLB is mapped to a terminal. Minimizing the wirelength required by a net is equivalent to finding a minimum Steiner tree. Figure 2.5 shows a net with three terminals. Figure 2.5(a) shows a Steiner tree that connects the net with a total wirelength of 9. Figure 2.5(b) shows a Steiner tree that connects the net with a total wirelength of 8. The Steiner tree in Figure 2.5(b) is the minimal Steiner tree, and is the optimal way to route the net. As the Steiner tree model does not consider the track capacity of each routing channel, the sum of the costs of all the minimum Steiner trees for the nets is usually less than the wirelength required by all the connections between CLBs. Moreover, the minimum Steiner tree problem is NP-hard [75]. More discussion on the Steiner tree problem and approximation algorithms for solving minimum Steiner tree problems can be found in [75-79].

- **Half-Perimeter Wirelength (HPWL)**

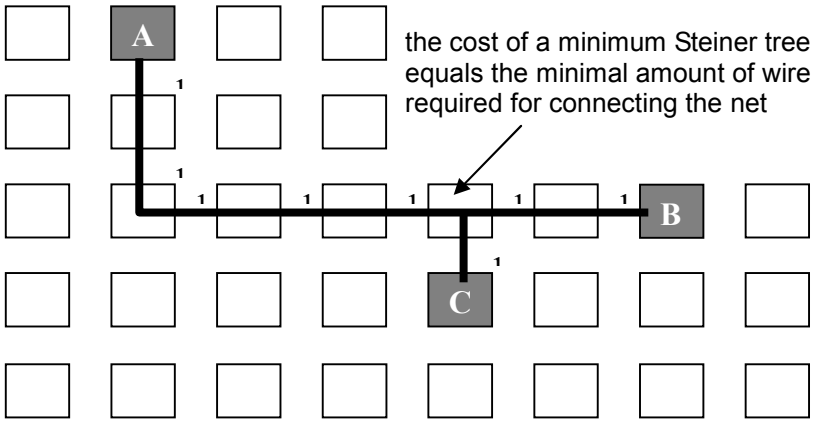
  The most commonly used wirelength model, and the one used by *VPlace*, is called the *Half-Perimeter Wirelength* (*HPWL*) model. The HPWL estimates the wirelength by half the perimeter of the smallest rectangle that surrounds all blocks in the net. Figure 2.6 shows a net connecting three blocks A, B and C. The bounding box (the smallest rectangle surrounding the net) shown in thin dashed lines has a perimeter of 16. The minimum rectilinear Steiner tree (shown with solid lines) has a total wirelength of 8, which is exactly half of the perimeter of the bounding box.

  Clearly, the HPWL model is exact for a net connecting two or three blocks (see Figure 2.6), but it underestimates the wirelength of a net connecting more than three blocks. To compensate for this underestimation, the HPWL model introduces a factor that is 1 for nets connecting 2 or 3 blocks, and gradually increases to 2.7933 for nets connecting 50 blocks. The formula for estimating the wirelength of net i is as following:

  $$Cost_{net\,i} = q(i_k) \times \{(\max_{\forall b \in net\,i} x_b - \min_{\forall b \in net\,i} x_b + 1) + (\max_{\forall b \in net\,i} y_b - \min_{\forall b \in net\,i} y_b + 1)\}$$



(a) A Steiner tree (also a minimum spanning tree) with total wirelength of 9



(b) A Steiner tree with total wirelength of 8

Figure 2.5: Steiner tree example with a 3-block net.

The values $\max_{\forall b \in net\,i} x_b$ and $\max_{\forall b \in net\,i} y_b$ are the largest coordinates in x- and y-dimension of all the blocks connected to net i, while $\min_{\forall b \in net\,i} x_b$ and $\min_{\forall b \in net\,i} y_b$ are the smallest coordinates. The value i_k is the cardinality of net i (the number of

blocks connected to net *i*). Table 2.1 gives the value of $q(i_k)$ when i_k is less than or equal to 50. For a net that has more than 50 blocks, the value of $q(i_k)$ linearly increases as follows: $q(i_k) = 2.7933 + 0.02616(i_k - 50)$.

Table 2.1: Weight of net with cardinality less than or equal to 50

i_k	$q(i_k)$	i_k	$q(i_k)$	i_k	$q(i_k)$	i_k	$q(i_k)$
1 – 3	1.0000	15	1.6899	27	2.1379	39	2.5064
4	1.0828	16	1.7304	28	2.1698	40	2.5356
5	1.1536	17	1.7709	29	2.2016	41	2.5610
6	1.2206	18	1.8114	30	2.2334	42	2.5864
7	1.2823	19	1.8519	31	2.2646	43	2.6117
8	1.3385	20	1.8924	32	2.2958	44	2.6371
9	1.3991	21	1.9288	33	2.3271	45	2.6625
10	1.4493	22	1.9652	34	2.3583	46	2.6887
11	1.4974	23	2.0015	35	2.3895	47	2.7148
12	1.5455	24	2.0379	36	2.4187	48	2.7410
13	1.5937	25	2.0743	37	2.4479	49	2.7671
14	1.6418	26	2.1061	38	2.4772	50	2.7933

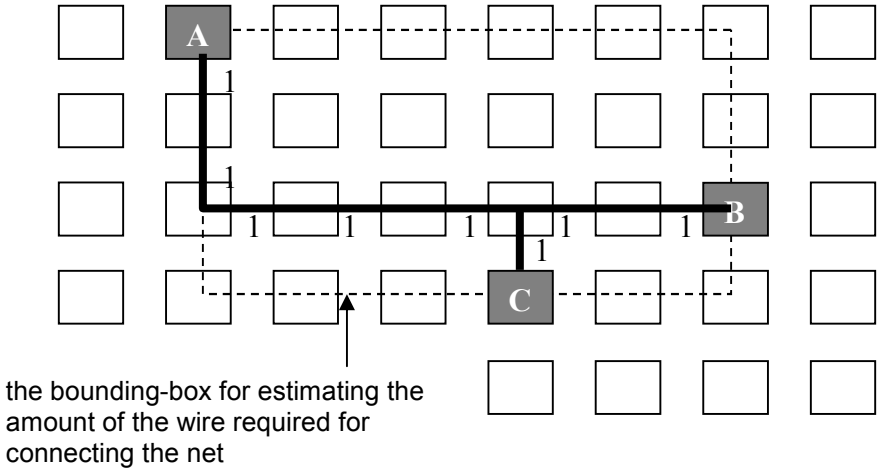
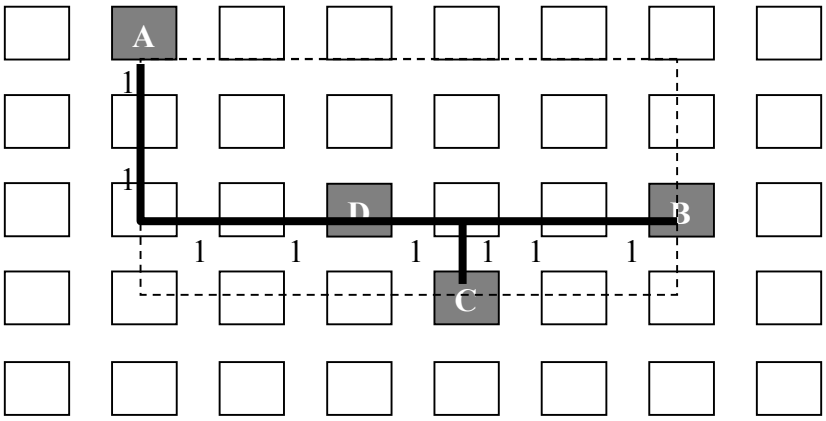
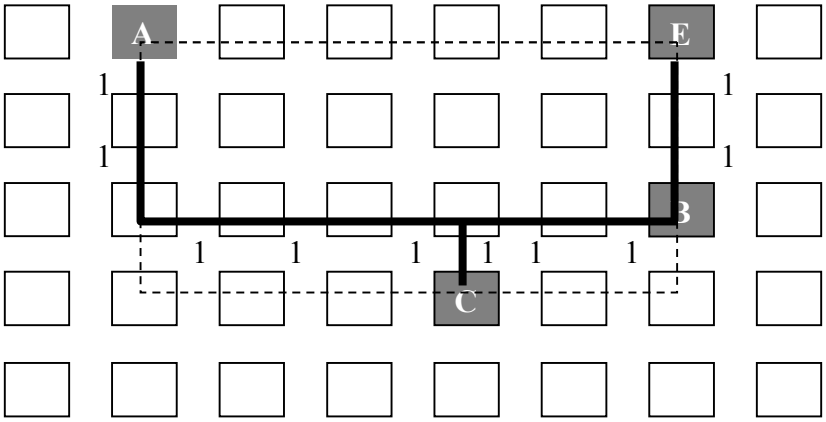


Figure 2.6: HPWL model for 3-block net.



(a) A net connecting 4 blocks with a wirelength of 8



(b) A net connecting 4 blocks with a wirelength of 10

Figure 2.7: Two nets with the same bounding-box size.

The main advantage of HPWL model is that it can be computed very efficiently in *O(1)* time on average [28]. Nevertheless, it has its weaknesses. Figure 2.7 shows a situation where the net in Fig. 2.7(a) and the net in Fig 2.7(b) have the same cardinality and their bounding-boxes have the same perimeter. Therefore, their HPWL estimates will be the same, too. However, the minimal wire-segments needed

to route the net in Fig. 2.7(a) is only 8, while the minimal wire-segments needed to route the net in Fig. 2.7(b) is 10. We can see that the HPWL model totally ignores the relative positions of the blocks inside the bounding-box, although these relative positions inside also affects the number of wire-segments needed to connect the net. Another disadvantage of the HPWL model is that it is not differentiable with respect to the variation of the positions of blocks, and hence it cannot be easily applied to analytic methods.

- **Clique Model and Quadratic Distance**

  As the HPWL model is not suitable for analytic methods due to its non-differentiability, researchers typically estimate wire length using *quadratic* distance. A circuit is modeled as a hypergraph $G_h(V_h, E_h)$ with vertices $V_h = \{v_1, v_2, \ldots v_n\}$ representing cells and hyperedges $E_h = \{e_1, e_2, \ldots e_n\}$ corresponding to signal nets. Vertices are weighted by cell area while hyperedges are weighted according to criticalities or multiplicities [21]. Vertices are either free or fixed. Cell placements in the x and y directions are captured by placement vectors $x=(x_1, x_2, \ldots x_n)$ and $y=(y_1, y_2, \ldots y_n)$.

  Circuit hypergraphs are typically transformed into graphs in which each hyperedge is represented by a set of equally weighted edges. The clique model replaces a net connecting k blocks with a complete graph with k vertices and $k(k-1)/2$ edges. Each vertex represents a block. The edge between vertices A and B is denoted as E_{AB}. Each edge E_{AB} has two properties: a weight W_{AB} and a length L_{AB}. All of the edge weights equal $1/k$[†], and the length L_{AB} equals the shortest wirelength (distance) between A and B. The wire length of a net is estimated as $\sum_{\forall edge \in net} W_{edge} L_{edge}^2$.

  As in the case of analytic placement algorithms, the variations of coordinates in x- and y-dimensions are often independent on each other. Therefore, the distances in x-

[†] Some timing-driven placement algorithms use criticality as the weight of the edge (see Chapter 7) but the computation is similar.

and *y*-dimensions are handled separately. The following example shows the *x*-dimensional (horizontal) quadratic distance of the net in Fig. 2.6. The clique model is shown in Figure 2.8. L_{AB} is the *x*-dimensional distance between *A* and *B*, which equals 5; L_{AC} is the *x*-dimensional distance between *A* and *C*, which equals 3; L_{BC} is the *x*-dimensional distance between *B* and *C*, which equals 2. As this net has 3 terminals, the weight of every edge is $1/k$ (i.e., 1/3). The *x*-dimensional quadratic distance of the net is calculated as follows:

$$\text{quadratic distance}_x = W_{AB}L_{AB}^2 + W_{AC}L_{AC}^2 + W_{BC}L_{BC}^2$$

which in this case equals $\frac{1}{3}\left(5^2 + 3^2 + 2^2\right) \approx 12.667$. The *y*-dimensional quadratic distance can be calculated in a similar way.

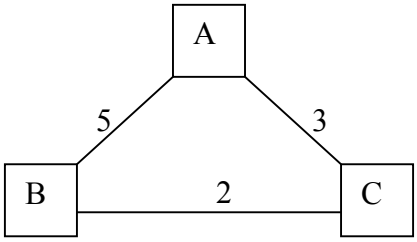


Figure 2.8: The clique model of the net in Figure 2.6 (*x*-dimension)

Clearly, quadratic distance does not directly reflect the wirelength of a net, but it can be applied easily to analytic placement. The total quadratic distance of all the nets can be minimized by solving linear equation systems, to which many mature mathematical methods (like CG and SOR) can be applied. The primary problem with Clique models is that for large hypergraphs they become prohibitively expensive due to the quadratic edge count. Consequently, large edges are either dropped completely, or a combination of clique and *Star models* (discussed below) are employed in which cliques are used to model small hyperedges and stars are used to model large hyperedges.

- **Star Model and Quadratic Distance**

  The star model adds a new vertex at the center of gravity and represents the original net by edges connecting the center to previously existing vertices. The edge between vertex A and the center is denoted as E_A. Each edge E_A has a length L_A equal to the distance between A and the center. The wire length of a net is estimated as $\sum_{\forall edge \in net} L_{edge}^2$.

  Like clique model, the wire length in x- and y-dimensions are handled separately. The following example shows the x-dimensional (horizontal) quadratic distance of the net in Fig. 2.6. The star model is shown in Figure 2.9. The x-coordinate of the center is computed as (1+4+6)/3≈3.67. L_A is the x-dimensional distance between A and the center, which equals 3.67–1=2.67; L_B is the x-dimensional distance between B and the center, which equals 4–3.67=0.33; L_C is the x-dimensional distance between C and the center, which equals 6-3.67=2.33. The x-dimensional quadratic distance of the net is calculated as follows:

$$\text{quadratic distance}_x = L_A^2 + L_B^2 + L_C^2$$

  which in this case equals $2.67^2 + 0.33^2 + 2.33^2 \approx 12.67$. The y-dimensional quadratic distance can be calculated in a similar way.

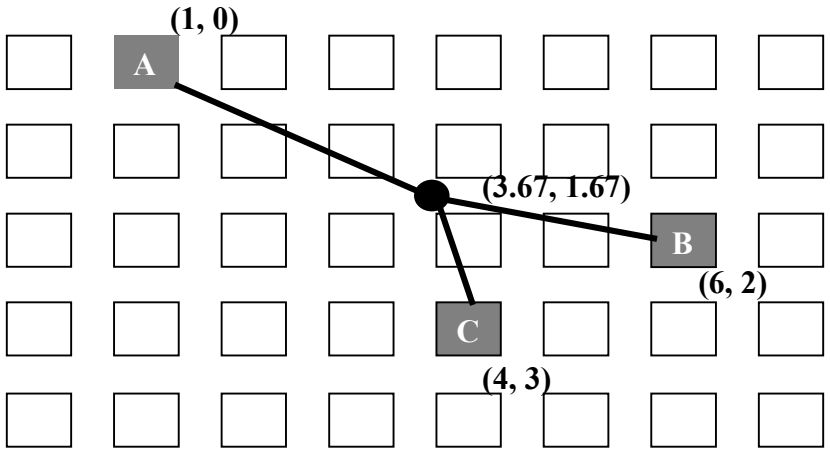


Figure 2.9: The star model of the net in Figure 2.6 (x-dimension)

- **Regularization of Linear Wirelength**

  Pure linear wirelength is rarely used in placement algorithms, as the objective function: $\sum_{i>j} \alpha_{ij} \mid x_i - x_j \mid$ is neither differentiable nor very accurate. (The actual routing length of a net is not simply the sum of the length of all the edges of the hypergraph, but the wirelength of the corresponding Steiner tree.) However, the regularization of linear wirelength is used in some ASIC analytic placers [21][108] [109] due to its differentiability and more accurate estimate compared with quadratic distance. The objective function of regularized linear wirelength is:

$$\Phi(x) = \sum_{i>j} \alpha_{ij} \sqrt{(x_i - x_j)^2 + \beta} : Hx = b$$

  where x_i and x_j are the x-coordinates of block i and block j; H represents various linear constraints. When $\beta \to 0$, $\Phi(x) \to \sum_{i>j} \alpha_{ij} \mid x_i - x_j \mid$. Since minimizing regularized linear wirelength is more difficult than minimizing quadratic distance (minimizing quadratic distance results in solving a linear equation system while minimizing regularized linear wirelength results in solving a non-linear system), regularization of linear wirelength is not as popular (in analytic methods) as quadratic distance.

2.5 Summary

In summary, existing wirelength models may be too expensive to be practically used in placement algorithms (e.g., minimum Steiner tree and minimum spanning tree model), too inaccurate to produce high-quality results (e.g. quadratic distance), or not differentiable so that they cannot be applied easily in analytic methods (e.g., HPWL model). In the next Chapter, we present a novel wirelength estimate, called the *Star+* model. Unlike the HPWL model, the Star+ model is differentiable, and hence suitable for analytic methods. Another feature of the Star+ model is that the computation of ΔC

caused by the swap of two blocks always runs in *O(1)* time. This also makes it suitable for use with SA-based methods, where millions of swaps may need to be evaluated efficiently. Finally, the Star+ model is also very accurate. Our results show that when the Star+ model replaces the HPWL model in VPlace, the Star+ model produces results that are as good (and in some cases better) than those produced by HPWL. As well, the actual runtime for Star+ is as fast (and in many cases slightly faster) as that of HPWL.

Based on the Star+ model, we introduce two novel analytic placement algorithms (Chapter 4 and Chapter 6). These algorithms differ from other analytic placers in that we do not employ quadratic distance, but a more accurate distance measure. This allows us to produce accurate results quickly.

2.6 Benchmarks

All 20 MCNC benchmarks, shown in Table 2.2, are used to measure the performance of all of the analytic methods developed in this thesis. We have chosen to use these benchmarks because most researchers use them to validate the experimental results. The suite consists of circuits ranging from a few hundred CLBs to nearly ten thousand CLBs.

Table 2.2: 20 MCNC benchmarks

Circuit	FPGA matrix	Number of CLBs	Number of Nets	Number of I/O pads	Maximum Fanout	Average Fanout
tseng	33x33	1047	1099	174	389	4.77
ex5p	33x33	1064	1072	71	324	4.73
apex4	36x36	1262	1271	28	208	4.52
misex3	38x38	1397	1411	28	186	4.52
diffeq	39x39	1497	1561	103	497	4.63
alu4	40x40	1522	1536	22	250	4.52
seq	42x42	1750	1791	76	234	4.46
apex2	44x44	1878	1916	41	148	4.49
s298	44x44	1931	1935	10	397	4.6
dsip	54x54	1370	1599	426	908	4.67
bigkey	54x54	1707	1936	426	461	4.38
frisc	60x60	3556	3576	136	887	4.82
elliptic	61x61	3604	3735	245	1471	4.68
spla	61x61	3690	3706	62	215	4.73
des	63x63	1591	1847	501	227	4.31
ex1010	68x68	4598	4608	20	303	4.49
pdc	68x68	4575	4591	56	261	4.74
s38417	81x81	6406	6435	135	1464	4.54
S38584.1	81x81	6447	6485	342	2742	4.41
clma	92x92	8383	8445	144	1170	4.61

Chapter 3

The Star+ Model

As the precise wirelength for a given placement can only be known *after* routing, accurate and fast *approximation* models for calculating the amount of wire required to connect all of the nets are needed for placement algorithms. The accuracy of these models directly affects the quality of the placements obtained when using these models. An inaccurate model will certainly degrade the quality of results. Moreover, a computationally expensive model will increase the running time of the algorithms that use these models.

As discussed in Section 2.4, the Half-Perimeter Wire-Length model [28] is the most commonly used wire-estimation model, and is used today in commercial placement tools including Altera's Quartus CAD tools. HPWL is both accurate and fast to compute. However, it cannot be used directly in analytic placement tools due to its non-differentiability. In this Chapter, we present a novel wire-length estimation model, called *Star+*, which is both differentiable and suitable for use in analytic placement. We use an empirical method to evaluate the Star+ model and compare it to the HPWL model used by VPR [28]. The comparison of the two wire-length estimation models involves using the VPR framework to place and route benchmark circuits into realistic FPGA

architectures, first using HPWL then using Star+. The delay and wiring requirements of each circuit implementation are computed using sophisticated models (that are already part of VPR), and from these results we are able to compare Star+ and HPWL. The objective of our experiments is not to show that Star+ is superior to HPWL, but to show that Star+ is *comparable* to HPWL, both with respect to runtime and quality of results produced. A direct comparison of both models to determine which, if either, is superior, would require testing both models on thousands of problem instances, and with hundreds or even thousands of different placement and routing parameters – something that is unnecessary for the work proposed here. Consequently, we deliberately limit the scope and extent of the experiments that follow.

The remainder of this Chapter is organized as follows: In Section 3.1 we introduce the Star+ wire-estimation model. In Section 3.2, we show that the Star+ wire estimate can be computed in $O(1)$ time. Then, in Section 3.3, we perform a series of experiments comparing Star+ head-to-head with HPWL. As the Star+ model contains parameters that affect its performance, in Section 3.4 we explain how appropriate values for these parameters were determined. In Section 3.5 we explain some limitations of Star+. Finally, we provide a summary of the results in Section 3.6.

3.1 Wire-estimation based on the Star+ model

Like the quadratic distance measure employed in quadratic placement tools [17][18], the Star+ model handles the *x*-dimension and the *y*-dimension separately. Therefore, for the sake of simplicity, only the *x*-dimension is discussed here. (The *y*-dimension is processed similarly.)

An important concept in the Star+ model is the *center-of-gravity* of a net. We define the center of gravity of a net as follows. First, for Net_l, we use k_l to represent the cardinality of the net (i.e., the number of blocks connected to Net_l). Let c_l represent the

center of gravity of net l; we define the x-coordinate of c_l as: $x_{cl} = \frac{1}{k_l} \sum_{\forall i \in Net_l} x_i$. The expression $i \in Net_l$ simply means that $block_i$ connects to Net_l, and x_i represents the x-coordinate of $block_i$. The Star+ model is effectively the 2-norm† of all of the distances from $block_i$ to the center of gravity of Net_l:

$$\left\| Net_l \right\|_x = \alpha \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} \qquad (Equation\ 3.1)$$

The factor α, like the factor $q(i_k)$ used in the HPWL model, is used to compensate for underestimation of the wire-length. The parameter β is a positive number, which is used to make Equation 3.1 always differentiable. Equation 3.1 has two important features: First, it is differentiable when β is greater than zero, and hence it is suitable for use with analytic methods. Second, the incremental change in cost caused by swapping two blocks (or moving a single block) can always be calculated in constant time. This feature is very important as it allows the Star+ model to also be used in move-based methods, like *VPlace* [28], where the time to calculate the incremental changes of the cost of a net (after performing a candidate swap) directly affects the performance of the algorithm.

The graphical representation of the Star+ model is based on a star model [18][83]. Figure 3.1 shows a star model of a net with 4 blocks: *A*, *B*, *C*, and *D*. Each block has a connection to the center of gravity of the net (represented by the bold dot). The length of each edge $(x_i - x_{cl})^2$ is the quadratic distance from $block_i$ to the center of gravity x_{cl}. However, the Star+ model is different from the traditional quadratic distance used in analytic placement. Analytic placers, using traditional quadratic distance, minimize the sum of all of the *squared* distances between any pair of blocks that connect to a common

† In linear algebra and related areas of mathematics, a norm is a function that assigns a positive length or size to all vectors in a vector space. The most commonly used norm is Star+, which is a special case of p-norm ($p \geq 1$). The p-norm of a vector $X=(x_1, x_2, \ldots, x_n)$ is presented as: $\left\| X \right\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$.

net. The Star+ model, on the other hand, minimizes the sum of the *square roots* of the sum of the quadratic distances between each block and the center of gravity of a net (Equation 3.1). Due to the linear nature of distance, the Star+ model can, theoretically, be more accurate than quadratic distance if the parameters α and β are assigned appropriate values. The accuracy of the Star+ model, however, comes at the expense of the complexity of the (analytic) algorithm that employs it. To minimize quadratic distance, only a *linear* equation system must be solved. To minimize the Star+ model a *non-linear* equation system must be solved, which is usually much harder and hence more time-consuming. (In Chapter 4 we introduce a special mechanism for reducing the time required to solve the non-linear equation systems based on conjugate-gradient [32].)

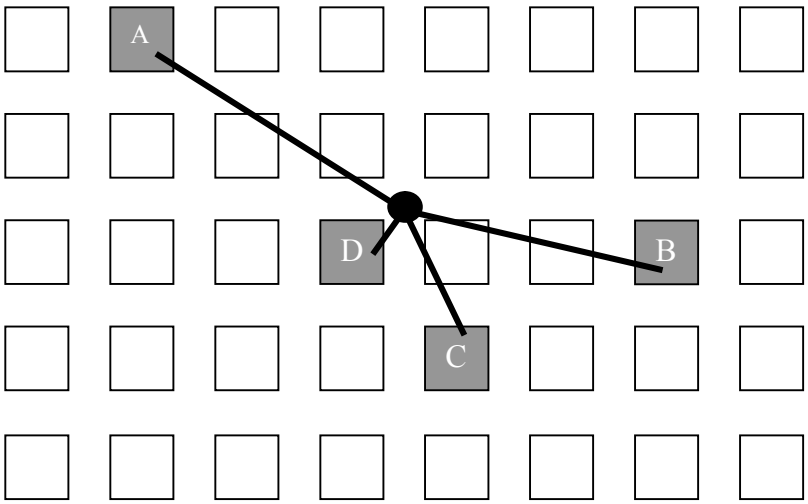


Figure 3.1: A star model of a 4-pin net

3.2 Constant-time update of cost

In its current form, Equation 3.1 is not suitable for estimating wire length, as it is too expensive to calculate each time a block moves position. Therefore, it must be transformed into a new form that allows *incremental* changes in cost (resulting from block movement) to be calculated in constant time. This can be done as follows:

$$
\begin{aligned}
\left\|Net_l\right\|_x &= \alpha \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} \\
&= \alpha \sqrt{\sum_{\forall i \in Net_l} ({x_i}^2 - 2x_i x_{cl} + {x_{cl}}^2) + \beta} \\
&= \alpha \sqrt{\sum_{\forall i \in Net_l} {x_i}^2 - 2 \sum_{\forall i \in Net_l} (x_i x_{cl}) + \sum_{\forall i \in Net_l} {x_{cl}}^2 + \beta} \\
&= \alpha \sqrt{\sum_{\forall i \in Net_l} {x_i}^2 - 2x_{cl} \cdot \sum_{\forall i \in Net_l} x_i + \sum_{\forall i \in Net_l} {x_{cl}}^2 + \beta} \\
&= \alpha \sqrt{\sum_{\forall i \in Net_l} {x_i}^2 - 2x_{cl} \cdot k_l x_{cl} + k_l {x_{cl}}^2 + \beta} \\
&= \alpha \sqrt{\sum_{\forall i \in Net_l} {x_i}^2 - k_l {x_{cl}}^2 + \beta}
\end{aligned}
$$

Let $U_l = \sum_{\forall i \in Net_l} {x_i}^2$ and $V_l = \sum_{\forall i \in Net_l} x_i = k_l x_{cl}$, then we have:

$$
\begin{aligned}
\left\|Net_l\right\|_x &= \alpha \sqrt{\sum_{\forall i \in Net_l} {x_i}^2 - k_l {x_{cl}}^2 + \beta} \\
&= \alpha \sqrt{\sum_{\forall i \in Net_l} {x_i}^2 - \frac{(k_l x_{cl})^2}{k_l} + \beta} \\
&= \alpha \sqrt{U_l - \frac{V_l^2}{k_l} + \beta} \qquad \text{(\textit{Equation 3.2})}
\end{aligned}
$$

Now suppose b is a block connected to Net_l (i.e., $b \in Net_l$) and that block b moves from position x_b to the new position x_b^{new}. This movement causes U_l, V_l and $\left\|Net_l\right\|_x$ to change values. The new value of U_l^{new} and V_l^{new} can be calculated, respectively, in constant time using following formulas:

$$U_l^{new} = U_l - x_b^2 + (x_b^{new})^2 \qquad \text{and} \qquad V_l^{new} = V_l - x_b + x_b^{new}. \qquad \text{(\textit{Equation 3.3})}$$

As a result, the new value of $\left\|Net_l\right\|_x$ can also be calculated in a *constant* time

using Equation 3.2.

To incorporate the Star+ model into VPlace [28] – VPR's placement tool – two variables representing T_l and S_l for each net must be introduced, and all of the functions that involve calculation of the cost of the nets must be changed to use the new Star+ model. In practice, the parameters α and β in Equation 3.2 are set to 1.59 and 1, respectively. (The previous values of 1 and 1.59 were determined empirically. A discussion of how these values were obtained is postponed to Section 3.4. However, we note that our experimentation reveals that the value of α *does not* have a significant effect on the quality of placement, while the value of β *does* have a significant effect on both the quality of the placement and its routability.)

3.3 Star+ Model Evaluation

The objective of our experiments is to show that the Star+ model is *accurate* enough to produce high-quality placements, and *fast* enough that it can be used with both analytic methods [18][19][25] as well as move-based methods, like those in [28][29][50][54]. As VPR [28] is the current public domain state-of-the-art place-and-route tool, we use it as a basis of comparison. Our methodology is as follows. First, we run VPR on all 20 MCNC [62] benchmarks using VPR's original placement algorithm, VPlace, which uses the HPWL wire-estimation model. Each placed circuit is then routed. Following routing, information about the minimum number of required *channels*, *critical path*, and *total wire length* is obtained for each circuit. The HPWL model in VPlace is then replaced with the Star+ model, and the process repeated. Finally, the routing results from both models are used to compare the speed and effectiveness of the two models head-to-head.

For the purpose of this experiment, we downloaded the VPR 4.3 source code, architecture file, and the complete set of 20 MCNC benchmark circuits used by VPR from [28]. We used the default architecture file as is, which assumes that each CLB contains 4 LUTs, and each LUT has 4 inputs and is paired with one flip-flop. We first ran

the benchmarks through the entire VPR flow having first configured VPlace to use the HPWL estimation model, and to report the minimum channel width when using its *breadth-first* strategy to route the components. We then repeated the experiment, but this time configured VPR to route the components using its *timing-driven* algorithm. We then replaced the HPWL model in VPlace with the Star+ model, and repeated the previous experiments (using both breadth-first and timing-driven timing options).

For each benchmark, each model, and each routing algorithm, VPlace was executed with the option `inner_num` equal to 1 and 10, respectively. VPlace uses `inner_num` to trade-off quality for speed. In particular, the number of moves attempted at each temperature equals `inner_num` x (the number of blocks)$^{4/3}$. The default value of `inner_num` is 10. Specifying an `inner_num` of 1 will speed up VPlace by a factor of 10, but will typically reduce placement quality by about 10 percent. Setting `inner_num` greater than 10 barely improves the quality of the placement, but does increase the running time of VPlace.

As discussed in Section 2.3, VPlace is based on simulated annealing which is stochastic. Therefore, running VPlace more than once with different seed values results in different placements being produced. In order to make the experiments more accurate, for each set of parameters, VPR is executed ten times, each time using a different (randomly generated) seed value.

3.3.1 Routability

We begin by considering how effective the Star+ model is in producing placements that are *routable*. As a measure of the quality of a routable placement, we measure the *minimum channel width* (or number of tracks per channel) that VPR's router needs to successfully route each circuit placement. Channel width is one of the most commonly used criterions for assessing the routability of a potential placement. The results of the previous experiments are given in Table 3.1, Table 3.2, Table 3.3, and Table 3.4,

respectively.

Table 3.1: Channel Width and Routing (`breadth_first` and `inner_num`=1)

	HPWL						Star+					
	CW	**SR**	**CW**	**SR**	**CW**	**SR**	**CW**	**SR**	**CW**	**SR**	**CW**	**SR**
alu4	9	0	10	4	11	10	9	0	10	8	11	10
apex2	11	1	12	8	13	10	11	1	12	10	13	10
apex4	12	1	13	10	14	10	12	0	13	9	14	10
Bigkey	5	0	6	1	7	10	5	0	6	9	7	10
Clma	12	3	13	10	14	10	12	4	13	9	14	10
Des	6	0	7	4	8	10	6	0	7	3	8	10
Diffeq	7	0	8	10	9	10	7	0	8	8	9	10
Dsip	5	0	6	3	7	7	5	0	6	5	7	9
Elliptic	11	0	12	9	13	10	11	3	12	10	13	10
ex1010	10	0	11	10	12	10	10	1	11	7	12	10
ex5p	13	2	14	9	15	9	13	0	14	10	15	10
Frisc	12	0	13	8	14	8	12	0	13	9	14	10
Misex3	10	0	11	8	12	10	10	0	11	4	12	10
Pdc	16	3	17	10	18	10	16	1	17	8	18	10
s298	7	0	8	9	9	10	7	1	8	9	9	10
s38417	7	0	8	8	9	9	7	0	8	4	9	9
s38584.1	8	0	9	8	10	10	8	0	9	10	10	10
Seq	11	2	12	10	13	10	11	1	12	10	13	10
Spla	13	0	14	10	15	10	13	0	14	4	15	10
Tseng	6	0	7	3	8	9	6	0	7	8	8	10

Note that when running VPR's router, the designer must also specify the minimum channel width (CW) that is to be used. As different placements may require different channel widths in order to be routed, several different channel widths are tried as indicated in table columns 2, 4, 6, 8, 10, and 12, respectively. The actual number of placements that were successfully routed (SR) for each channel width is reported in columns, 3, 5, 7, 9, 11, and 13, respectively. (Note: if five (or more) of the ten placements are found to be routable for a given channel width, we consider this a "success". We then shade the smallest successful CW entries in each row of the table.) For example, consider *Alu4* in Table 3.4. For both the HPWL and Star+ model, when the channel width is 9, none of the 10 placed circuits was found to be routable. However, when the channel width was increased to 10, 4 of the 10 placements using HPWL were successfully routed,

while 8 of the 10 placements found using Star+ were successfully routed. Finally, when the channel width was increased to 11, all 10 placements, for both HPWL and Star+, were found to be routable. Therefore, in this case, shading is used to indicate CW=10 and CW=11 as the smallest "successful" channel widths found by Star+ and HPWL, respectively.

Table 3.2: Channel Width and Routing (`breadth_first` and `inner_num`=10)

	HPWL						Star+					
	CW	**SR**	**CW**	**SR**	**CW**	**SR**	**CW**	**SR**	**CW**	**SR**	**CW**	**SR**
alu4	9	0	10	9	11	10	9	0	10	10	11	10
apex2	10	0	11	6	12	10	10	0	11	5	12	10
apex4	11	0	12	6	13	10	11	0	12	2	13	10
Bigkey	5	0	6	1	7	10	5	0	6	4	7	10
Clma	11	1	12	10	13	10	11	0	12	7	13	10
Des	6	0	7	2	8	10	6	0	7	8	8	10
Diffeq	6	0	7	5	8	10	6	0	7	2	8	10
Dsip	5	0	6	5	7	10	5	0	6	9	7	10
Elliptic	9	0	10	6	11	10	9	0	10	1	11	8
ex1010	9	0	10	4	11	10	9	0	10	6	11	10
ex5p	12	0	13	7	14	10	12	0	13	4	14	10
Frisc	11	0	12	6	13	10	11	0	12	0	13	9
Misex3	10	0	11	10	12	10	10	0	11	7	12	10
Pdc	15	0	16	7	17	10	15	0	16	3	17	10
S298	6	0	7	7	8	10	6	0	7	5	8	10
S38417	6	0	7	0	8	10	6	0	7	0	8	9
S38584.1	7	0	8	8	9	10	7	0	8	6	9	10
Seq	10	0	11	9	12	10	10	0	11	1	12	10
Spla	12	0	13	8	14	10	12	0	13	1	14	6
Tseng	6	0	7	10	8	10	6	0	7	10	8	10

Table 3.3: Channel Width and Routing (`timing_driven` and `inner_num`=1)

	HPWL						Star+					
	CW	SR	CW	SR	CW	SR	CW	SR	CW	SR	CW	SR
alu4	9	0	10	1	11	7	9	0	10	0	11	9
Apex2	11	0	12	4	13	10	11	0	12	7	13	10
Apex4	12	0	13	3	14	8	12	0	13	2	14	10
bigkey	6	3	7	9	8	10	6	5	7	10	8	10
Clma	12	0	13	5	14	10	12	0	13	4	14	8
Des	6	0	7	3	8	10	6	0	7	1	8	9
Diffeq	7	0	8	2	9	10	7	0	8	7	9	10
Dsip	5	0	6	0	7	8	5	0	6	5	7	9
elliptic	11	0	12	1	13	8	11	2	12	4	13	10
ex1010	11	4	12	10	13	10	11	3	12	9	13	10
ex5p	13	0	14	5	15	10	13	0	14	0	15	9
Frisc	12	0	13	3	14	9	12	0	13	3	14	8
misex3	11	2	12	9	13	10	11	0	12	8	13	10
Pdc	17	2	18	8	19	10	17	0	18	8	19	10
s298	7	0	8	5	9	10	7	2	8	8	9	10
s38417	7	0	8	5	9	10	7	0	8	5	9	10
s38584.1	8	1	9	5	10	10	8	1	9	10	10	10
Seq	11	0	12	3	13	9	11	0	12	6	13	9
Spla	14	1	15	8	16	10	14	0	15	5	16	8
Tseng	6	0	7	0	8	10	6	0	7	2	8	10

Table 3.5 summarizes the previous experimental results. The entries in the table indicate the smallest "successful" routable channel width. The last row of the table shows the aggregate of the smallest "successful" routable channel widths for each situation. Overall, it can be seen that both HPWL and Star+ performed similarly across the entire set of benchmarks and for all situations, with Star+ slightly outperforming HPWL when `inner_num`=1 (215 versus 216 using breadth first and 223 versus 226 using timing-driven). A close look at Tables 3.1 – 3.4 reveals that the Star+ model finds a smaller minimum "successful" routable channel width in 18% of the test cases, while the HPWL model finds a smaller minimum "successful" routable channel width in 20% of the test cases.

Table 3.4: Channel Width and Routing (`timing_driven` and `inner_num`=10)

	HPWL						Star+					
	CW	SR	CW	SR	CW	SR	CW	SR	CW	SR	CW	SR
Alu4	9	0	10	3	11	9	9	0	10	7	11	10
apex2	10	0	11	4	12	10	10	0	11	4	12	10
apex4	12	0	13	6	14	10	12	0	13	4	14	10
bigkey	6	3	7	9	8	10	6	5	7	10	8	10
clma	11	0	12	6	13	10	11	0	12	3	13	10
Des	6	0	7	3	8	9	6	0	7	4	8	10
diffeq	6	0	7	4	8	10	6	0	7	3	8	10
dsip	5	0	6	3	7	10	5	0	6	5	7	10
elliptic	10	3	11	7	12	10	10	2	11	7	12	10
ex1010	10	0	11	6	12	10	10	0	11	7	12	10
ex5p	13	2	14	9	15	10	13	0	14	6	15	10
frisc	12	0	13	6	14	10	12	0	13	6	14	10
misex3	10	0	11	8	12	10	10	0	11	4	12	10
Pdc	16	0	17	3	18	10	16	0	17	2	18	9
s298	6	0	7	2	8	9	6	0	7	3	8	10
s38417	6	0	7	3	8	10	6	0	7	3	8	10
s38584.1	8	2	9	6	10	10	8	1	9	10	10	10
Seq	10	0	11	4	12	10	10	0	11	4	12	9
spla	13	0	14	2	15	10	13	0	14	2	15	10
tseng	6	1	7	8	8	10	6	1	7	8	8	10

3.3.2 Critical Path Delay

We now turn our attention to critical-path delay. Table 3.6 reports the critical-path delay computed by VPR's router when performing the experiments described in Section 3.3.1. Column 1 identifies the benchmark. Column 2 gives the channel width used by the router (Note: When providing a minimum channel width to VPR's router for each benchmark, we used the minimum successful channel width (from Table 3.3 and Table 3.4) plus 1. (By using a channel width one more than the minimum, the probability of the router finding a feasible route is effectively 1 in all cases.) Column 3 is the number of times of successful routing out of 10. Column 4 gives the average[†] critical-path delay for HPWL

[†] VPR is run 10 times on each benchmark as per the experiment in Section 3.3.1

when VPlace is run with `inner_num` equal to 1. Columns 5 and 6 provide similar information for the Star+ model. Column 7 shows the p-value (see the next section, Statistical Testing for explanation). Columns 8 – 12 give similar information as Columns 3 – 7, but for `inner_num` equal to 10. Note that the results presented in Table 3.6 were obtained running VPR's router with the *timing-driven* option invoked. (Results were also obtained using the *breadth-first* timing option; however, these results in all cases were consistent with, but inferior to, those found using the timing-driven option. As such, they are not reported.)

Table 3.5: Summary of Minimum Routable Channel Widths

	Breadth_first				**Timing_driven**			
	inner_num 1		**inner_num 10**		**inner_num 1**		**inner_num 10**	
	HPWL	**Star+**	**HPWL**	**Star+**	**HPWL**	**Star+**	**HPWL**	**Star+**
alu4	11	10	10	10	11	11	11	10
apex2	12	12	11	11	13	12	12	12
apex4	13	13	12	13	14	14	13	14
Bigkey	7	6	7	7	7	6	7	6
Clma	13	13	12	12	13	14	12	13
Des	8	8	8	7	8	8	8	8
Diffeq	8	8	7	8	9	8	8	8
Dsip	7	6	6	6	7	6	7	6
Elliptic	12	12	10	11	13	13	11	11
ex1010	11	11	11	10	12	12	11	11
ex5p	14	14	13	14	14	15	14	14
Frisc	13	13	12	13	14	14	13	13
Misex3	11	12	11	11	12	12	11	12
Pdc	17	17	16	17	18	18	18	18
S298	8	8	7	7	8	8	8	8
S38417	8	9	8	8	8	8	8	8
S38584.1	9	9	8	8	9	9	9	9
Seq	12	12	11	12	13	12	12	12
Spla	14	15	13	14	15	15	15	15
Tseng	8	7	7	7	8	8	7	7
Total	**216**	**215**	**200**	**206**	**226**	**223**	**215**	**215**

Table 3.6: Critical Path Delay (unit: ns)

	CW	Inner_num 1					Inner_num 10				
		HPWL		Star+		P-value	HPWL		Star+		P-value
		ST	CPD	ST	CPD		ST	CPD	ST	CPD	
Alu4	11	7	120.331	9	106.477	0.0006	9	113.6717	10	104.823	0.0082
Apex2	13	10	128.77	10	109.399	1E-07	10	125.1346	10	108.013	1E-06
Apex4	14	8	127.922	10	117.623	0.001	10	122.6053	10	104.512	2E-06
Bigkey	8	10	100.935	10	79.6366	4E-06	10	100.0536	10	75.9327	6E-10
Clma	14	10	264.999	8	248.781	0.0625	10	252.9958	10	254.389	0.5168
Des	8	10	123.01	9	142.761	0.0119	9	136.5118	10	142.17	0.4948
Diffeq	9	10	106.112	10	97.1525	0.0711	10	90.33062	10	88.3868	0.5933
Dsip	7	8	91.0482	9	75.7863	0.0034	10	93.37907	10	76.9176	6E-05
Elliptic	13	8	257.387	10	224.639	0.0033	10	206.6148	10	210.407	0.7445
ex1010	13	10	205.552	10	186.939	0.0182	10	202.9452	10	192.969	0.2136
ex5p	15	10	116.071	9	109.359	0.1208	10	125.2613	10	102.643	1E-06
Frisc	14	9	227.362	8	210.73	0.2222	10	189.0848	10	193.943	0.43
Misex3	13	10	108.431	10	103.505	0.1367	10	105.6976	10	104.492	0.6061
Pdc	19	10	254.422	10	234.849	0.0051	10	217.5874	10	219.941	0.7644
s298	9	10	240.983	10	215.34	0.0007	10	203.189	10	202.723	0.9335
s38417	9	10	196.969	10	159.805	9E-05	10	163.1709	10	131.942	3E-07
s38584.1	10	10	123.888	10	117.656	0.1231	10	119.709	10	110.079	0.0239
Seq	13	9	123.035	9	111.032	0.0246	10	118.0495	10	110.467	0.0663
Spla	16	10	205.085	7	185.875	0.0023	10	188.0682	10	173.829	0.0318
Tseng	8	10	81.7572	10	82.3753	0.8034	10	75.83124	10	75.2984	0.6969
Total		**189**	**3204.07**	**188**	**2919.72**		**198**	**2949.891**	**200**	**2783.88**	

Our results show that when VPlace is run with `inner_num` equal to 1, Star+ results in a lower critical-path delay compared with HPWL for 18 of the 20 benchmarks. Moreover, the placements found using Star+ are, on average, 9 percent faster than those found when using HPWL. Similarly, when VPlace is run with `inner_num` equal to 10, Star+ results in a lower critical-path delay compared with HPWL for 15 of the 20 benchmarks. Moreover, the placements found using Star+ are, on average, 6 percent faster than those found using HPWL.

3.3.2.1 Statistical Testing

The previous results were somewhat unexpected. To verify whether or not the results of the previous experiments were due to variance, we employed the Student's t-test [84].

The Student's t-test is a statistical tool typically used to make inferences from samples that are relatively small in size. The motivation for using Student's t-test for a small sample size is that the sample's mean and standard deviation may not reflect the true mean and standard deviation of the entire population that was sampled. Student's t-test provides a probability of confidence, or *P*-value, for the null hypothesis that the means of two populations are equal, given two sets of sample data - one taken from each population. It is common for a significance level of 0.05 to be chosen as a fixed probability of wrongly rejecting the null hypothesis, if it is true. The null hypothesis is rejected at the 5% significance level for P-values that are less than 0.05. A caveat of Student's t-test is that the sample sets are assumed to be *normally* distributed. For cases where the sample set data is not normal, a non-parametric, *ranked t-test* can be used to relax the normality assumption. The ranked t-test assigns each sampled data instance with a rank that is used instead of the actual data's value for the purposes of calculating the t-test confidence probability. The ranks are assigned by combining the sample sets together and sorting them in descending order based on their value, after which ranks are assigned in increasing order, starting at one and increasing by one for each rank. Once these ranks have been assigned, the rank values are separated based on which sample set they were originally from and the t-test is performed on these ranks instead of on the original data values.

For the previous experiments summarized in Table 3.6, the Student's t-test can provide insight into whether or not the performance of HPWL and Star+ with respect to producing routable placements is attributable to chance. Using the data in Table 3.6, the Student's t-test was used to test the null hypothesis that the mean value of the results found using HPWL and Star+, respectively, are equal, implying that the differences in results between the two models are due to "chance". The alternative hypothesis states that HPWL and Star+ are unique models that provide different results, not attributable to "chance". The corresponding t-test P-value scores for the previous four experiments are reported in Table 3.7 (columns 4 and 7). The null hypothesis is rejected in favour of the alternative hypothesis in cases where the P-value for a set of results is less than 0.05. This corresponds to a significance level of 5%. For cases where the P-value is greater than

0.05, the null hypothesis is not rejected.

Table 3.7: Results of Student T-test

	inner_num 1			Inner_num 10		
	HPWL	**Star+**	**P-value**	**HPWL**	**Star+**	**P-value**
Alu4	120.3309	106.4766	0.000564	113.6717	104.8231	0.00817
Apex2	128.7704	109.3992	1.4E-07	125.1346	108.0127	1.35E-06
Apex4	127.9216	117.6226	0.001031	122.6053	104.5116	1.9E-06
Bigkey	100.9347	79.63662	4.07E-06	100.0536	75.93271	5.51E-10
Clma	264.9987	248.7808	0.062485	252.9958	254.3889	0.516848
Des	123.0099	142.7608	0.011871	136.5118	142.1703	0.494806
Diffeq	106.112	97.15254	0.071095	90.33062	88.38679	0.593258
Dsip	91.04823	75.78627	0.003382	93.37907	76.91763	6.44E-05
Elliptic	257.3873	224.6386	0.00333	206.6148	210.4074	0.744468
ex1010	205.5515	186.9388	0.018189	202.9452	192.9694	0.213586
ex5p	116.0711	109.3591	0.120808	125.2613	102.6426	1.47E-06
Frisc	227.3616	210.7299	0.222162	189.0848	193.9425	0.430033
Misex3	108.4306	103.5046	0.136724	105.6976	104.4922	0.60606
Pdc	254.4219	234.8487	0.005147	217.5874	219.941	0.7644
S298	240.9833	215.3401	0.000691	203.189	202.7228	0.933478
S38417	196.9685	159.805	9.31E-05	163.1709	131.9415	2.61E-07
S38584.1	123.8882	117.6562	0.123138	119.709	110.0792	0.023875
Seq	123.0353	111.0318	0.024585	118.0495	110.467	0.066259
Spla	205.0847	185.875	0.002318	188.0682	173.8293	0.031827
Tseng	81.75721	82.37529	0.803405	75.83124	75.29837	0.696864
Total	**3204.068**	**2919.718**		**2949.891**	**2783.877**	

The P-values in Table 3.7 reveal that for `inner_num =1` the null hypothesis is rejected for 13 of the 20 cases; indicating that in 65% of the test cases, the difference between the Star+ and HPWL models with respect to critical path delay is statistically significant. Moreover, Table 3.7 reveals that for `inner_num = 10` the null hypothesis is rejected for 9 of the 20 cases. Thus, from this we can conclude that variance can account for the difference in wirelength estimates in approximately 40% of the test cases. For the remaining 60% of the test cases, the Star+ and HPWL models were found to be different, with Star+ outperforming HPWL with respect to critical-path delay. However, again we would emphasize that our goal here is not to present Star+ as a superior wirelength estimation model to HPWL, but comparable, which these results suggest.

3.3.2.2 Insight into performance of Star+ versus HPWL

A possible insight into why Star+ outperforms the HPWL model in some cases with respect to critical-path delay is now given. Observe that when the HPWL model is used to perform placement, only the positions of the blocks on the four sides of the bounding box are taken into account. If the two ends of the path with the longest delay happen to be on the sides of the bounding box, minimizing the bounding box will move these ends inside the current box, and hence reduce the longest path delay. Otherwise, minimizing the bounding box only decreases the wire-length estimate and does nothing to shorten the path(s) with the longest delay(s). Generally, the more blocks a net has, the smaller the chance that the two ends of the longest path will lie on the sides of the bounding box. Conversely, if the Star+ model is used in lieu of the HPWL model, we can see from Equation 3.1 that the positions of *all* blocks in the net are considered. Minimizing the Star+ distance pulls all the blocks towards the center-of-gravity of the net. Even if the two ends of the longest path are not on the sides of the bounding box (imagine there still is a bounding box surrounding all of the blocks in the net), they are still pulled towards the center-of-gravity, directly reducing the wire-length between them. Furthermore, if the two ends are farther from the center-of-gravity (i.e., closer to the sides of the bounding box), they will be pulled "harder" towards the center due to the square effect in Equation 3.1.

Figure 3.2 shows the placement of a net (clma: 661) obtained by VPR using the HPWL model. Figure 3.3 shows the placement of the same net by VPR using the Star+ model. In both plots, each dot represents a block, and the rectangle surrounding the dots is the bounding box. All together, this net connects 31 blocks. It can be seen that there are more blocks on or near the sides of the bounding box in Fig. 3.2 than in Fig. 3.3. This is because HPWL is more likely to move blocks near the sides of the box, while the Star+ model is more likely to move blocks close to the center-of-gravity. Although one cannot definitely state the net in Fig. 3.2 will have a longer critical path than the net in Fig. 3.3 until routing is performed, intuitively, we believe that this may be the case for the reasons mentioned.

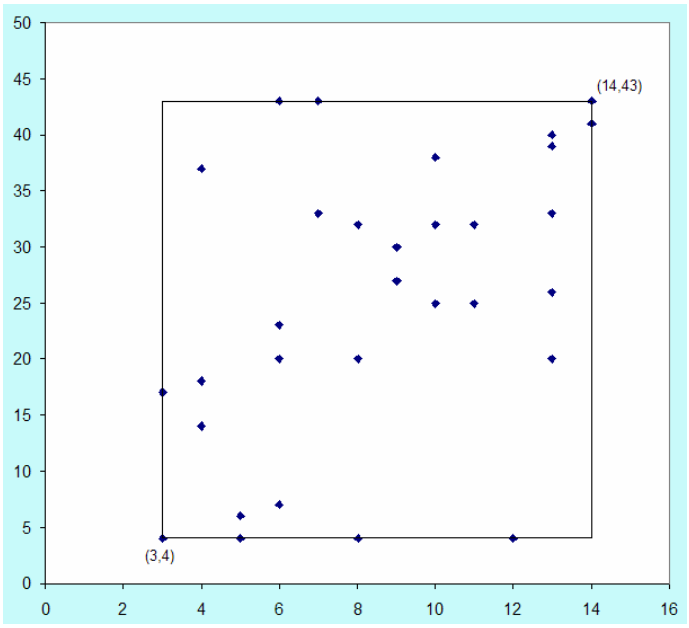


Figure 3.2: The placement of Net clma:661 obtained using bounding box

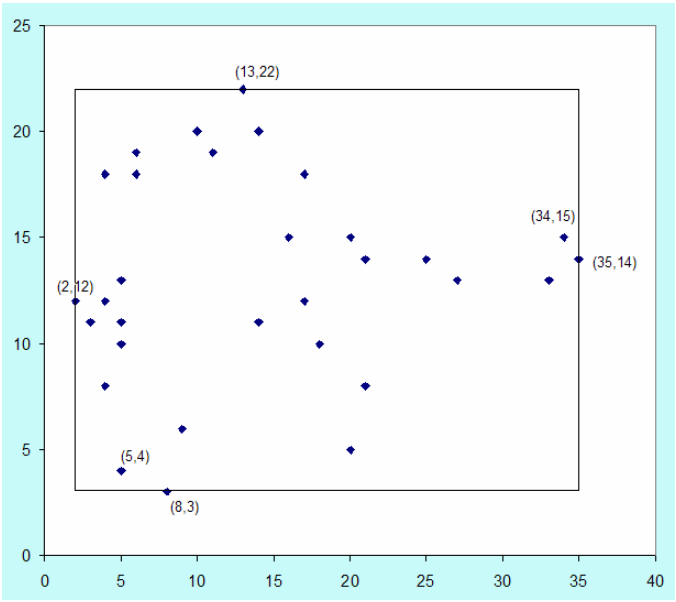


Figure 3.3: The placement of Net clma:661 obtained using Star+

3.3.3 CPU Running Time

One of the major reasons that the HPWL model is so popular is that calculation of a bounding box can be performed quickly. Calculating the bounding box of a net from scratch is linear with respect to the number of blocks in the net. However, Betz et. al. [28] have developed a method, called *incremental bounding box evaluation*, which can re-compute the bounding box in a constant amount of time on average. This is crucial for SA-based placers, because the most computationally expensive part of evaluating a swap is computing the change in cost that the swap would produce.

To implement incremental bounding box evaluation, one has to store the coordinates of the four sides of the bounding box (x_{min}, x_{max}, y_{min}, y_{max}) and the number of blocks on these sides (N_{xmin}, N_{xmax}, N_{ymin}, N_{ymax}) for each net. Figure 3.4 lists the pseudo-code used to update x_{min} and N_{xmin} values for a net.

```
if (x_new != x_old) {                  // block x has moved
        if (x_new < x_min) {           // block x moves outward
                x_min = x_new;         // bounding box becomes larger
                N_xmin = 1;
        }

        else if (x_new = x_min) { // block x lies on the old x_min edge
                N_xmin ++;
        }

        else if (x_old = x_min) {  // block x moves inward
                if (N_xmin >1) {  // bounding box unchanged
                        N_xmin --;
                }
                else {          // bounding box becomes smaller
                        recompute bounding box from scratch
                }
        }
}
```

Figure 3.4: Pseudo-code of incremental bounding box evaluation

The only case for which the net bounding box must be recomputed from scratch is when the block moves inward and it is also the only block on a side of the bounding box. In this case, the re-computation takes *O(k)* time, while in all other cases it is *O(1)*. In contrast, the re-computation of the Star+ model always takes *O(1)* time. To implement the constant time re-computation of the Star+ model, one has to store T_l, which is $\sum_{\forall i \in Net_l} x_i^2$, and S_l, which is $\sum_{\forall i \in Net_l} x_i$, for each net *l*. Figure 3.5 lists the pseudo- code for re-computing the Star+ model of Net *l*.

One immediately notices that the re-computation of the Star+ model is extremely straightforward and very easy to implement. Most importantly, it is *O(1)* in all cases. It should be noted, however, that the Star+ model uses Equation 3.2, which has a square root operation, while the HPWL model requires multiplication and addition operations to be performed.

```
if (x_new != x_old) {                    // block x has moved
        delta = x_new - x_old;
        S += delta;
        T += delta * (x_new + x_old);
}
```

Figure 3.5: Pseudo-code for re-computing the Star+ model of net *l*

Table 3.8 reports the average running time of VPlace when performing the experiments described in Section 3.3.1. Column 1 identifies the benchmark. Columns 2 and 3 give the average† runtime for HPWL and Star+, respectively, when VPlace is run with `inner-num` equal to 1. (Running VPlace with an `inner_num` of 10 simply causes the runtime to increase by a factor of 10 and, hence, is not shown.) The last row

† VPR is run 10 times on each benchmark as per the experiment in Section 3.3.1

of Table 3.8 shows the aggregate runtime for VPR for each model. Clearly, there is very little difference in the actual runtimes.

Table 3.8: CPU Running Time (unit: seconds)

	HPWL	Star+
alu4	2.66	2.91
apex2	3.86	4.06
apex4	2.20	2.30
Bigkey	3.83	3.70
Clma	33.30	33.22
Des	3.47	3.67
Diffeq	2.95	2.95
Dsip	2.80	2.89
Elliptic	10.11	10.09
ex1010	13.55	14.28
ex5p	1.83	1.86
Frisc	9.59	9.94
misex3	2.53	2.69
Pdc	13.24	14.33
S298	3.45	4.00
S38417	22.69	21.66
S38584.1	22.30	21.27
Seq	3.45	3.81
Spla	9.84	10.64
Tseng	1.86	1.89
Total	**169.50**	**172.15**

It should be noted, however, the runtimes reported in Table 3.8 ultimately depend on the number of swaps performed during the optimization process. Thus, the fact that the total time required by Star+ is slightly more than the total time required when using HPWL does not necessarily mean that HPWL is faster to compute. On the contrary, changes in cost can be computed faster using the Star+ model as the size of the nets becomes larger. To illustrate this, we randomly generated nets with cardinalities (number of blocks) of 2, 3, 5, 10, 50, 100, 500, 1000 and 2000. Each net was then randomly placed on a 100 by 100 FPGA chip. We then performed 1 million *improving* swaps/moves on each net, and calculated the average time required by Star+ and HPWL to re-compute the

wirelength estimates. The results are shown below in Table 3.9, where the times recorded in the table are given in nanoseconds. Notice that the time required by HPWL to re-compute the wire-length estimate varies from 19 nanoseconds for a small net with just 2 blocks to 8380 nanoseconds for a large net with 2000 blocks. However, the Star+ model requires a small re-computation time of only 53 nanoseconds for all size nets.

Table 3.9: Re-computing time for HPWL and Star+

Cardinality	**HPWL** (ns)	**Star+** (ns)
2	19	53
3	26	53
5	38	53
10	50	53
50	227	53
100	431	53
500	2127	53
1000	4213	53
2000	8380	53

3.3.4 Wirelength

As a final basis of comparison, we compare the Star+ and HPWL models with respect to the total number of *wire segments* required by VPR's router to route each placement. Table 3.10 reports the total number of wire segments required to route each placement when performing the experiments described in Section 3.3.1. Column 1 identifies the benchmark. Column 2 gives the channel width used by the router. Columns 3 and 4 give the successful times (out of 10) and the average[†] number of wire segments required to route each placement for HPWL when VPlace is run with `inner_num` equal to 1. Columns 5 and 6 provide similar information for the Star+ model. Column 7 gives the p-value. Columns 8 – 12 provide similar information as Columns 3 – 7 when VPlace is run with `inner_num` equal to 10.

[†] VPR is run 10 times on each benchmark as per the experiment in Section 3.3.1

Table 3.10: The number of Wire Segments Needed for Successful Routing

	CW	inner_num 1					Inner_num 10				
		HPWL		Star+		P-value	HPWL		Star+		P-value
		ST	WL	ST	WL		ST	WL	ST	WL	
alu4	11	7	22038.4	9	21090.8	3E-05	9	21016.22	10	20542.7	0.0011
Apex2	13	10	32545.5	10	31361	8E-05	10	30637.5	10	30722.7	0.5886
Apex4	14	8	22865.1	10	22136.1	0.0002	10	21848	10	21448	0.0003
Bigkey	8	10	22395.7	10	22387.8	0.9619	10	18504.6	10	18568.3	0.6616
Clma	14	10	142509	8	138219	0.0012	10	133591.6	10	133427	0.7296
Des	8	10	29161.1	9	29488	0.19	9	24757.67	10	27118.3	6E-05
Diffeq	9	10	16263.4	10	15555.8	0.0002	10	14675.7	10	14502.7	0.0495
Dsip	7	8	17171	9	17599	0.2093	10	14581.7	10	14471.9	0.6589
Elliptic	13	8	53811.4	10	50040	1E-06	10	45912.2	10	45194.5	0.1423
ex1010	13	10	72613.2	10	71220.2	0.0041	10	70864.2	10	69777.7	0.0012
ex5p	15	10	19923.5	9	19583.1	0.084	10	18647.6	10	19067.2	0.0003
Frisc	14	9	59957.2	8	57915.5	0.0018	10	55274	10	56284.4	0.0006
Misex3	13	10	22699.7	10	21751.2	9E-06	10	21870.7	10	20924.5	5E-08
Pdc	19	10	104298	10	103669	0.3293	10	99046.3	10	100052	0.0125
s298	9	10	22703.3	10	22461.5	0.0548	10	21346	10	21688.4	0.0088
s38417	9	10	66586.4	10	66592.4	0.992	10	61764.3	10	61771.1	0.9861
s38584.1	10	10	63514.7	10	60020.6	2E-08	10	57098.7	10	55975.4	0.0042
Seq	13	9	29610.9	9	28377.4	1E-05	10	28059.3	10	27879.4	0.0672
Spla	16	10	71193.8	7	70870	0.4931	10	67361.7	10	69260.1	8E-05
Tseng	8	10	10419.7	10	9932.7	7E-05	10	9423.4	10	9363.2	0.4156
Total		**189**	**902282**	**188**	**880271**		**198**	**836281.4**	**200**	**838039**	

Our results show that when VPlace is run with `inner_num` equal to 1, Star+ results in a lower total number of wire segments compared with HPWL for 17 of the 20 benchmarks. However, the placements found using Star+ are, on average, only 2.4 percent less than those found when using HPWL. In contrast, when VPlace is run with `inner_num` equal to 10, Star+ results in a lower total number of wire segments compared with HPWL for only 11 of the 20 benchmarks. In addition, the number of wire segments required when using the Star+ model is, on average, 0.2 percent more than those found when using HPWL. On closer inspection, Star+ outperforms HPWL for 28 of the 40 cases, with 12 of these cases having a P-value less than 0.05 (indicating that the difference between the two models cannot be attributable to chance).

3.4 Parameter Tuning

The Star+ model contains two adjustable parameters: α and β. As first discussed in Section 3.1, α is responsible for compensating for the average difference between Star+ estimate and the actual number of wire segments used after routing; β is responsible for improving the quality of the placement. In all of the previous experiments, α and β were set to 1.59 and 1.0, respectively. We now discuss how these values were arrived at.

To determine an appropriate value for β, we replaced the HPWL model in VPlace with the Star+ model. We then ran VPlace multiple times on each of the 20 MCNC benchmarks with the parameter β set to 0.5, 0.6, …, 1.5, respectively. As VPlace is stochastic, for each value of β, VPlace was executed five times with five randomly generated seed values. We then used VPR's router to route the resulting placements, and recorded the number of the times a successful routing was found and the average number of wire segments required by the routing solution. When routing the placements, we used the same (minimum) channel widths used in Section 3.3.

The results are shown in Table 3.11, Table 3.12 and Table 3.13. In each table, column 1 identifies the benchmark. Column 2 lists the channel widths used by the router. Columns 3 and 4 indicate the number of times of successful routing found (maximum is 5) and the average number of wire segments, respectively, for a particular value of β. The remaining columns provide similar information for other values of β. For example, considering *Alu4* when β equals 0.7, all 5 attempts at routing are successful, and the average number of wire segments is 21596.

A summary of the results in Table 3.11 – Table 3.13 is given in Table 3.14. With respect to finding placements that lead to successful routing, it can be seen that using a value of β=1 (or β=1.1) results in the largest number (95) of successful routings. With regards to producing placement solutions with the smallest number of wire segments, it can be seen that the smaller β is, the better the placements are. Using β=0.5 resulted in

the smallest number of wire segments (865799) on average. However, it also resulted in the smallest number of successful routable placements been found (81). Given that a placement is of no value if it cannot be routed, we chose to set β=1. However, we would note that the difference in (average) wire-length when β=0.5 and β=1 is very small (approximately 1.7%).

Table 3.11: Routing Results for Different Values of β (between 0.5 and 0.7)

	CW	0.5		0.6		0.7	
		SR	Wire	SR	Wire	SR	Wire
Alu4	11	5	21216	5	21521	5	21596
apex2	13	5	30109	5	30217	5	30381
apex4	14	5	21489	4	21931	5	21978
Bigkey	8	5	22169	5	22486	5	22544
Clma	14	5	135776	5	136797	5	137084
Des	8	5	30811	5	30795	5	30727
Diffeq	9	5	15653	3	15220	3	15223
Dsip	7	5	17557	5	18356	5	18431
Elliptic	13	3	52711	3	51575	4	51641
Ex1010	13	5	64890	5	66034	5	66146
Ex5p	15	5	19867	4	19567	4	19633
Frisc	14	3	57883	2	58686	3	58985
Misex3	13	1	20697	4	21152	4	21261
Pdc	19	1	101621	4	101393	4	101943
s298	9	5	20109	5	20451	5	20570
s38417	9	3	65293	2	65203	3	65122
s38584.1	10	4	62174	5	62682	5	62799
Seq	13	5	27691	5	27695	5	27832
Spla	16	1	67620	2	67582	2	67661
Tseng	8	5	10463	5	10263	5	10291
Total		81	865799	83	869607	87	871850

Table 3.12: Routing Results for Different Values of β (between 0.8 and 1.1)

	0.8		0.9		1		1.1	
	SR	Wire	SR	Wire	SR	Wire	SR	Wire
Alu4	5	21637	5	21899	5	21091	5	21631
apex2	5	30395	5	30718	5	31361	5	30519
apex4	5	22128	5	22174	5	22136	5	22374
Bigkey	5	22539	5	22626	5	22388	5	22595
Clma	5	136881	5	138275	4	138219	5	138937
Des	5	30952	5	31145	5	29488	5	31272
Diffeq	4	15365	4	15468	5	15556	4	15391
Dsip	5	18586	5	18585	5	17599	4	18710
Elliptic	4	52227	4	52148	5	50040	5	52079
Ex1010	5	66013	5	66936	5	71220	5	66621
Ex5p	4	19623	4	19506	4	19583	5	19785
Frisc	3	58534	5	59561	4	57916	4	58539
Misex3	4	21149	4	21107	5	21751	5	21584
Pdc	4	102679	4	101750	5	103669	5	102966
s298	5	20608	5	20782	5	22462	5	20677
s38417	4	65428	4	66277	5	66592	5	66916
s38584.1	5	63447	5	63498	5	60021	5	63168
Seq	5	27658	5	28090	5	28377	5	27816
Spla	3	67942	3	68273	3	70870	3	68498
Tseng	5	10294	5	10293	5	9933	5	10388
Total	90	874087	92	879111	95	880271	95	880465

With β=1, we now turn our attention to finding an appropriate value for α. Recall that α does not affect solution quality directly (like β), but is used as a common multiplier to compensate for the difference between the estimated number of wire segments and the actual number of wire segments required after routing.

Table 3.13: Routing Results for Different Values of β (between 1.2 and 1.5)

	1.2		1.3		1.4		1.5	
	SR	Wire	SR	Wire	SR	Wire	SR	Wire
Alu4	5	21981	5	21746	5	21588	5	21744
apex2	5	30653	5	30357	5	30498	5	30477
apex4	5	21867	5	22234	5	21985	4	22281
Bigkey	5	22526	5	22558	5	22695	5	22689
Clma	5	138894	5	137336	5	137728	5	139416
Des	5	31376	5	31213	5	31255	5	31173
Diffeq	4	15412	4	15351	4	15396	3	15404
Dsip	4	18433	4	18474	4	18417	4	18488
Elliptic	5	51743	5	52539	5	52001	5	52458
ex1010	5	67844	5	66706	5	66825	5	66595
ex5p	5	19750	5	19861	5	19707	5	19785
Frisc	4	59547	3	59126	3	59450	3	58880
Misex3	5	21060	5	21264	5	21423	5	21574
Pdc	4	102678	4	102324	4	102842	4	103980
s298	5	20840	5	20647	5	20890	5	20621
s38417	4	65775	4	65917	3	65214	2	66930
s38584.1	5	63970	5	63245	5	63269	5	63089
Seq	5	27921	5	28083	5	28114	5	28108
Spla	3	68685	2	68461	2	68436	1	68407
Tseng	5	10271	5	10416	5	10279	5	10329
Total	93	881230	91	877857	90	878011	86	882427

Table 3.14: Routing Results for Different Values of β (Summary)

β	SR	Wire
0.5	81	865799
0.6	83	869607
0.7	87	871850
0.8	90	874087
0.9	92	879111
1	95	880271
1.1	95	880465
1.2	93	881230
1.3	91	877857
1.4	90	878011
1.5	86	882427

To determine an appropriate value for α, we experimented with several different values of α. Table 3.15 shows the results with α equal to 1.58, 1.59 and 1.6, respectively. Column 1 identifies the benchmark. Column 2 gives the channel widths used by the router. Column 3 is the average number of wire segments obtained after routing. Column 4 gives the estimated number of wire segments when α equals 1.58. Columns 5 and 6 are the estimates when α equals 1.59 and 1.6, respectively. When α equals 1.59, the total estimate (879712) is closest to the actual number of wire segments after routing (shown in column 3 (880271)). (When α is less than 1.58 or greater than 1.6, the estimates will be even farther away from the number of wire segments after routing, and hence are not listed in the table.)

Table 3.15: Experimental Results of Different α Values

	CW	Routing	1.58	1.59	1.6
Alu4	11	21091	20177	20305	20433
apex2	13	31361	30379	30571	30763
apex4	14	22136	20259	20387	20515
Bigkey	8	22388	21922	22061	22199
Clma	14	138219	146711	147640	148568
Des	8	29488	28119	28297	28475
Diffeq	9	15556	16151	16253	16355
Dsip	7	17599	18509	18626	18744
Elliptic	13	50040	50835	51157	51478
ex1010	13	71220	67680	68108	68537
Ex5p	15	19583	18029	18143	18257
Frisc	14	57916	57200	57562	57924
misex3	13	21751	20817	20949	21080
Pdc	19	103669	98639	99263	99887
s298	9	22462	21050	21183	21317
S38417	9	66592	68123	68555	68986
S38584.1	10	60021	63284	63685	64086
Seq	13	28377	27506	27681	27855
Spla	16	70870	68363	68796	69229
Tseng	8	9933	10426	10492	10558
Total		880271	874180	879712	885245

3.5 Limitations of the Star+ Model

The Star+ model is mainly developed for island-style FPGAs. As other styles of FPGAs (e.g., row-based FPGAs and hierarchical FPGAs) have different routing architectures, using Star+ on these types of FPGAs may provide lower estimation accuracy. For row-based FPGAs, since x-dimension and y-dimension are asymmetric, a possible way to improve the accuracy of Star+ is to try different values for α and β on x- and y-dimensions.

For modern FPGA architectures, there is hard logic (e.g., multipliers, DSP blocks, etc.) at fixed positions on the FPGA chip. If these hard logic blocks are within the area of the Star+ model of a net (see Fig. 3.6 for an example), they affect the available routing resources that may be used to route the net. Therefore, Star+ is not as accurate in this scenario. Consequently, solutions obtained by using analytical methods based on Star+ may suffer a loss in quality. However, it is worthwhile to note that HPWL and quadratic distance also suffer from the same problem.

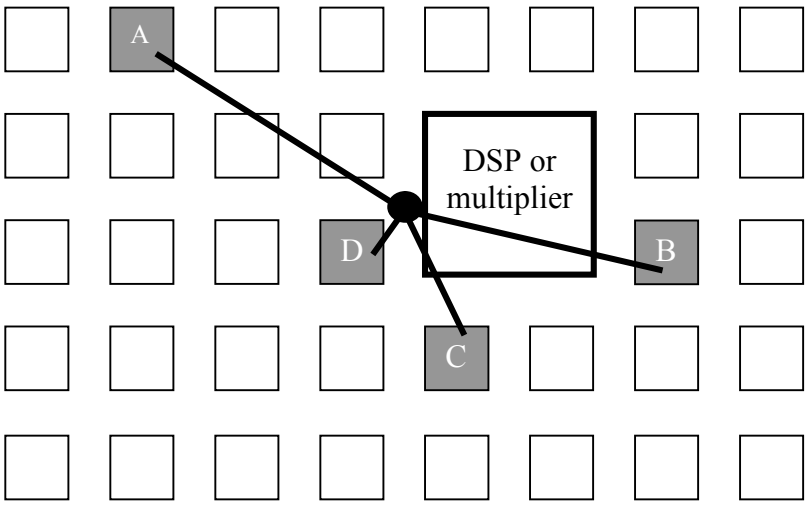


Figure 3.6: A hard logic within the Star+ model of a net

3.6 Summary

In this chapter, we introduced a new wire-length estimation model that is suitable for use both with move-based and analytic placement tools. VPR [28] was used to compare the Star+ model to the traditional HPWL model with respect to routability, critical path delay, CPU running time, and wire-length. Both models were tested using the 20 MCNC [62] benchmarks and with various parameter settings for VPR's placement and routing tools. The following was observed:

- The Star+ model slightly outperforms the HPWL model when `inner_num=1` in terms of minimum channel width and total wire-length.

- The Star+ model outperforms HPWL by 6-9% in terms of critical-path delay, and for 60% of the benchmarks the difference in performance between Star+ and HPWL is statistically significant.

- The Star+ model is differentiable.

- Computing the change in cost resulting from swapping a pair of blocks is always an $O(1)$ operation. Moreover, it was shown that as the net size increases, Star+ outperforms HPWL with respect to the time required to re-compute the wire-length estimate following an improving swap/move.

The effect of the Star+ model's adjustable parameters, α and β, was also studied. The following was decided:

- β affects both the number of placements found that are actually routable and the wire segments required to route a placement. Overall, the best value for β is 1.

- α compensates for the difference between the estimated number of wire segments and the actual number of wire segments required after routing. When set to 1.59, the estimate is closest to the actual number.

Therefore, we conclude that the Star+ model is indeed suitable for use with analytic methods, which we discuss next in Chapter 4.

Chapter 4

Modifying Conjugate Gradient for Placement

Most approaches to analytic placement are based on *quadratic* programming [17]. Quadratic placement algorithms use *squared* wire length as the objective function and try to minimize it by repeatedly solving a system of linear equations. In practice, quadratic placement algorithms are fast and hence capable of handling very large placement problems. However, the quality of solutions produced is often inferior compared with those found using slower, swap-based algorithms, like VPR [27]. This is a partial result of the fact that the objective of quadratic programming is to minimize squared wire length, not *linear* wire length.

To compensate, we propose using the Star+ wire-estimation model presented in Chapter 3 as part of an analytic placement tool based on *Conjugate Gradient (CG)*. The CG method is one of the most popular iterative methods for solving large systems of linear equations. It is very effective for systems of the form $Ax=b$, where x is an unknown vector, b is a known vector, and A is a known, sparse, positive-definite matrix. The essential trade-off in changing a squared wire-length objective into a "near" linear wire-

length objective is that the resulting equations system that must be solved is no longer linear, and hence harder to solve. In particular, two problems arise when employing a non-linear objective function. First, for the FPGA placement problem, the equations system is not sparse, but *dense*. Second, with a traditional solver, like conjugate gradient, the Hessian[†] matrix (i.e., A) must be re-computed on each iteration. Both of these facts result in a runtime for each iteration of $O(n^2)$.

In this Chapter, we show how the runtime of each iteration of conjugate gradient can be reduced to O(n). The basic idea is to avoid computing the Hessian matrix on each iteration by calculating a single value that indicates both the direction and distance to move in the problem's search space.

The remainder of the chapter is organized as follows. In Section 4.1 we provide a brief introduction to conjugate gradient. In Section 4.2 we show how conjugate gradient can be applied to FPGA placement. Finally, in Section 4.3 we summarize our important contributions.

4.1 Conjugate Gradient Method

The earliest conjugate gradient method was devised by Fletcher and Reeves [102]. If the objective function $f(x)$ is quadratic and is minimized exactly in each search direction, it has the desirable feature of converging in at most n iterations because its search directions are conjugate (or A-orthogonal) (see Section 4.1.1), where A is the Hessian matrix of $f(x)$. In practice, conjugate gradient methods are also powerful on general functions. This method represents a major improvement in convergence over steepest-descent methods [32] with only a marginal increase in computational effort compared to the latter. It combines current information about the gradient vector with that of gradient vectors from previous iterations (a memory feature) to obtain the new search direction. The new search direction is computed by a linear combination of the current gradient and

[†] See Section 4.1

the previous search direction. The main advantage of this method is that it requires only a small amount of information to be stored at each stage of calculation and thus can be applied to very large problems.

4.1.1 Standard Conjugate Gradient Algorithm

Consider the problem of minimizing a quadratic function $f(x) = \frac{1}{2}x^T Ax + c^T x$. If A is symmetric and positive-definite, we can find a set of n linearly independent search directions $d_{(0)}, d_{(1)}, \Lambda\ , d_{(n-1)}$ that are mutually conjugate with respect to A, i.e., all the directions satisfy the conjugacy conditions:

$$d_{(i)}^T A d_{(j)} = 0,\ i \neq j, 0 \leq i \text{ and } j < n.$$

The Conjugate Gradient (CG) method is as follows. We start with an initial point $x_{(0)}$ and an initial direction $d_{(0)}$. We minimize $f(x)$ along $d_{(0)}$ to obtain $x_{(1)}$ by making $f'(x_{(1)}) = 0$, and obtain $d_{(1)}$ by letting $d_{(0)}^T A d_{(1)} = 0$. Then, from $x_{(1)}$, we minimize $f(x)$ along $d_{(1)}$ to obtain $x_{(2)}$ by making $f'(x_{(2)}) = 0$, and obtain $d_{(2)}$ by letting $d_{(0)}^T A d_{(2)}$ and $d_{(1)}^T A d_{(2)} = 0$. This procedure may be repeated at most n times. Finally, we minimize $f(x)$ along $d_{(n-1)}$ to obtain $x_{(n)}$ by making $f'(x_{(n)}) = 0$. The point $x_{(n)}$ is the minimum solution. Generally, we can terminate CG early as long as it converges. In fact, the CG iterations can be terminated at any ith iteration $(0 \leq i < n)$ if $x_{(i)}$ is close enough to the minimum. The CG method can also be used to optimize problems where the objective function $f(x)$ is not quadratic, provided it is still positive-definite. In practice, it is usually more effective than direct methods when the equation systems are large and sparse.

To obtain an understanding of what is meant by positive-definite, Figure 4.1 shows the graph of an arbitrary positive-definite function $f(x)$. Figure 4.2 gives the

contours of $f(x)$. The values of $f(x)$ at the points on the same curve are equal. The black dot is the point (degenerate curve) where $f(x)$ has the minimum value.

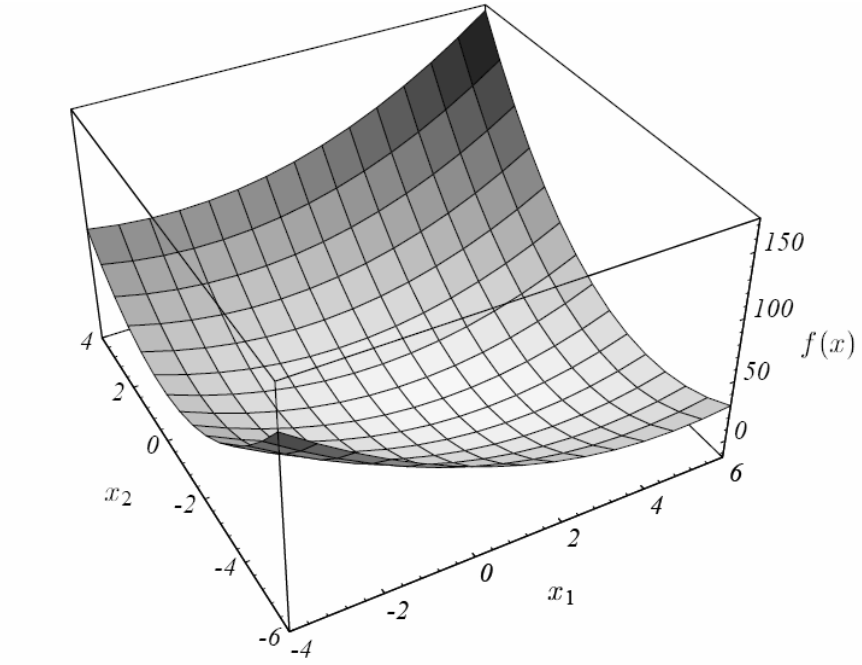


Figure 4.1: The graph of a positive-definite function $f(x)$ [32]

The gradient $f'(x)$ of $f(x)$ is a vector field that, for a given point *x*, points in the direction of the greatest increase of $f(x)$. By Newton's theory, $f'(x)$ equals zero at the point where $f(x)$ is minimum (or maximum). Figure 4.3 illustrates the gradient vectors for $f(x)$. At the bottom of the bowl shown in Figure 4.1, the gradient is zero. One can minimize $f(x)$ by setting $f'(x)=0$.

However, directly solving the equation systems $f'(x)=0$ is usually impractical. An iterative way to find the point *x* at which $f'(x)$ equals zero is to start at an arbitrary *x* and slide down to the bottom of the bowl step-by-step. At each step we move to a new *x* that makes $f'(x)$ closer to zero, and eventually we reach a point *x* at which $f'(x)$ is close enough to zero to enable termination.

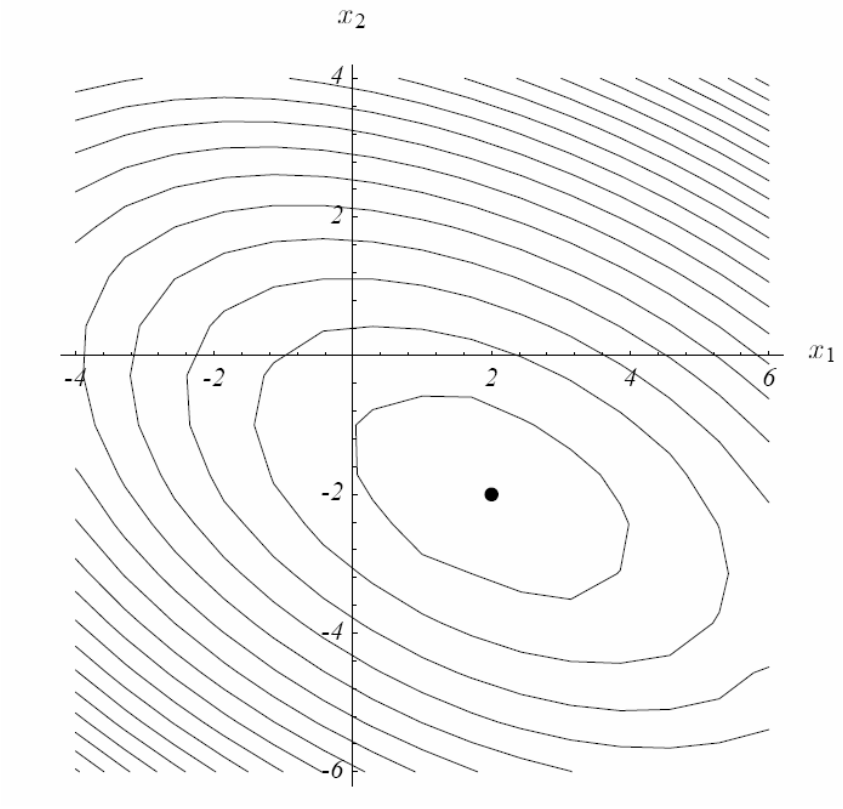


Figure 4.2: The contours of $f(x)$ [32]

The question is which direction and how big a step we should take at each point *x*, so that we can move to the bottom of the bowl as quickly as possible. Assume $x_{(0)}$ is the initial point, $x_{(i)}$ is the point at the *i*th step, $d_{(i)}$ is the direction we should move from point $x_{(i)}$, and $\alpha_{(i)}$ is the size of the move. Thus

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)} \qquad \text{(Equation 4.1)}$$

The CG method uses the following procedure to calculate the direction and how big a step we should take. First, calculate the initial direction:

$$d_{(0)} = r_{(0)} = -f'(x_{(0)}) \qquad \text{(Equation 4.2)}$$

The residual $r_{(0)}$ indicates how far $f'(x)$ is from zero at the initial point $x_{(0)}$ (remember: our goal is to find the x where $f'(x)=0$). Then for every iteration $i \geq 0$, compute the following:

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T f''(x_{(i)}) d_{(i)}} \qquad \text{(Equation 4.3)}$$

The previous equation calculates how far to move along the direction $d_{(i)}$. The Hessian $f''(x_{(i)})$ is an n x n matrix, in which the element at the jth row and kth column is the second partial derivative of $f(x)$ with respect to x_j and x_k at the point $x_{(i)}$. (x is a vector with n elements: $x_1, x_2, \ldots x_n$, where $x_{(i)}$ is the vector at the ith step.) Note that $r_{(i)}^T r_{(i)}$ equals the square of the 2-norm of the residual $r_{(i)}$, and $d_{(i)}^T f''(x_{(i)}) d_{(i)}$ can be looked at as the square of the 2-norm of the direction multiplying coefficient matrix.

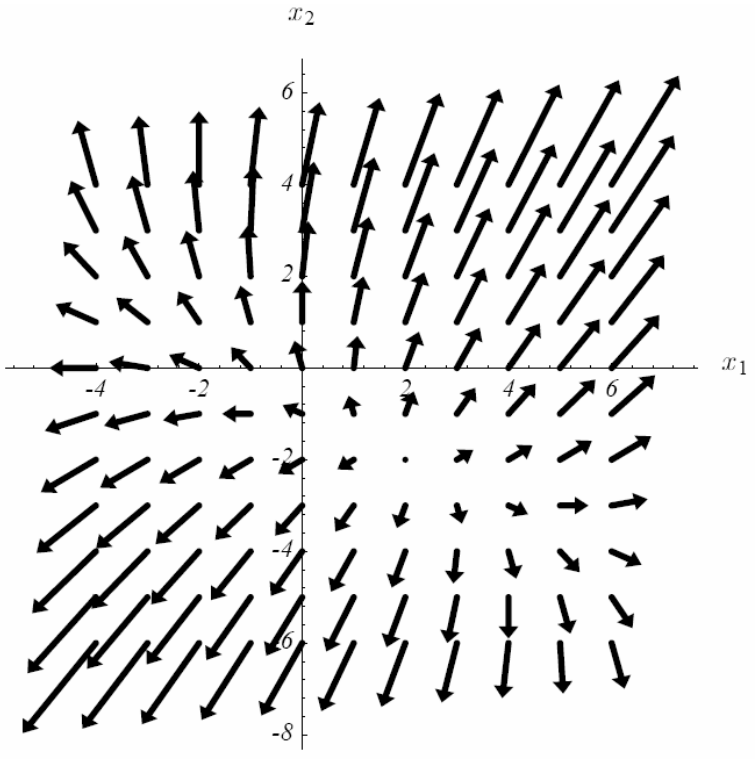


Figure 4.3: The gradient $f'(x)$ of $f(x)$ [32]

$$r_{(i+1)} = -f'(x_{(i+1)}) \qquad \text{(Equation 4.4)}$$

calculates the new residual.

$$\beta_{(i+1)} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}} \qquad \text{(Equation 4.5)}$$

$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)} \qquad \text{(Equation 4.6)}$$

calculates the new direction. When $f(x)$ is quadratic, any two different $d_{(i)}$ s obtained using above formulas will be conjugate to each other. For nonlinear CG (i.e. $f(x)$ is not quadratic), the less similar $f(x)$ is to a quadratic function, and hence the more quickly the directions lose conjugacy. Figure 4.4 suggests the procedure of finding the minimum point using the CG method.

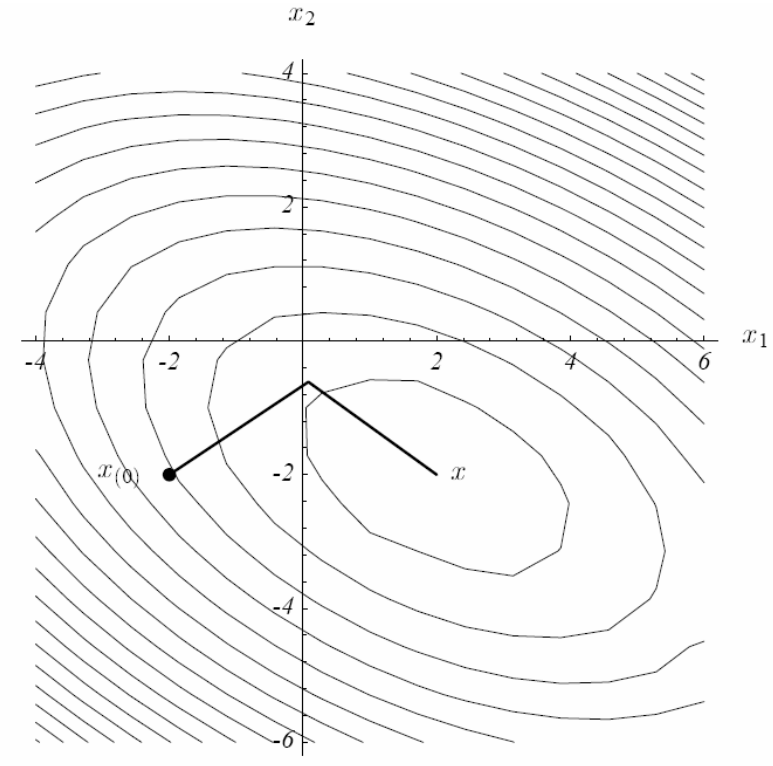


Figure 4.4: The Conjugate Gradient method [32]

The CG method is efficient for sparse systems, in which $f''(x_{(i)})$ is a sparse matrix. For dense systems, the computation of $f''(x_{(i)})$ is $O(n^2)$ and therefore makes each iteration of CG to be $O(n^2)$. In such cases, CG may be undesirable compared with direct methods. However, for specific applications, researchers are often able to find ways to calculate α without re-computing $f''(x_{(i)})$ entirely. In particular, if we can calculate $d_{(i)}^T f''(x_{(i)}) d_{(i)}$ in linear time, then there is no need to spend $O(n^2)$ time computing $f''(x_{(i)})$.

In the next section, we will discuss how to apply CG on FPGA placement and how to keep each iteration of the CG placement algorithm linear in time.

4.2 Conjugate Gradient Placement

The optimization goal of the target CG placement algorithm is to minimize the amount of wire segments required to connect all the nets after a circuit is placed. Since CG requires that the gradient $f'(x)$ and Hessian matrix $f''(x)$ are computable, the objective function $f(x)$ must be built using differentiable estimation models. Hence HPWL is not applicable for the target CG placement algorithm.

4.2.1 Objective Function $f(x)$

In Chapter 3, the Star+ model was compared with the HPWL model and found to be at least as effective as HPWL with respect to wire-length and routability, and often better for critical-path delay. However, and most importantly, the Star+ model is also *differentiable*, and hence suitable for use in analytic methods. Based on the Star+ model (see Equation 3.1), the wire-length estimate of a net after being placed can be calculated as:

$$\|Net_l\| = \alpha \times \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} + \alpha \times \sqrt{\sum_{\forall i \in Net_l} (y_i - y_{cl})^2 + \beta}$$

Since the total wire-length estimate of a circuit is the sum of the wire-length estimates of all the nets, the wire-length estimate can be expressed as follows:

$$\begin{aligned} &\sum_l \left(\alpha \times \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} + \alpha \times \sqrt{\sum_{\forall i \in Net_l} (y_i - y_{cl})^2 + \beta} \right) \\ &= \sum_l \left(\alpha \times \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} \right) + \sum_l \left(\alpha \times \sqrt{\sum_{\forall i \in Net_l} (y_i - y_{cl})^2 + \beta} \right) \end{aligned} \quad \text{(Equation 4.7)}$$

where x_i and y_i are the coordinates of Block i, and x_{cl} and y_{cl} are the coordinates of the center of gravity (see Section 3.1).

Equation 4.7 consists of two parts. The first part that contains only x-coordinates is the wire-length estimate along x-coordinate axis, and the second part that contains only y-coordinates is the wire-length estimate along the y-coordinate axis. All of the x_is are independent of any y_is, and vice-versa. Thus, minimizing Equation 4.7 is equivalent to minimizing each part separately. For the sake of simplicity, we only discuss the wire-length estimate along x-coordinate axis. The part that estimates the wire-length along y-coordinate can be solved in a similar way.

Based on Equation 4.7, we define the objective function $f(x)$ as:

$$f(x) = \sum_l \left(\alpha \times \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} \right) \quad \text{(Equation 4.8)}$$

4.2.2 Gradient $f'(x)$

To implement CG, we need to calculate the gradient of $f(x)$. The gradient $f'(x)$ is defined as:

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \text{M} \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix}$$

The $f'(x)$ is a vector that points in the direction of the greatest increase of $f(x)$ at a given point $x = (x_1, x_2, \Lambda\ , x_n)^T$. In order to make calculation simpler, we define

$$S_l = \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} \qquad \text{(Equation 4.9)}$$

Therefore, $f(x) = \sum_l (\alpha \cdot S_l) = \alpha \sum_l S_l$ (Equation 4.10)

To calculate $\frac{\partial}{\partial x_j} f(x)$, we calculate $\frac{\partial S_l}{\partial x_j}$ first. From the definition of S_l, we have:

$$\begin{aligned} S_l &= \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} \\ &= \sqrt{\sum_{\forall i \in Net_l} (x_i^2 - 2x_i x_{cl} + x_{cl}^2) + \beta} \\ &= \sqrt{\sum_{\forall i \in Net_l} x_i^2 - \sum_{\forall i \in Net_l} 2x_i x_{cl} + \sum_{\forall i \in Net_l} x_{cl}^2 + \beta} \\ &= \sqrt{\sum_{\forall i \in Net_l} x_i^2 - 2x_{cl} \sum_{\forall i \in Net_l} x_i + x_{cl}^2 \sum_{\forall i \in Net_l} 1 + \beta} \\ &= \sqrt{\sum_{\forall i \in Net_l} x_i^2 - 2x_{cl} \cdot k_l x_{cl} + x_{cl}^2 \cdot k_l + \beta} \end{aligned}$$

$$= \sqrt{\sum_{\forall i \in Net_l} x_i^2 - 2k_l x_{cl}^2 + k_l x_{cl}^2 + \beta}$$

$$= \sqrt{\sum_{\forall i \in Net_l} x_i^2 - k_l x_{cl}^2 + \beta}$$

If $j \notin Net_l$, it is obvious that $\frac{\partial S_l}{\partial x_j} = 0$. If $j \in Net_l$, the partial derivative of S_l with respect to x_j is:

$$\frac{\partial S_l}{\partial x_j} = \frac{\partial}{\partial x_j} \sqrt{\sum_{\forall i \in Net_l} x_i^2 - k_l x_{cl}^2 + \beta}$$

$$= \frac{1}{2\sqrt{\sum_{\forall i \in Net_l} x_i^2 - k_l x_{cl}^2 + \beta}} \frac{\partial}{\partial x_j} \left(\sum_{\forall i \in Net_l} x_i^2 - k_l x_{cl}^2 + \beta \right)$$

$$= \frac{1}{2S_l} \frac{\partial}{\partial x_j} \left[\sum_{(\forall i \in Net_l) \wedge (i \neq j)} x_i^2 + \sum_{(\forall i \in Net_l) \wedge (i = j)} x_i^2 - k_l x_{cl}^2 + \beta \right]$$

$$= \frac{1}{2S_l} \left[\frac{\partial}{\partial x_j} \sum_{(\forall i \in Net_l) \wedge (i \neq j)} x_i^2 + \frac{\partial}{\partial x_j} \sum_{(\forall i \in Net_l) \wedge (i = j)} x_i^2 - \frac{\partial}{\partial x_j} k_l x_{cl}^2 + \frac{\partial}{\partial x_j} \beta \right]$$

$$= \frac{1}{2S_l} \left[\frac{\partial}{\partial x_j} \sum_{(\forall i \in Net_l) \wedge (i \neq j)} x_i^2 + \frac{\partial}{\partial x_j} x_j^2 - k_l \frac{\partial}{\partial x_j} x_{cl}^2 + \frac{\partial}{\partial x_j} \beta \right]$$

$$= \frac{1}{2S_l} \left[2 \sum_{(\forall i \in Net_l) \wedge (i \neq j)} x_i \frac{\partial x_i}{\partial x_j} + 2x_j \frac{\partial x_j}{\partial x_j} - 2k_l x_{cl} \frac{\partial x_{cl}}{\partial x_j} + \frac{\partial \beta}{\partial x_j} \right]$$

Because $\frac{\partial x_i}{\partial x_j} = 0$ (when $i \neq j$), $\frac{\partial x_j}{\partial x_j} = 1$, $\frac{\partial x_{cl}}{\partial x_j} = \frac{1}{k_l}$ and $\frac{\partial \beta}{\partial x_j} = 0$, the above equation can be simplified as follows:

$$\frac{\partial S_l}{\partial x_j} = \frac{1}{2S_l} \left(2 \sum_{(\forall i \in Net_l) \wedge (i \neq j)} x_i \times 0 + 2x_j \times 1 - 2k_l x_{cl} \frac{1}{k_l} + 0 \right)$$

$$= \frac{1}{2S_l} \left(2x_j - 2x_{cl} \right)$$

$$= \frac{x_j - x_{cl}}{S_l}$$

In general,

$$\frac{\partial S_l}{\partial x_j} = \begin{cases} \frac{x_j - x_{cl}}{S_l}, & \text{if } j \in Net_l \\ 0, & \text{if } j \notin Net_l \end{cases} \qquad \text{(Equation 4.11)}$$

Based on Equation 4.10 and 4.11, the partial derivative of $f(x)$ with respect to x_j is:

$$\frac{\partial}{\partial x_j} f(x) = \frac{\partial}{\partial x_j}\left(\alpha \sum_l S_l\right) \qquad \text{(From Equation 4.10)}$$

$$= \alpha\left(\sum_l \frac{\partial S_l}{\partial x_j}\right)$$

$$= \alpha\left(\sum_{\forall l: j \in Net_l} \frac{\partial S_l}{\partial x_j} + \sum_{\forall l: j \notin Net_l} \frac{\partial S_l}{\partial x_j}\right)$$

$$= \alpha\left(\sum_{\forall l: j \in Net_l} \frac{x_j - x_{cl}}{S_l} + \sum_{\forall l: j \notin Net_l} 0\right) \qquad \text{(From Equation 4.11)}$$

$$= \alpha \sum_{\forall l: j \in Net_l} \frac{x_j - x_{cl}}{S_l} \qquad \text{(Equation 4.12)}$$

Recall that these values are the (j^{th}) components of vector $f'(x)$, the gradient of $f(x)$.

4.2.3 Hessian Matrix $f''(x)$

CG requires computing Hessian matrix. Each element of the Hessian matrix $f''(x)$ is a second-order partial derivative of $f(x)$, which can be calculated using following procedures.

The diagonal elements (at *j*th row and *j*th column) are:

$$\frac{\partial^2}{\partial x_j^2} f(x) = \frac{\partial}{\partial x_j}\left(\alpha \sum_{\forall l: j \in Net_l} \frac{x_j - x_{cl}}{S_l}\right) \qquad \text{(From Equation 4.12)}$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{\partial}{\partial x_j}\frac{x_j - x_{cl}}{S_l}\right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{1}{S_l}\frac{\partial}{\partial x_j}(x_j - x_{cl}) + (x_j - x_{cl})\frac{\partial}{\partial x_j}\frac{1}{S_l}\right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{1}{S_l}(1 - \frac{1}{k_l}) + (x_j - x_{cl})\frac{-1}{S_l^2}\frac{\partial S_l}{\partial x_j}\right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{1}{S_l}(1 - \frac{1}{k_l}) - \frac{1}{S_l}\frac{x_j - x_{cl}}{S_l}\frac{\partial S_l}{\partial x_j}\right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{1}{S_l}(1 - \frac{1}{k_l} - \frac{x_j - x_{cl}}{S_l}\frac{\partial S_l}{\partial x_j})\right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left[\frac{1}{S_l}\left(1 - \frac{1}{k_l} - (\frac{\partial S_l}{\partial x_j})^2\right)\right] \qquad \text{(From Equation 4.11)}$$

Each off-diagonal element at *k*th row and *j*th column is ($j \neq k$):

$$\frac{\partial^2}{\partial x_k \partial x_j} f(x) = \frac{\partial}{\partial x_k}\left(\alpha \sum_{\forall l: j \in Net_l} \frac{x_j - x_{cl}}{S_l}\right) \qquad \text{(From Equation 4.12)}$$

$$= \alpha\left(\sum_{\forall l: j \in Net_l} \frac{\partial}{\partial x_k}\frac{x_j - x_{cl}}{S_l}\right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{1}{S_l}\frac{\partial}{\partial x_k}(x_j - x_{cl}) + (x_j - x_{cl})\frac{\partial}{\partial x_k}\frac{1}{S_l}\right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{1}{S_l}(0 - \frac{\partial x_{cl}}{\partial x_k}) + (x_j - x_{cl})\frac{-1}{S_l^2}\frac{\partial S_l}{\partial x_k}\right) \qquad (\text{When } j \neq k,\ \frac{\partial x_j}{\partial x_k} = 0)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{-1}{S_l} \frac{\partial x_{cl}}{\partial x_k} + \frac{-1}{S_l} \frac{x_j - x_{cl}}{S_l} \frac{\partial S_l}{\partial x_k} \right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{1}{S_l} (-\frac{\partial x_{cl}}{\partial x_k} - \frac{x_j - x_{cl}}{S_l} \frac{\partial S_l}{\partial x_k}) \right)$$

$$= \alpha \sum_{\forall l: j \in Net_l} \left(\frac{1}{S_l} (-\frac{\partial x_{cl}}{\partial x_k} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k}) \right) \qquad \text{(From Equation 4.11)}$$

$$= \alpha \left[\sum_{\forall l: j \in Net_l \wedge k \in Net_l} \left(\frac{1}{S_l} (-\frac{\partial x_{cl}}{\partial x_k} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k}) \right) + \sum_{\forall l: j \in Net_l \wedge k \notin Net_l} \left(\frac{1}{S_l} (-\frac{\partial x_{cl}}{\partial x_k} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k}) \right) \right]$$

$$= \alpha \left[\sum_{\forall l: j \in Net_l \wedge k \in Net_l} \left(\frac{1}{S_l} (-\frac{\partial x_{cl}}{\partial x_k} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k}) \right) + 0 \right] \quad (\text{When } k \notin Net_l, \ \frac{\partial S_l}{\partial x_k} = \frac{\partial x_{cl}}{\partial x_k} = 0)$$

$$= \alpha \sum_{\forall l: j,k \in Net_l} \left(\frac{1}{S_l} (-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k}) \right)$$

To summarize, we have the elements of Hessian matrix $f''(x)$:

$$\frac{\partial^2 f(x)}{\partial x_j \partial x_k} = \begin{cases} \alpha \sum_{\forall l: j \in Net_l} \left[\frac{1}{S_l} \left(1 - \frac{1}{k_l} - (\frac{\partial S_l}{\partial x_j})^2 \right) \right], & \text{if } j = k \\ \alpha \sum_{\forall l: j,k \in Net_l} \left(\frac{1}{S_l} (-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k}) \right), & \text{if } j \neq k \end{cases} \qquad \text{(Equation 4.13)}$$

The computation of $f''(x)$ is O(n^2), which is too long for iterative methods. Since $f''(x)$ is only used in Equation 4.2, if we can calculate $d^T f''(x) d$ directly in a shorter time we do not have to calculate $f''(x)$ especially. Fortunately, the following method computes $d^T f''(x) d$ in linear time. The direction vector is $d = (d_1, d_2, \text{K}, d_n)^T$. The product $d^T f''(x) d$ is a single real number. By the definition of the product of a vector and a matrix, we have:

$$d^T f''(x) d = \sum_j \sum_k d_j \frac{\partial^2 f(x)}{\partial x_j \partial x_k} d_k \qquad \text{(Equation 4.14)}$$

$$= \sum_j \left(\sum_{k=j} d_j \frac{\partial^2 f(x)}{\partial x_j \partial x_k} d_k + \sum_{k \neq j} d_j \frac{\partial^2 f(x)}{\partial x_j \partial x_k} d_k \right)$$

$$= \sum_j \left\{ \sum_{k=j} d_j \alpha \sum_{\forall l: j \in Net_l} \left[\frac{1}{S_l} \left(1 - \frac{1}{k_l} - \left(\frac{\partial S_l}{\partial x_j} \right)^2 \right) \right] d_k + \sum_{k \neq j} d_j \alpha \sum_{\forall l: j,k \in Net_l} \left(\frac{1}{S_l} \left(-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k} \right) \right) d_k \right\}$$

(From Equation 4.13)

$$= \alpha \sum_j \left\{ \sum_{k=j} d_j \sum_{\forall l: j \in Net_l} \frac{1}{S_l} d_k + \sum_{k=j} d_j \sum_{\forall l: j \in Net_l} \left[\frac{1}{S_l} \left(-\frac{1}{k_l} - \left(\frac{\partial S_l}{\partial x_j} \right)^2 \right) \right] d_k + \sum_{k \neq j} d_j \sum_{\forall l: j,k \in Net_l} \left(\frac{1}{S_l} \left(-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k} \right) \right) d_k \right\}$$

$$= \alpha \sum_j \left\{ \sum_{k=j} d_j \sum_{\forall l: j \in Net_l} \frac{1}{S_l} d_k + \sum_{k} d_j \sum_{\forall l: j,k \in Net_l} \left(\frac{1}{S_l} \left(-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k} \right) \right) d_k \right\}$$

$$= \alpha \left\{ \sum_j \sum_{\forall l: j \in Net_l} \frac{1}{S_l} d_j d_j + \sum_j \sum_k \sum_{\forall l: j,k \in Net_l} \left(\frac{1}{S_l} \left(-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k} \right) \right) d_j d_k \right\}$$

$$= \alpha \left\{ \sum_{\forall l} \sum_{\forall j \in Net_l} \frac{1}{S_l} d_j d_j + \sum_{\forall l} \sum_{\forall j \in Net_l} \sum_{\forall k \in Net_l} \left(\frac{1}{S_l} \left(-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k} \right) \right) d_j d_k \right\}$$

$$= \alpha \sum_{\forall l} \left\{ \sum_{\forall j \in Net_l} \frac{1}{S_l} d_j d_j + \sum_{\forall j \in Net_l} \sum_{\forall k \in Net_l} \left(\frac{1}{S_l} \left(-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k} \right) \right) d_j d_k \right\}$$

$$= \alpha \sum_{\forall l} \frac{1}{S_l} \left\{ \sum_{\forall j \in Net_l} d_j d_j + \sum_{\forall j \in Net_l} \sum_{\forall k \in Net_l} \left(-\frac{1}{k_l} - \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k} \right) d_j d_k \right\}$$

$$= \alpha \sum_{\forall l} \frac{1}{S_l} \left\{ \sum_{\forall j \in Net_l} d_j d_j - \sum_{\forall j \in Net_l} \sum_{\forall k \in Net_l} \frac{1}{k_l} d_j d_k - \sum_{\forall j \in Net_l} \sum_{\forall k \in Net_l} \frac{\partial S_l}{\partial x_j} \frac{\partial S_l}{\partial x_k} d_j d_k \right\}$$

$$= \alpha \sum_{\forall l} \frac{1}{S_l} \left\{ \sum_{\forall j \in Net_l} d_j d_j - \frac{1}{k_l} \sum_{\forall j \in Net_l} \sum_{\forall k \in Net_l} d_j d_k - \sum_{\forall j \in Net_l} \sum_{\forall k \in Net_l} \frac{\partial S_l}{\partial x_j} d_j \frac{\partial S_l}{\partial x_k} d_k \right\}$$

$$= \alpha \sum_{\forall l} \frac{1}{S_l} \left\{ \sum_{\forall j \in Net_l} d_j^2 - \frac{1}{k_l} \sum_{\forall j \in Net_l} \left(d_j \sum_{\forall k \in Net_l} d_k \right) - \sum_{\forall j \in Net_l} \left(\frac{\partial S_l}{\partial x_j} d_j \sum_{\forall k \in Net_l} \frac{\partial S_l}{\partial x_k} d_k \right) \right\}$$

We know that $\sum_{\forall k \in Net_l} d_k$ and $\sum_{\forall k \in Net_l} \frac{\partial S_l}{\partial x_k} d_k$ are independent of j. We also have $\sum_{\forall j \in Net_l} d_j = \sum_{\forall k \in Net_l} d_k$ and $\sum_{\forall j \in Net_l} \frac{\partial S_l}{\partial x_j} d_j = \sum_{\forall k \in Net_l} \frac{\partial S_l}{\partial x_k} d_k$. Therefore, the above equation (Equation 4.14) can be simplified to:

$$d^T f''(x) d = \alpha \sum_{\forall l} \frac{1}{S_l} \left\{ \sum_{\forall j \in Net_l} d_j^2 - \frac{1}{k_l} \left(\sum_{\forall j \in Net_l} d_j \right) \left(\sum_{\forall k \in Net_l} d_k \right) - \left(\sum_{\forall j \in Net_l} \frac{\partial S_l}{\partial x_j} d_j \right) \left(\sum_{\forall k \in Net_l} \frac{\partial S_l}{\partial x_k} d_k \right) \right\}$$

$$= \alpha \sum_{\forall l} \frac{1}{S_l} \left\{ \sum_{\forall j \in Net_l} d_j^2 - \frac{1}{k_l} \left(\sum_{\forall j \in Net_l} d_j \right)^2 - \left(\sum_{\forall j \in Net_l} \frac{\partial S_l}{\partial x_j} d_j \right)^2 \right\} \qquad \text{(Equation 4.15)}$$

Although using Equations 4.14 and 4.15 will give the same result, the time complexities to calculate these two equations are significantly different. The complexity of Equation 4.14 is $O(n^2)$, as the Hessian matrix has n^2 elements. In contrast, the complexity of Equation 4.15 is $O(n)$. We will clarify this in the next paragraph.

We know computing x_{cl} and S_l are both O(k_l). From Equation 4.11, calculating $\frac{\partial S_l}{\partial x_j}$ is $O(1)$ if we already know x_{cl} and N_l. In Equation 4.15, the entire calculation in the curved brackets is $O(k_l)$ (i.e., linear to the cardinality of Net l). Therefore, the time complexity of Equation 4.15 is linear to the sum of the cardinalities of all the nets. As the sum of the cardinalities of all the nets equals the sum of the fan in/outs of all the blocks, the complexity is also linear to the sum of the fan in/outs of all the blocks. Due to the physical limit of FPGA architecture, each block can only connect to a certain number of nets. That means the sum of the fan in/outs of all the blocks is $O(n)$ (n is the number of blocks). As a result, the time complexity of Equation 4.15 is $O(n)$. (For example, our experimentation shows that for a benchmark with 1500 blocks and 1500 nets, the CG placement that calculates $f''(x)$ each iteration is about 70 times slower compared with the CG placement that uses Equation 4.15 to calculate $d^T f''(x) d$ directly. Since

calculating $f''(x)$ is in $O(n^2)$, it will be even slower when the size of benchmark is larger.)

Up to now, we have introduced all the basic theory of the CG placement algorithm. Figure 4.5 gives the pseudo code as a summary of the whole procedures. The first six lines initialize all x-coordinates, and calculate the initial residue r_0 and search direction d_0. Within the while loop, each iteration successively calculates $\alpha = (\frac{r^T r}{d^T f''(x) d})$, updates x, calculates new residue $newr$, $\beta = (\frac{newr^T newr}{r^T r})$ and search direction d. The iterations are terminated when i reaches the maximum number of iterations.

4.3 Conclusion

In this chapter, we presented an analytic placement algorithm that uses conjugate gradient method and the Star+ net model. An important feature of the algorithm is that the computation complexity of each iteration is $O(n)$ even though the target system is not sparse. In the next chapter, we will present a pre-placement algorithm that initializes the coordinates of the blocks and will describe a bi-partitioning method that legalizes the solutions obtained from the CG placement algorithm. These additional algorithms are needed to make CG practical for obtaining good placements.

Initialize all x_i s
For each net l
{

$$x_{cl} = \frac{1}{k_l} \sum_{\forall i \in Net_l} x_i$$

$$S_l = \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + 1}$$

For each block $j \in Net_l$, $\frac{\partial S_l}{\partial x_j} = \frac{x_j - x_{cl}}{S_l}$

}

For each block j, $d_j = r_j = -\frac{\partial}{\partial x_j} f(x) = -\sum_{\forall l: j \in Net_l} \frac{\partial S_l}{\partial x_j}$

$i = 0$

While $i <$ n
{

$$d^T f''(x) d = \sum_{\forall l} \frac{1}{S_l} \left\{ \sum_{\forall j \in Net_l} d_j^2 - \frac{1}{k_l} \left(\sum_{\forall j \in Net_l} d_j \right)^2 - \left(\sum_{\forall j \in Net_l} \frac{\partial S_l}{\partial x_j} d_j \right)^2 \right\}$$

$x = x + (\frac{r^T r}{d^T f''(x) d}) d$ //update all x coordinates

For each net l
{

$$x_{cl} = \frac{1}{k_l} \sum_{\forall i \in Net_l} x_i$$

$$S_l = \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + 1}$$

For each block $j \in Net_l$, $\frac{\partial S_l}{\partial x_j} = \frac{x_j - x_{cl}}{S_l}$

}

For each block j, $newr_j = -\frac{\partial}{\partial x_j} f(x) = -\sum_{\forall l: j \in Net_l} \frac{\partial S_l}{\partial x_j}$

$d = newr + (\frac{newr^T newr}{r^T r}) d$
$r = newr$
$i = i + 1$

} //end of while

Figure 4.5: Pseudo-code of CG placement algorithm

Chapter 5

Pre-Placement and Legalization Methods

In Chapter 4, we introduced the *theoretical background* of our conjugate-gradient placement paradigm. In this Chapter, we introduce the *essential components* of our conjugate-gradient placement prototype. More specifically, in Section 5.1 we introduce a new algorithm for temporarily pre-placing I/O pads on to the FPGA. Without pre-assigning I/O pads, minimizing wire length would be vacuous, as collapsing all moveable blocks onto one point (CLB) would yield the best possible objective function value of $f'(x) = 0$. In Section 5.2, we briefly describe the recursive partitioning approach [86] that is employed to realize legal placements. As discussed in Chapter 4, because the Star+ wirelength is separable into horizontal and vertical components, numerical optimization can be applied independently in both directions to obtain (x,y) coordinates for each moveable block. In practice, however, blocks tend to overlap and concentrate in the center of the FPGA. Consequently, a legalization step is required to map the "global placement" (actually, a "continuous solution obtained using a non-linear objective") back to the original discrete problem. In Section 5.3, we provide an overall description of the *Conjugate Gradient* (*CG*) placement algorithm. In Section 5.4, we discuss the

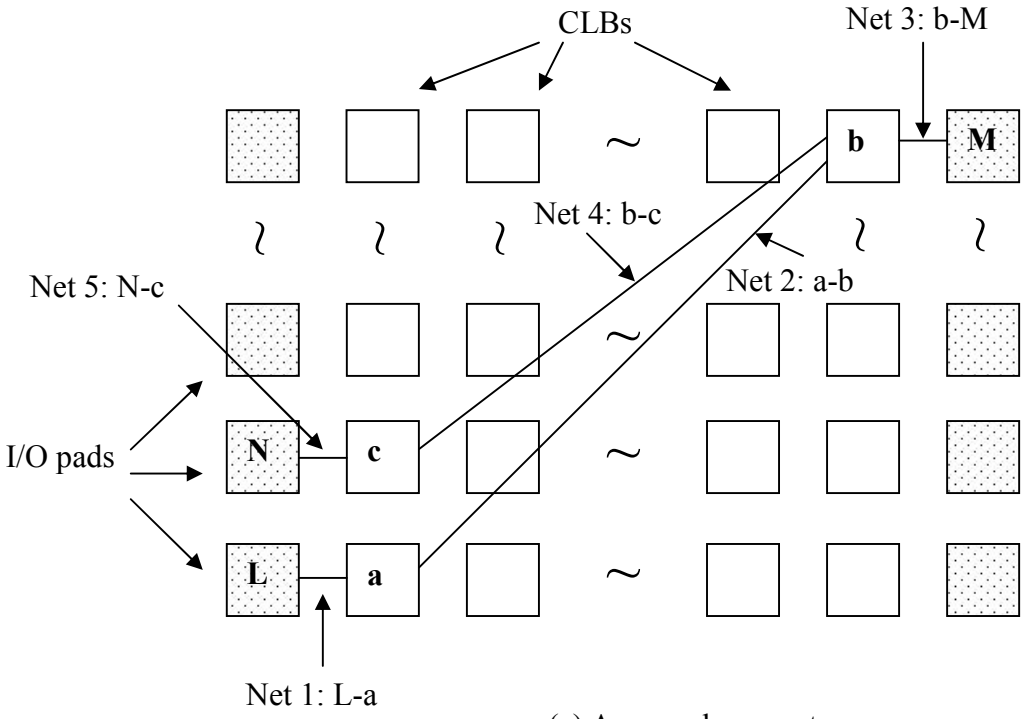
convergence of CG placement algorithm. In Section 5.5, we present the experimental results of Shrubbery and CGH. Finally, in Section 5.6 we provide a summary of the main contributions of this Chapter.

5.1 I/O Pad Pre-placement

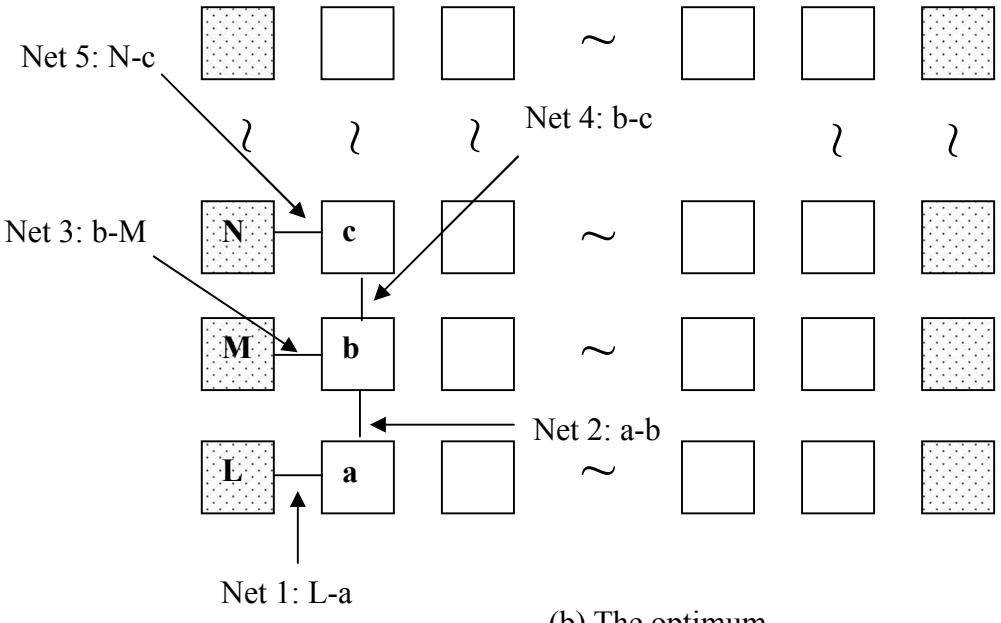
In general, pre-placement is a procedure that temporarily assigns some I/O pads and/or logic blocks to certain locations on the FPGA. It is an essential step of any analytic placement method; without pre-placement, only trivial solutions will be obtained when solving the equation system $f'(x)=0$. For example, the solution where all of the x_is are zero will make $f'(x)=0$.

To avoid obtaining trivial solutions, we choose to pre-place all of the I/O pads. The reason that we choose to pre-place I/O pads rather than the logic blocks is because I/O-pad placement is an easier, *one-dimensional* problem to solve. We use a novel graph-based pre-placement algorithm, which we call *Shrubbery*, to achieve a pre-placement of I/O pads with reasonable quality, prior to placing logic blocks. This pre-placement is *not* the final placement of I/O pads, but *helps* with the placement of logic blocks later on as some of the logic blocks share connections with the various I/O blocks.

The *goal* of the pre-placement is to place the I/O pads in such a way that those with higher connectivity are placed closer together than the I/O pads with lower connectivity. Also, this pre-placement provisionally locates components on the periphery of the FPGA, which causes other components to be distributed throughout the chip. Figure 5.1(a) shows a simple example with three I/O pads **L**, **M** and **N**, three logic blocks **a**, **b** and **c**, and five nets Net 1: L-a, Net 2: a-b, Net 3: b-M, Net 4: b-c, and Net 5: N-c. The connection between I/O pads **L** and **M** involves three nets Net 1: L-a, Net 2: a-b, and Net 3: b-M, while the connection between I/O pads **L** and **N** involves four nets Net 1: L-a, Net 2: a-b, Net 4: b-c, and Net 5: N-c.



(a) A poor placement



(b) The optimum

Figure 5.1: The pre-placement of I/O blocks

In this case, the quality of placement in Figure 5.1(a) is poor. This is partially

caused by I/O pad **M** being placed too far away from I/O pads **L** and **N**. Moving logic block **b** towards logic blocks **a** and **c** improves the placement, but this movement increases the amount of wire needed to connect **b** and **M**. However, if we move I/O pad **M** beside I/O pads **L** and **N**, as shown in Fig. 5.1(b), we will use the least amount of wire to connect the blocks. This is what the I/O pad pre-placement technique presented in the next section intends to do.

5.1.1 Terminology

Throughout the remainder of this section, we use the following definitions and terminology when describing *Shrubbery*.

The following symbols index scalar objects or *sets* of scalar objects: *i* and *j* are used to index vertices; *a* and *b* are used to index root vertices of shrubs, making them shrub, grove, or hedge identifiers. An edge (e_{ij}) in a graph is identified by the pair of vertices v_i and v_j it connects. The cost or *weight* of an edge e_{ij} in a graph is identified by w_{ij}. We also make use of the following *definitions* (see Fig. 5.2 for assistance):

Definition 5.1: A **shrub** S_a consists of a set of edges S_a^e and a set of vertices S_a^v where v_a is the unique terminal (root) in S_a^v. The edges in S_a^e form a tree rooted at v_a, meeting all vertices of S_a^v, and all $e_{ij} \in S_a^e$ have end vertices $v_i, v_j \in S_a^v$. Shrubs will "grow" to encompass more vertices as the algorithm proceeds.

Once an edge or a vertex is included in a shrub, its *shrub membership* $s(v_i)$ does not change. The function $s(v_i) = undefined$, if vertex v_i does not belong to any shrub. Otherwise, $s(v_i) = a$, the identifying root of the containing shrub. Note that a vertex can belong to at most one shrub.

Definition 5.2: A **Path** P_{ij} is a sequence of distinct edges connecting vertices v_i and v_j. Two paths P_{ik} and P_{kj} with no common vertices except for vertex v_j can be concatenated

to form a single, longer path; that is, $P_{ik} \parallel P_{kj}$ is the union of distinct edges and vertices in P_{ik} and P_{kj}. A path P_{ai} entirely within a shrub represents the unique path, by which vertex v_i was reached from root vertex v_a during shrub growth. The function $N(P_{ai})$ returns a list of all vertices in the path P_{ai}. The function $E(P_{ai})$ returns a list of all edges in the path P_{ai}.

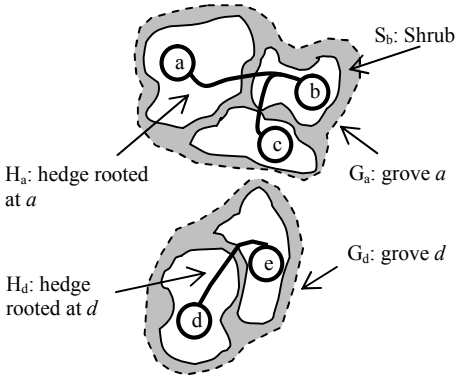


Figure 5.2: Illustration of shrub, hedge, and grove

Definition 5.3: A **Steiner Tree** is a tree connecting a subset of vertices, called terminals, in an undirected, weighted graph. In the context of placement, a minimum weight Steiner tree is a measure of placement quality (wirelength) and represents the optimal way to connect a net (set of pins that must be connected together).

Definition 5.4: A **grove** G_a is a union of shrubs $\bigcup_{b \in M} S_b$ where M is an index set of member shrubs and $a = \min_k \{k \in M\}$. More specifically, $G_a^e = \bigcup_{b \in M} S_b^e$ and $G_a^n = \bigcup_{b \in M} S_b^n$.

Definition 5.5 (Hedge) A **hedge** H_a is the partial Steiner tree that connects the terminals of grove G_a . H_a consists of two sets $H_a^e \subseteq G_a^e$ and $H_a^n \subseteq G_a^n$, its edges and vertices, respectively.

Definition 5.6 (Root-distance) Every vertex v_k within a shrub S_a has a **root-distance** d_k^a, which is the cost of the path by which vertex v_k was reached from root of S_a (vertex v_a), during shrub growth; that is, $d_k^a = \sum_{e_{ij} \in P_{ak}} w_{ij}$, where w_{ij} is the cost of edge e_{ij}.

Every vertex within a shrub except the root vertex has a parent (the previous vertex on the path from root of S_a). Let's suppose vertex v_i is vertex v_j's parent, the following relationship exists: $d_j^a = d_i^a + w_{ij}$, where w_{ij} is the cost of edge e_{ij}.

Definition 5.7: A **Candidate edge** e_{ij} for shrub S_a is an edge that can possibly be chosen by S_a for the next shrub expansion, where vertex v_i is in S_a, vertex v_j is not in S_a, and vertices v_i and v_j do not belong to the same grove.

Definition 5.8: A **Candidate edge set** of shrub S_a is denoted as S_a^c, which contains all candidate edges of S_a for the next step of shrub expansion.

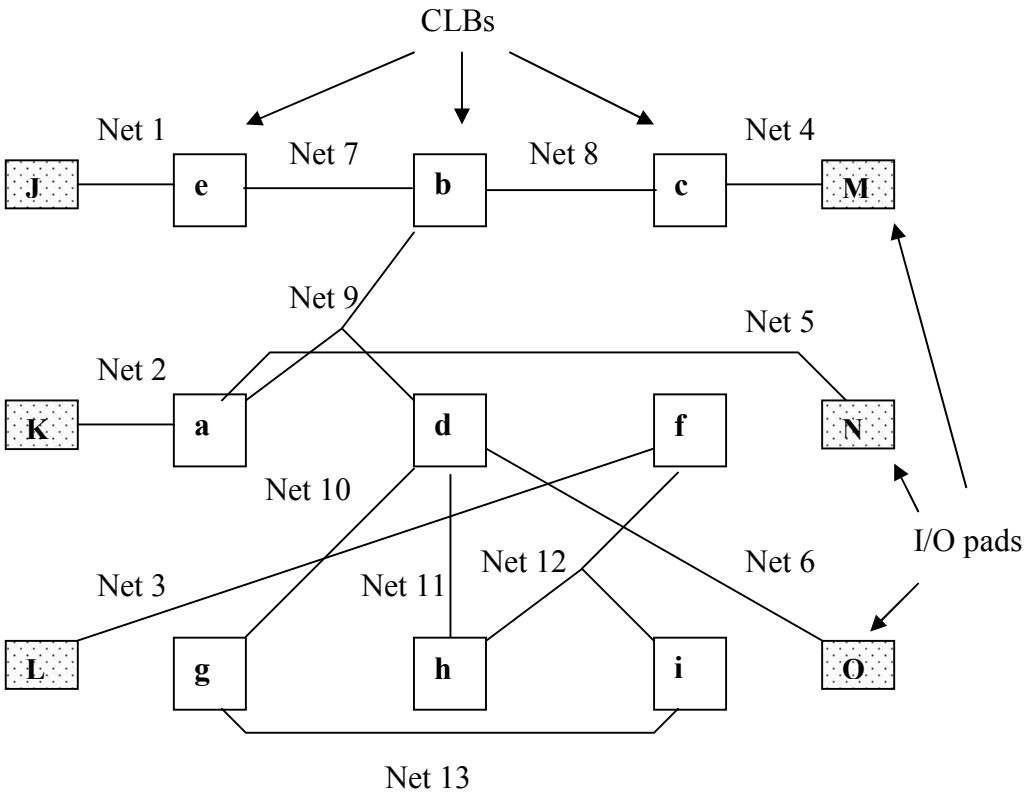
Definition 5.9: An **elected edge** of Shrub S_a is an edge, which has been chosen from the candidate edge set of Shrub S_a for the next step of shrub expansion. There is only one elected edge for each shrub at any step of shrub expansion

Definition 5.10: An **Elected edge set** (denoted as E_E) contains the elected edges of all the shrubs.

5.1.2 Shrubbery Example

To place the I/O pads, we first transform the original circuit into a graph *G = (V, E).* Each logic block corresponds to a vertex $\subseteq V$. I/O pads are treated as special vertices, which we call terminals (*T*). A net with *k* blocks or I/O pads is transformed into a *k*-clique with equally weighted edges; i.e., each edge in the clique has a weight of $\frac{1}{k-1}$. If there

is more than one edge between a pair of vertices, all of these edges are merged into a single edge with a new edge weight equal to the sum of all of the original edge weights. After the graph is transformed into a simple graph, each edge weight is changed to its reciprocal. We use this edge weight as a measure of the connectivity. Figure 5.3 shows an arbitrary circuit, while Fig. 5.4 shows the circuit's corresponding graph representation. (Note: for simplicity, all of the nets contain at most 3 blocks. Therefore, all resulting cliques contain at most 3 edges.)



Net 1: J-e
Net 2: K-a
Net 3: L-f
Net 4: M-c
Net 5: N-a
Net 6: O-d
Net 7: b-e
Net 8: b-c
Net 9: a-b-d
Net 10: d-g
Net 11: d-h
Net 12: f-h-I
Net 13: g-i

Figure 5.3 An arbitrary circuit.

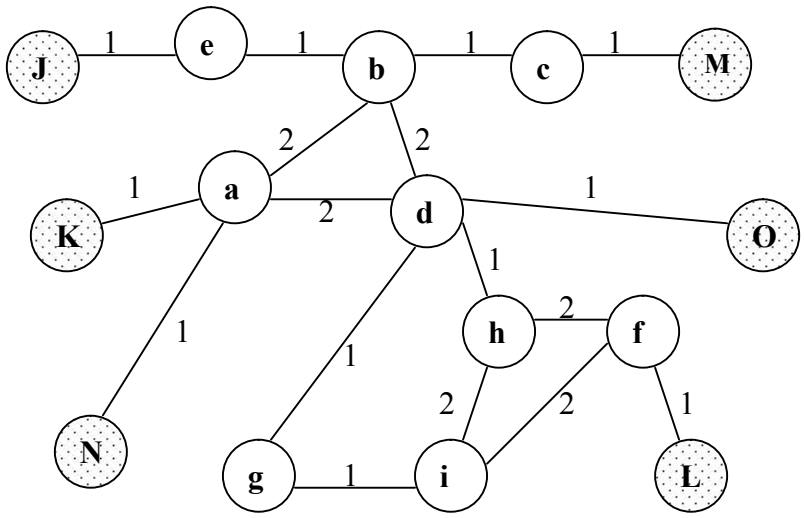


Figure 5.4 The corresponding graph

After the transformation is complete, the Shrubbery algorithm computes the relative order of the I/O pads. It starts by simultaneously growing individual shortest-path trees rooted at every terminal vertex. We refer to these trees as "shrubs". Shrubs grow using a modified version of Dijkstra's shortest-path algorithm [85]. Initially, each shrub consists of a single terminal vertex, its root. Then, shrubs are extended by adding one edge and vertex to one shrub at each step of the algorithm. At every step, each shrub has one nearest adjacent vertex not belonging to the same shrub and having the minimum cost path to its root. The shrub with the (globally) nearest adjacent vertex will grow and expand to include its nearest vertex. That shrub will then determine its next nearest vertex and become a candidate with the remaining shrubs, as Shrubbery selects another edge and vertex. Eventually all the interior vertices will belong to one of the shrubs. Each shrub will then contain only vertices that are at least as close to its terminal (i.e., I/O pad) as to any other terminals. More detailed information of the algorithm can be found in Section 5.1.3.

Figures 5.5 - 5.7 show the procedure of applying Shrubbery on the graph given in Figure 5.4. In these figures, there are six shrubs corresponding to six I/O pads. Each

shrub is shown using dashed lines and grows from its terminal I/O pad "inward" toward other shrubs.

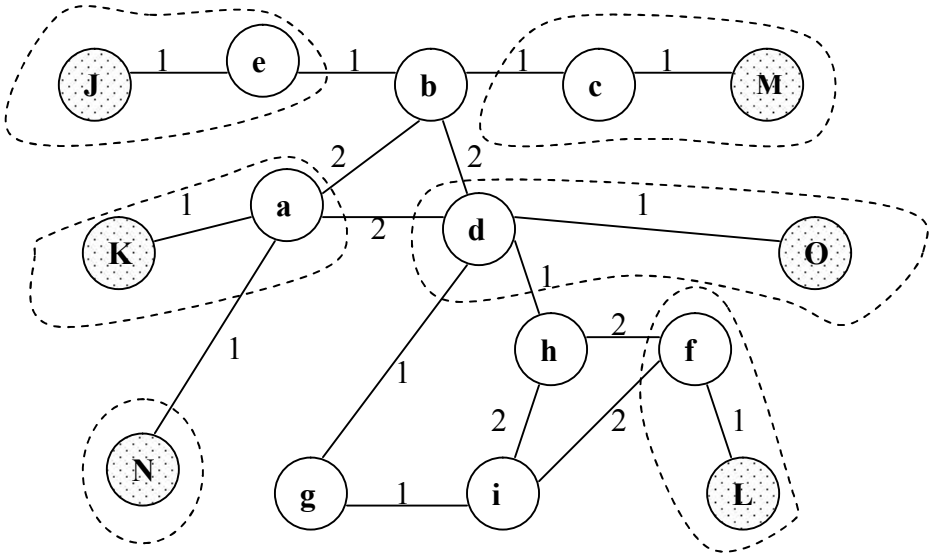


Figure 5.5 The shrubs when distance is 1

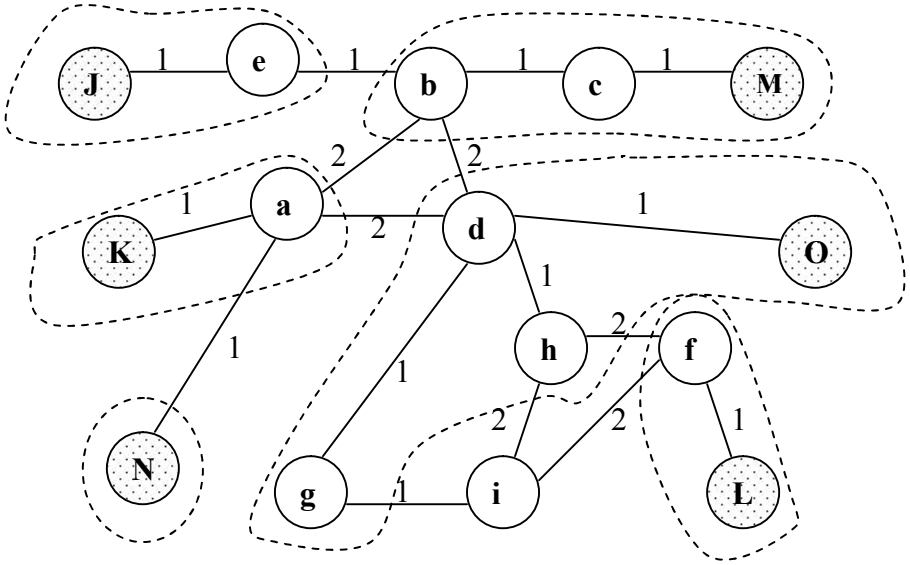


Figure 5.6 The shrubs when distance is 2

At a distance of 1, shrub **J** includes vertex (block) **e**; shrub **K** includes vertex **a**; shrub **L** includes **f**; shrub **M** includes **c**; and shrub **O** includes **d**. Whether vertex **a** joins shrub **K** or shrub **N** is arbitrary (Fig. 5.5). At a distance of 2, shrub **M** adds vertex **b**; shrub **O** includes vertices **g** and **h**. Again, whether vertex **b** joins shrub **M** or shrub **J** is arbitrary (Fig. 5.6). At a distance of 3, shrub **L** includes vertex **I**, and all shrubs meet (Fig. 5.7).

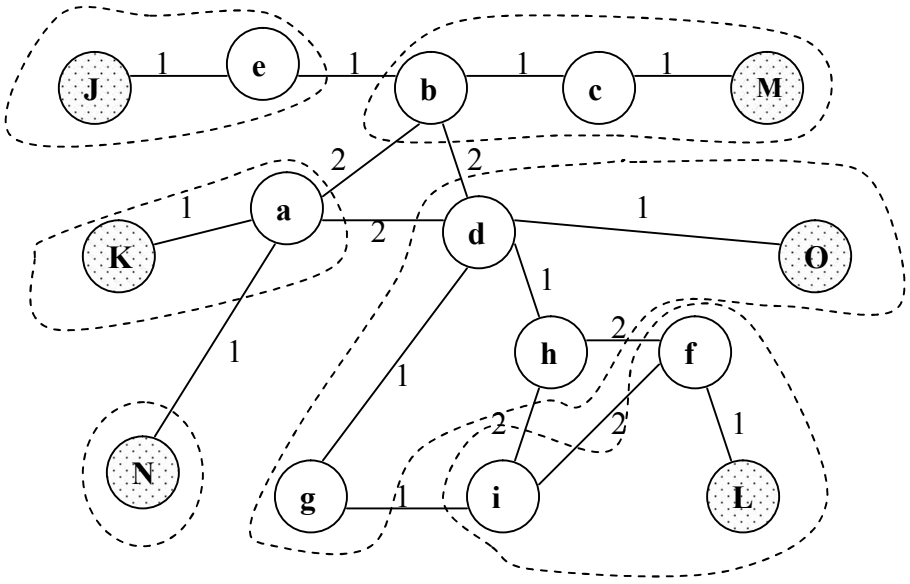


Figure 5.7 The shrubs when distance is 3

Each shrub starts with an *association* tree. During the growth of shrubs, when two shrubs meet the first time, their association trees combine into one tree (more than one shrub can refer to one association tree). For example, in Figure 5.4, shrub *n* and shrub *k* meet first. Their association trees combine to *kn*. Then shrubs *m* and *j* meet, and their trees combine to *jm*. Then *l* and *o* meet, and their trees combine to *lo*. Then *k* and *m* meet. The trees *kn* and *jm* combine to form *jmkn* (if there is more than one way to join two strings, choose the combination with the fewest letters between *m* and *k*). Then *k* and *o*

meet, and their association trees *jmkn* and *lo* combine to *jmknol* (Please note: this time *k* and *o* **cannot** be adjacent to each other. There are two ways to join the two strings: *jmknol* (*lonkmj*) and *jmknlo* (*olnkmj*). There is only one letter (*n*) between *k* and *o* in *jmknol*, while there are two letters (*nl*) between *k* and *o* in *jmknlo*. Therefore, *jmknol* is chosen.) At this moment, all shrubs are attached, and the final tree *jmknol* implies the degree of association between shrubs and thus between their terminals. Since the terminals represent I/O pads, this implies the degree of association between I/O pads and suggests an initial relative (ordering) placement of I/O pads. In the previous example, *j* is adjacent *m*; *m* is adjacent to *k*; and so on. Figure 5.8 gives the final tree. (Note: The I/O pads can now be assigned to actual positions (locations) on the FPGA (see Section 5.1.3). Again, we stress that this assignment may change later after the logic blocks have been placed.)

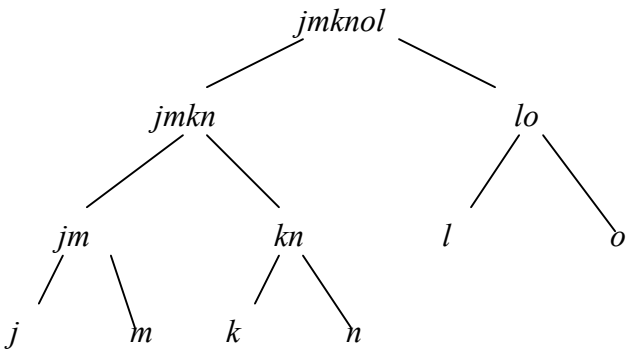


Figure 5.8 The final tree

5.1.3 Shrubbery Algorithm

A formal description of the Shrubbery algorithm is presented in Fig. 5.9. The algorithm begins (line 0) with a search for the edge e_{ij} from the elected edge set E whose weight w_{ij} extends any shrub, say S_a, by the smallest root-distance. Once determined, if the vertex that edge e_{ij} meets (x_j) does not already belong to a shrub, the original shrub (S_a) expands

to include both the new edge and new vertex. Moreover, the unique path from the root of the shrub (x_a) to vertex x_j is recorded (line3), and vertex x_j is made an official member of the shrub S_a (line 4). Observe (line 5) that the root-distance of vertex x_j is simply the root-distance from the root of the shrub to vertex x_i (d_{ai}) plus the weight (w_{ij}) of the new edge.

[0] **do**
[1] Let e_{ij} satisfy $e_{ij} \in E$ and
$\forall T, \min_{i,j}(d_{ai} + w_{ij} : x_i \in S_a, x_j \notin S_a)$
[2] **if** $x_j \notin S_b, \forall b \in T$ add x_j and e_{ij} to s_a
[3] $P_{aj} = P_{ai} \| P_{ij}$
[4] $s(x_j) = a$
[5] $d_{aj} = d_{ai} + w_{ij}$
[6] **else if** $x_j \in S_b$
[7] $G_a^n = G_a^n \cup G_v^n$
[8] $G_a^e = G_a^e \cup G_v^e \cup \{e_{ij}\}$
[9] $H_a^n = H_a^n \cup H_b^n \cup N(P_{ai}) \cup N(P_{bj})$
[10] $H_a^e = H_a^e \cup H_b^e \cup E(P_{ai}) \cup E(P_{bj}) \cup \{e_{ij}\}$
[11] Join two strings including x_i and x_j respectively, in a way that the number of letters between them is minimum.
[12] discard G_b and H_b
[13] **until** G_a contains all *m* terminals
[14] after *m*-1 merges, we get a string representing the relative position of all I/O pads.

Figure 5.9 Shrubbery algorithm

If, however, the vertex "closest" to S_a, x_j, was found to be part of another shrub S_b (line 6), both groves will merge to form a single grove (lines 7 and 8), and their respective hedges will be connected by a new hedge segment. Specifically, e_{ij} will connect P_{ia} and P_{jb} to form a new hedge segment P_{ab}. The new hedge will consist of the union of all of the nodes (line 9) and edges (line 10) involved. Once the new hedge (H_a) and grove (G_a) are formed, the two corresponding strings in which each letter represents the root of a shrub (i.e. a terminal or an I/O pad) join into one string (line 11). If two

letters are adjacent to each other within the new string, the corresponding I/O pads should be placed next to each other. Line 12 just discards G_b and H_b that are no longer useful.

These steps repeat until all terminals (I/O pads) are included in one grove (line 13). At last, we get one string that is formed after *m*-1 merges. This string implies the relative position of all I/O pads. Two I/O pads should be placed beside each other if their corresponding letters are next to each other in the final string. If there are fewer number of I/O pads (*m*) than I/O blocks around the perimeter of the FPGA (*p*), the I/O pads are *evenly* distributed around the perimeter of the FPGA. More specifically, if there are *m* I/O pads that must be pre-assigned locations around the perimeter of an FPGA chip with a perimeter *p*, any two adjacent I/O pads will be placed $d = p / m$ away from each other. The first I/O pad is always placed at the top left corner; then the second I/O pad is placed at a distance *d* away from the first I/O pad, travelling in a clock-wise direction. In general, the *i*th I/O pad is placed at a distance (*i*-1)*d* away from the first I/O pad, again travelling in a clock-wise direction around the perimeter of the FPGA. For example, Figure 5.10 gives the placement of the I/O pads (*jmknol*) shown in Figure 5.8 on a 3 × 3 FPGA chip. In this case, the perimeter *p* is 12, and there are 6 I/O pads, thus the distance *d* between any two adjacent I/O pads is 2. (Note: *d* does not have to be an integer, since we do not require a legal I/O pad placement at this stage.)

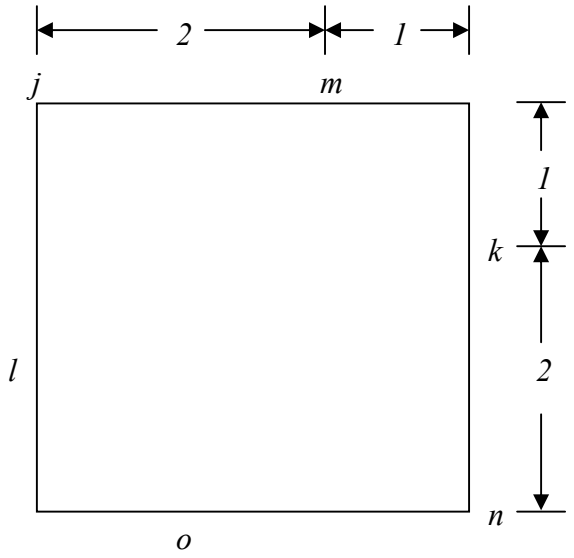


Figure 5.10 Placement of I/O pads shown in Fig 5.8

5.1.4 Implementation and Time Complexity

To achieve an efficient implementation for Shrubbery, all edges and vertices are stored in separate Fibonacci heaps. We also make use of the well-known union-find data structure when determining whether a newly encountered vertex (during shrub growth) belongs to a different grove or simply (another) part of the existing grove.

In Fig. 5.11, the operations in lines 3-9 and 15 take *O(1)* time. Therefore, the time complexity of the algorithm is dominated by the time spent performing the operations in lines 1, 2, and 10-14. Since the time for finding an element with the smallest key value from a heap is *O(log n)*, a single execution of line 2 takes *O(log|V|)* time. Using union-find to perform the operations in lines 11-14, the resulting operations each take *O(log|V|)* time. The most expensive step in the algorithm is the first (line 1), which requires at most |E| iterations. As each iteration of the do-until loop takes *O(log|V|)* time, we can see that shrubbery has a time complexity of *O(|E|log|V|)*.

5.2 Legalizing Solutions using Recursive Bi-partitioning

By pre-assigning I/O pads to temporary locations, we add constants to the equation system $f'(x)=0$ and effectively avoid producing trivial solutions. In this section, we will discuss how to deal with infeasible solutions obtained by solving $f'(x)=0$. Specifically, we will show how to convert an infeasible solution due to block overlap into a feasible solution (placement).

Due to the physical characteristics of the FPGA architecture, the coordinates of the blocks must obey two basic rules:

1. All the coordinates must be integers.
2. Each position (CLB) on the FPGA chip can only be occupied by at most one block.

However, the solutions obtained by solving the equation system $f'(x)=0$ are usually non-integers. Moreover, the positions of the blocks tend to locate in the center of the placement area (the so-called overlap problem). To deal with these problems, a bisection technique [17] is used recursively to divide the blocks into smaller regions until each region only contains one configurable logic block.

At first, the entire FPGA chip is vertically split into two partitions by putting half of the blocks with smaller *x*-coordinates in the left partition and the other half blocks with larger *x*-coordinates in the right one. Then, each of the two partitions is divided horizontally into two sub-partitions by putting half of the blocks with smaller *y*-coordinates in the bottom sub-partition and the half with larger *y*-coordinates into the top partition. This type of bi-partitioning is alternatively performed vertically and horizontally until the FPGA is divided into partitions where each partition contains only one block. Once a partition contains only one block, the *x*- and *y*-coordinates of the block are assigned to the integer values closest to the coordinates of the center of the partition.

Figure 5.11 shows the pseudo-code of the Bi-partitioning algorithm (a partition of an FPGA is a part of the FPGA chip that contains at least one CLB or I/O pad). More information can be found in [17].

5.3 The CG Placement Algorithm

It is time to put all of the parts together to build the whole CG placement algorithm. A pseudo-code description on the algorithm is given below in Fig. 5.12. Initially, Shrubbery is used to pre-place all of the I/O pads in order to prevent CG from producing trivial solutions. The main body of the algorithm consists of two nested loops: the inner loop runs CG for *number_CG* iterations. Due to block overlap, the solution produced is usually infeasible and, therefore, passed to the Bi-partitioning algorithm [17] to be converted into a feasible solution. The outer loop runs both CG and Bi-partitioning

methods, while each iteration reduces the number of CG iterations (*number_CG*) by γ. (Throughout the remainder of this thesis, we set γ to 0.1; that is, after each iteration of the while loop, we reduce the number of iterations that CG subsequently performs by 10 percent. Empirical justification for setting γ equal to 0.1 is given later in Section 5.5.) The exit criterion for terminating the outer loop is when *number_CG* reaches a value less than or equal to 1.

```
if  a partition has more than one CLB{
        if (the x-side length of the partition > y-side length) {
                sort all blocks assigned to the partition according to their x-coordinates;
                divide the partition to two partitions A and B along x-dimension;
                divide all the blocks into two groups:
                        group A has all the smaller x-coordinates;
                        group B has all the larger x-coordinate;
        }

        else {
                sort all blocks assigned to the partition according to their y-coordinates;
                divide the partition to two partitions A and B along y-dimension;
                divide all the blocks into two groups:
                        group A has all the smaller y-coordinates;
                        group B has all the larger y-coordinate;
        }

        if the number of blocks in group A is greater than 0{
                assign blocks in group A to partition A;
                bi-partition partition A;
        }

        if the number of blocks in group B is greater than 0{
                assign blocks in group B to partition B;
                bi-partition partition B;
        }

}
else{
        legalize the x- and y-coordinates of the only block in the partition;
}
```

Figure 5.11: Pseudo-code of bi-partitioning algorithm

```
Use Shrubbery to pre-place I/O pads;          // the positions of I/O pads may
                                              // change later when running CG.

number_CG = max (x-size of FPGA, y-size of FPGA);

while(number_CG > 1)
{
        run CG for number_CG iterations;
        recursively bi-partition the placement solution obtained from CG;
        number_CG = number_CG × (1 - γ)
}
```

Figure 5.12: Entire CG placement algorithm

In practice, the CG does not consider the physical positions of the CLBs and I/O pads on the FPGA. When CG converges, the blocks will move towards the center of the FPGA. The blocks will not be assigned to CLBs and will often overlap on another. If CG were to be run again, the blocks will be focused even further into the center of the FPGA. Therefore, it is crucial to perform the legalization step after each iteration of CG. The legalization step will transform the infeasible placement solution produced by CG into a feasible placement solution. CG can then be re-started from the feasible solution. By running partitioning after CG on each iteration, the algorithm can gradually converge towards a feasible high-quality placement.

5.4 Convergence of CG

In Section 5.3, we reported that the decision was made to reduce the number of iterations CG is performed by 10 percent (i.e., $\gamma = 0.1$) after each iteration of the outer while loop. We now provide some empirical evidence for this decision. Figure 5.13 shows the quality of placements for the 20 MCNC benchmarks when γ is set to 10%, 20%, …, 90%. For each value of γ, the total estimated number of wire segments for all 20 benchmarks is shown. When γ is set to 10%, the estimated total number of wire segments is 942435. As the value of γ increases, the number of wire segments required for successful routing also

increases. When γ reaches 90%, the number of wire segments needed increases by 19.1% (1122351) compared with when γ is 10%.

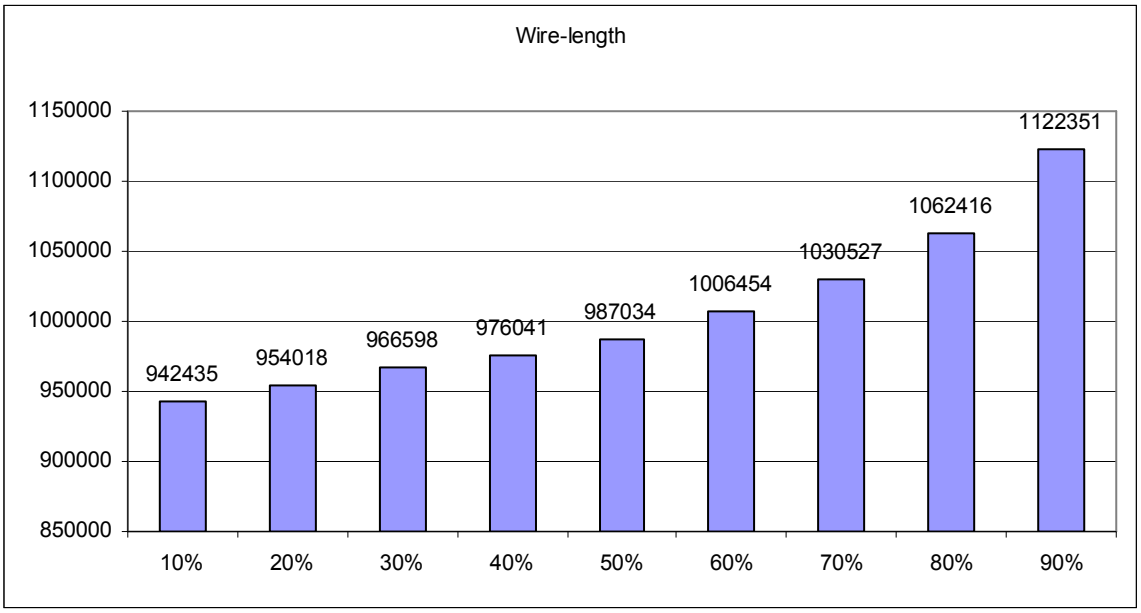


Figure 5.13: Wirelength with different reduction rates of the iteration number

Figure 5.14 gives the total CPU running time for all 20 MCNC benchmarks. For each value of γ, the total CPU running time is shown (in seconds). When γ equals 10%, the total CPU running time is 38.11 seconds. As γ increases, the total CPU running time reduces, as expected. However, when γ is increased to 90%, the total CPU running time reduces to 6.72 seconds.

These experiments show that the parameter γ is an adjustable parameter that allows the user to trade-off runtime versus solution quality (much like the parameter **inner_num** used in VPR [28]). If the user wants better quality solutions, a smaller value of γ should be used. If the user wants a faster placement, a larger value of γ can be used. It should be noted that regardless of the value of γ, the proposed conjugate gradient placement method is very fast. Therefore, throughout the remainder of this thesis we leverage this speed and choose to set γ to 0.1 in an attempt to find high-quality solutions.

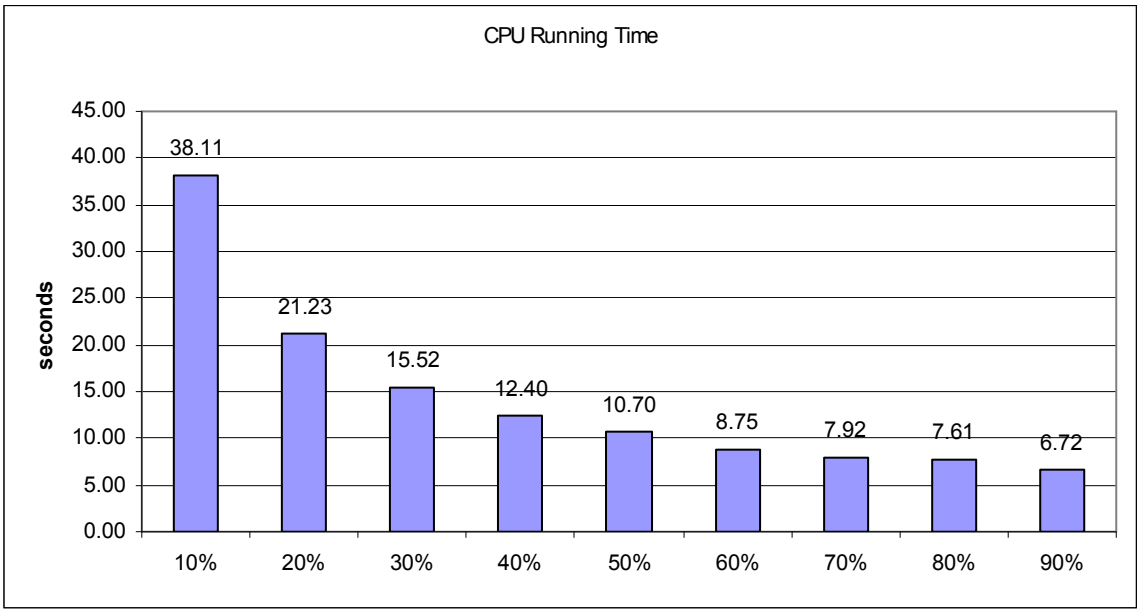


Figure 5.14: CPU running time with different reduce rates of the iteration number

5.5 Experimental Results

In this section, we first run a few tests to determine the effectiveness of Shrubbery. Then, we compare CG placement with VPR with respect to runtime, critical-path delay, and wirelength following routing.

5.5.1 Shrubbery versus Random Pre-Placement

To determine the overall effectiveness of Shrubbery, we ran the conjugate-gradient placement method described in the previous section two different ways, and then compared the results head-to-head. The first approach involved running Shrubbery once to pre-place the I/O pads followed by CG placement (as shown in Fig. 5.12). The second approach involved randomly pre-placing the I/O block around the perimeter of the FPGA

followed by CG placement. The random pre-placement approach initializes the *x*- and *y*-coordinates of each I/O block with random values. In particular, assuming that the size of the FPGA is *nx* x *ny*, the *x*-coordinate of an I/O block is set to a random integer between 0 and *nx*, while the *y*-coordinate of the block is set to a random integer between 0 and *ny*.

Table 5.1 shows the comparison between Shrubbery pre-placement and random pre-placement. The first column identifies the benchmark. The second column contains the Star+ wirelength estimates of the placement solutions obtained after using Shrubbery to perform pre-placement. Columns 3 and 4 show the CPU running times (in seconds) of Shrubbery (pre-placement) and the CG placement algorithm including Shrubbery, respectively. Column 5 gives the percentage of total CPU time required by Shrubbery to perform pre-placement. As Shrubbery is a deterministic algorithm it need only be run once. When performing random placement, however, ten separate runs were performed. Column 6 indicates the *average* Star+ wirelength estimates of the placement solutions obtained when starting with 10 random pre-placements, while Column 7 indicates the total CPU running time (in seconds) required to place each benchmark ten times.

With regards to estimated wirelength, the results in Table 5.1 show that Shrubbery never finds a "bad" placement; that is, it never finds a placement appreciably worse than that found using a random pre-placement strategy. Moreover, a closer inspection of Table 5.1 reveals that Shrubbery pre-placement outperformed random pre-placement on 14 out of 20 benchmarks (70%). In the best case (s38417) Shrubbery outperformed the average random pre-placement strategy by almost 10 percent, while the performance of Shrubbery over all 20 benchmarks is 1.2% better than that of the random pre-placement strategy.

Shrubbery's main advantage over the random pre-placement strategy, however, is its speed. Recall that Shrubbery is deterministic. Thus, it needs to be run only once followed by conjugate-gradient. According to Column 5, the time required to perform conjugate gradient accounts for approximately 91% to 97% of the total run time. Now, in the case of the random pre-placement strategy, individual random placements may be

quite poor requiring a "pool" of solutions (e.g., 10) to be created and evaluated. Note that each random trial requires conjugate-gradient to be run. If we assume that each random pre-placement is instantaneous, the time required to run CG multiple times versus running CG once (when using Shrubbery) is significant. For example, it can be clearly seen from Table 5.1 that running Shrubbery and conjugate gradient once is approximately 9 times faster than running conjugate gradient ten times to generate a pool of 10 (random) pre-placements.

Table 5.1: Shrubbery pre-placement vs. random pre-placement.

	Shrubbery				Random	
	Wire-length	**Pre-placement time**	**Total CPU time**	**Percentage**	**Wire-length**	**CPU time**
Tseng	11288	0.03125	0.34375	9.09%	11706	3.125
Ex5p	18875	0.03125	0.34375	9.09%	19121	3.125
Apex4	21484	0.03125	0.4375	7.14%	21500	4.0625
Misex3	22033	0.03125	0.5	6.25%	22140	4.6875
Diffeq	17339	0.03125	0.5625	5.56%	18025	5.3125
alu4	20990	0.03125	0.5625	5.56%	21437	5.3125
Seq	28485	0.046875	0.6875	6.82%	29021	6.40625
Apex2	31771	0.03125	0.765625	4.08%	32010	7.34375
s298	22161	0.03125	0.765625	4.08%	21696	7.34375
Dsip	21929	0.03125	0.828125	3.77%	23304	7.96875
Bigkey	25264	0.046875	0.953125	4.92%	26301	9.0625
Frisc	61315	0.078125	2.078125	3.76%	60168	20
Elliptic	53487	0.078125	2.0625	3.79%	56499	19.84375
Spla	73681	0.078125	2.171875	3.60%	71793	20.9375
Des	32974	0.046875	1.109375	4.23%	32332	10.625
ex1010	74126	0.09375	2.921875	3.21%	73943	28.28125
Pdc	103418	0.109375	3.03125	3.61%	103253	29.21875
S38417	72645	0.171875	4.859375	3.54%	78680	46.875
S38584.1	72761	0.140625	4.90625	2.87%	73785	47.65625
Clma	156409	0.25	8.21875	3.04%	157064	79.6875
Total	**942435**	**1.421875**	**38.10938**	**3.73%**	**953772**	**366.875**

5.5.2 CG versus VPR

We now compare CG to VPR [28] – the state-of-the-art academic place and route tool. Table 5.2 compares the running time (in seconds) of CG placement with that of VPR with

`inner_num` set to 1 and 10, respectively. Both placement tools were tested using all 20 MCNC benchmarks. The last row of the table shows the total running time to place all 20 benchmarks. The total runtime for CG is 264.5 seconds compared with 169.5 seconds for VPR when run in its fastest mode with inner_num =1. This means that CG is 56% slower than VPR when run in its fastest mode of operation. However, CG is more than 5 times faster than VPR when VPR is run with inner_num =10.

Table 5.2: Running time of CG and VPR in Seconds

	CG	VPR	
		inner_num=1	inner_num=10
Tseng	1.41	1.86	18.6
Ex5p	1.31	1.83	18.3
Apex4	1.72	2.2	22
Misex3	2.06	2.53	25.3
Diffeq	2.42	2.95	29.5
alu4	2.41	2.66	26.6
Seq	3.08	3.45	34.5
Apex2	3.53	3.86	38.6
s298	3.53	3.45	34.5
Dsip	4.78	2.8	28
bigkey	5.25	3.83	38.3
Frisc	11.52	9.59	95.9
Elliptic	12.42	10.11	101.1
Spla	12.34	9.84	98.4
Des	7.48	3.47	34.7
ex1010	18.11	13.55	135.5
Pdc	18.63	13.24	132.4
S38417	38.17	22.69	226.9
S38584.1	41.02	22.3	223
Clma	73.33	33.3	333
Total	**264.52**	**169.51**	**1695.1**

We now turn our attention to solution quality as measured by critical-path delay. Recall that CG and VPR use two different net models for estimating wirelength: Star+ and HPWL, respectively. Consequently, any comparison between CG and VPR, with regards to solution quality, must be performed *after* routing. When using VPR's router, the router is configured to perform timing-driven routing which attempts to improve circuit speed by reducing critical-path delay.

Table 5.3 compares the critical-path delays found when using CG and VPR with `inner_num=1` and `inner_num=10`, respectively. (Note: the results presented for VPR are the average of 10 independent runs.) The results show that when VPR is run with inner_num=1 (fastest option), CG finds lower critical-path delays for 11 of the 20 cases. The average reduction for these 11 cases is 13 percent. In the 9 cases that CG fails to find a lower critical-path delay, the average increase in delay is 6.6 percent. Overall, CG finds a 2 percent reduction in critical-path delay compared with VPR.

Table 5.3: Critical-path delays (CG vs. VPR)

	CW	CG	VPR	
			inner_num=1	inner_num=10
Alu4	12	99.01	120.331	113.6717
Apex2	13	108.37	128.77	125.1346
Apex4	15	128.23	127.922	122.6053
bigkey	9	60.77	100.935	100.0536
Clma	16	285.18	264.999	252.9958
Des	12	121.15	123.01	136.5118
Diffeq	9	112.16	106.112	90.33062
Dsip	9	75.07	91.0482	93.37907
elliptic	13	259.96	257.387	206.6148
Ex1010	14	238.75	205.552	202.9452
Ex5p	16	124.98	116.071	125.2613
Frisc	14	249.61	227.362	189.0848
Misex3	14	99.92	108.431	105.6976
Pdc	21	222.77	254.422	217.5874
S298	9	234.20	240.983	203.189
S38417	10	213.06	196.969	163.1709
S38584.1	10	115.30	123.888	119.709
Seq	14	104.22	123.035	118.0495
Spla	16	213.07	205.085	188.0682
Tseng	8	78.01	81.7572	75.83124
Total		**3144**	**3204**	**2950**

Not surprisingly, when VPR is run with `inner_num=10`, VPR performs much better with respect to critical-path delay. CG finds a better solution for 9 of the 20 cases. The overall average increase in critical-path delay is 6.6%.

We now turn our attention to wirelength. Table 5.4 compares the placements produced by CG and VPR (with `inner_num=1` and `inner_num=10`, respectively)

with respect to wirelength following routing. For this comparison, the router uses a breadth-first strategy. The reason for using a breadth-first strategy is because a breadth-first routing strategy seeks to find a successful routing by minimizing the number of required wire segments to make all connections.

Table 5.4 Wirelength (CG vs. VPR)

	CW	CG	VPR	
			inner_num=1	inner_num=10
Alu4	11	20854	22038	21016
Apex2	13	32629	32546	30638
Apex4	14	23082	22865	21848
bigkey	8	25596	22396	18505
Clma	15	152826	142509	133592
Des	12	34297	29161	24758
Diffeq	9	16861	16263	14676
Dsip	9	25547	17171	14582
elliptic	13	49536	53811	45912
Ex1010	13	75553	72613	70864
Ex5p	15	21142	19924	18648
Frisc	14	60668	59957	55274
Misex3	13	24255	22700	21871
Pdc	19	106503	104298	99046
S298	9	21388	22703	21346
S38417	10	72340	66586	61764
S38584.1	10	64958	63515	57099
Seq	13	30061	29611	28059
Spla	16	72155	71194	67362
Tseng	8	9880	10420	9423
Total		**940131**	**902282**	**836281**

In Table 5.4, Column 1 identifies the benchmark by name. Column 2 indicates the channel width used by the router. The third column shows the actual wirelength required when using CG. The fourth and fifth columns show the total wirelength required by VPR (with `inner_num`=1 and `inner_num`=10, respectively). All data for VPR is the average of 10 independent runs.

The results show that when VPR is run with inner_num=1, CG uses 4.2 percent more wirelength on average. When `inner_num`=10, the overall improvement of VPR over CG in terms of wirelength is 12 percent.

In short, CG is more than 5 times faster than VPR (`inner_num`=10) but 56% slower than VPR (`inner_num`=1). When VPR is run with `inner_num`=1, CG gets solutions with 2% less delay while using 4.2% more wire segments. When VPR is run with `inner_num`=10, CG gets solutions with 6.6% more delay and uses 12% more wire segments.

5.6 Summary

In this chapter, we developed a pre-placement algorithm, called Shrubbery, to pre-place I/O blocks. This guarantees that the CG placement algorithm produces non-trivial placements. We demonstrated that Shrubbery is able to outperform a random pre-placement strategy that seeks to find the best (initial) placement by generating a pool of random placements, both with respect to solution quality and runtime. Unlike the random pre-placement strategy, Shrubbery only requires a single application of CG, which is much faster than the multiple applications of CG that are required to create a pool of random pre-placements. Most importantly, we were able to show, both theoretically and empirically, that the running time of Shrubbery is extremely small. Finally, to avoid illegal placements we employed an existing bi-partitioning algorithm [17] that legalizes the solutions obtained from CG placement.

From the experimental results, we conclude that CG is competitive with VPR in its fast mode (`inner_num`=1), but produces slightly lower quality solutions than VPR when `inner_num`=10. Also, by reducing the runtime of each iteration from $O(n^2)$ to $O(n)$, we have improved the speed of CG significantly and made it about 5 times faster than VPR when `inner_num`=10. The speed of CG can be further improved by 80% if we increase the value of γ from 0.1 to 0.2 at the cost of only 1.2% increase in wirelength estimate. However, as an effective "linear search" algorithm CG converges much more slowly when used to solve a nonlinear equation system. The convergence of CG depends

on how close the objective function approximates "quadratic". With a constant quadratic objective function for n variables and an exact line search, CG will converge in n or fewer iterations. However, as our objective function is based on a near-linear model, CG loses conjugacy quickly (This is a typical issue when using CG for solving nonlinear equation systems [32].)

In the next Chapter, we present another analytical method based on *Successive Over-Relaxation* (*SOR*), which does not require the objective function to be "quadratic" and actually converges much faster than the CG method when used to solve the placement problem based on Star+.

Chapter 6

Successive Over-Relaxation Placement

In Chapter 4, a placement method based on Conjugate Gradient was introduced. Conjugate gradient is classified as an *iterative* method. In general, an iterative method starts with an initial solution (or guess). It then incorporates the solution into a recurrence formula from which another approximate solution is generated. This process repeats until a final solution is found. Ideally, the sequence of solutions produced should eventually converge to the exact solution; that is, a placement with minimum total wire length should be found. There is, however, an important caveat that must be considered. In order to converge to an exact solution, the sequence of approximate solutions generated should increasingly resemble the exact solution. When minimizing quadratic functions, an exact solution can be found in at most n iterations (Section 4.1). However, for non-linear problems (like the one considered in this thesis), the search directions on each iteration can quickly lose conjugacy [32]. As a result, *slower* progress towards the exact solution is made.

In this chapter, we present a second analytic placement algorithm based on Successive Over-Relaxation (SOR) [33]. Like conjugate gradient, SOR is also an iterative method. However, unlike conjugate gradient, SOR does not require search directions to be conjugate to each other. As a result, SOR can often converge to a final solution extremely quickly.

The remainder of this chapter is organized as follows. Section 6.1 provides all of the necessary background for understanding the SOR method. In Section 6.2 we show how SOR can be used to implement a placement algorithm based on the Star+ model presented in Chapter 3. Section 6.3 presents ordering heuristics and relaxation techniques for improving the performance of SOR placement. In Section 6.4, VPR [28], SOR and CG are applied to the 20 MCNC benchmarks and compared with respect to solution quality and run time. In Section 6.5, we compare the convergence of SOR and VPR. In Section 6.6, we present a hybrid approach by combining SOR and VPR. Finally, in Section 6.7 we provide a summary of the contributions of this Chapter.

6.1 Background

Successive Over-Relaxation is a numerical method for improving the convergence speed of the Gauss-Seidel method [33] for solving systems of linear equations. The Gauss-Seidel method, in turn, is an improved version of the Jacobi method [33] for solving systems of linear equations. Therefore, to prepare the reader for the Successive Over-relaxation method, we begin with a brief overview of the Jacobi and Gauss-Seidel methods.

6.1.1 Jacobi Method

The Jacobi method is an algorithm for solving linear equation systems of the form $Ax = b$. The algorithm is always guaranteed to converge when the coefficient matrix A is

diagonally dominant. (A matrix is said to be diagonally dominant if for every row of the matrix the magnitude of the diagonal element is greater than the magnitude of all other (non-diagonal) entries in that row.) The matrix A can be looked at as the sum of three matrices: $A = D + (L + U)$, where D, L and U represent the diagonal, lower triangular and upper triangular parts of A. Using simple substitution, the original equations system can be re-written as follows:

$$Dx + (L+U)x = b$$
$$Dx = b - (L+U)x$$
$$x = D^{-1}b - D^{-1}(L+U)x \qquad \text{(Equation 6.1)}$$

Note that as D is diagonal, therefore it is easy to invert. Equation 6.1 can now be converted into an iterative search method as shown below:

$$x^{(k+1)} = D^{-1}b - D^{-1}(L+U)x^{(k)}$$

where k is the iteration count. When implementing the Jacobi iteration, an element-based formula is used:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}b_i - \frac{1}{a_{ii}}\sum_{j \neq i} a_{ij}x_j^{(k)}, i = 1,2,\Lambda\ ,n. \qquad \text{(Equation 6.2)}$$

Notice that to compute $x_i^{(k+1)}$ the values for $x_i^{(k)}$ must be retained from one iteration to the next. A summary of the Jacobi method is provided below in Fig. 6.1.

6.1.2 Gauss-Seidel Method

The Gauss-Seidel method is an improved version of the Jacobi method. The main difference between the two methods is that unlike the Jacobi iteration that retains the

updated values until the next iteration, the Gauss-Seidel iteration uses the newest values immediately. More specifically, the Jacobi iteration (Equation 6.2) can be rewritten as:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} b_i - \frac{1}{a_{ii}} \sum_{j<i} a_{ij} x_j^{(k)} - \frac{1}{a_{ii}} \sum_{j>i} a_{ij} x_j^{(k)}, i = 1,2,\Lambda\ ,n.$$

```
for k = 1 step 1 until convergence
{
        for i = 1 to n
        {
                s = 0
                for j = 1 to n
                {
                        if ( j != i ) s = s + a_ij x_j^(k-1)
                }
                x_i^(k) = (b_i - s) / a_ii
        }
}
```

Figure 6.1: Jacobi method

Notice that once we calculate $x_i^{(k+1)}$, we have calculated all $x_j^{(k+1)}$s where $j < i$ (assuming we calculate all the x_is in order starting from x_1 to x_n in each iteration). Therefore, we can use $\frac{1}{a_{ii}} \sum_{j<i} a_{ij} x_j^{(k+1)}$ instead of $\frac{1}{a_{ii}} \sum_{j<i} a_{ij} x_j^{(k)}$ in above equation, and speed up the convergence. An element-based formula of Gauss-Seidel can be expressed as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} b_i - \frac{1}{a_{ii}} \sum_{j<i} a_{ij} x_j^{(k+1)} - \frac{1}{a_{ii}} \sum_{j>i} a_{ij} x_j^{(k)}, i = 1,2,\Lambda\ ,n. \qquad \text{(Equation 6.3)}$$

The Gauss-Seidel method can also be expressed in matrix form. As with the Jacobi method, the matrix A can be expressed as the sum of three matrices: $A = D + L + U$, where D, L and U denote the diagonal, strictly lower triangular, and strictly upper triangular parts of A, respectively. Through simple substitution, this leads to:

$$(D + L)x + Ux = b$$

$$(D + L)x = b - Ux$$

$$x = (D + L)^{-1}(b - Ux)$$

Therefore, the matrix form of Gauss-Seidel can be expressed as:

$$x^{(k+1)} = (D + L)^{-1}(b - Ux^{(k)})$$

Note that Gauss-Seidel is guaranteed to converge if the matrix A is either diagonally dominant or symmetric and positive definite. Figure 6.2 gives the pseudo code of the Gauss-Seidel method.

```
for k = 1 step 1 until convergence
{
        for i = 1 to n
        {
                s = 0;
                for j = 1 to n
                {
                        if ( j != i ) s = s + a_ij x_j
                }
                x_i = (b_i - s) / a_ii
        }
}
```

Figure 6.2: Gauss-Seidel method

Although Equation 6.3 looks more complicated than Equation 6.2, the implementation of Gauss-Seidel is actually easier than the Jacobi method. Since the computation of $x_i^{(k+1)}$ uses only the elements of $x^{(k+1)}$ that have already been computed and the elements of $x^{(k)}$ that have yet to be advanced to iteration k+1, there is no need to store both vectors $x^{(k+1)}$

and $x^{(k)}$ at the same time. The computation can be done in place by replacing $x^{(k)}$ with $x^{(k+1)}$.

6.1.3 Successive Over-Relaxation

Having described both the Jacobi and Gauss-Seidel methods, we now turn our attention to the Successive Over-Relaxation method (which we use throughout the remainder of this Chapter). Successive Over-Relaxation was devised by Young and Frankel in 1950 [34] as a technique to speed up convergence of the Gauss-Seidel method. Given a linear equation system of the form $Ax = b$, we let $A = D + L + U$, where D, L and U denote the diagonal, strictly lower triangular, and strictly upper triangular parts of A. The Successive Over-Relaxation iteration is then defined by the following recurrence relation:

$$x^{(k+1)} = (D + \omega L)^{-1}\left(\omega b + (D - \omega D - \omega U)x^{(k)}\right)$$

where ω is a relaxation factor. If matrix A is symmetric and positive-definite, Successive Over-Relaxation iteration always converges when $0 < \omega < 2$. When ω =1, SOR iteration reduces to Gauss-Seidel method.

Like the Gauss-Seidel method, the computation of SOR can also be performed in place. The actual implementation of SOR uses the following element-based iteration formula:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k)} - \sum_{j>i} a_{ij}x_j^{(k)}\right), i = 1,2,\Lambda\ ,n. \quad \text{(Equation 6.3)}$$

Figure 6.3 gives the pseudo code of the SOR method.

6.2 SOR Placement

In this section, we explain how the SOR method can be used to implement a placement algorithm based on the *Star+* model presented in Chapter 3. To implement SOR placement, it is necessary to begin by finding the recurrence relation between $x_i^{(k+1)}$ and $x_i^{(k)}$. When $f(x)$ is positive definite, we can minimize $f(x)$ by solving $f'(x)=0$. Recall from Section 4.2.2 that $f'(x)$ is a vector that points in the direction of greatest increase of *f(x)* at a given point $x=(x_1,x_2,...x_n)^T$ and is defined as follows:

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \text{M} \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix}.$$

```
for k = 1 step 1 until convergence
{
        for i = 1 to n
        {
                s = 0;
                for j = 1 to n
                {
                        if ( j != i ) s = s + a_ij x_j
                }
                x_i = (1 - ω) x_i + (ω / a_ii)(b_i - s)
        }
}
```

Figure 6.3: Successive Over-Relaxation method

In Equation 4.12 (see Section 4.2.2) it was shown that for the Star+ model, the partial derivative of *f(x)* with respect to x_j is $\frac{\partial}{\partial x_j} f(x) = \alpha \sum_{\forall l: j \in Net_l} \frac{x_j - x_{cl}}{S_l}$. By making $\frac{\partial}{\partial x_j} f(x) = 0$, we obtain the equation:

$$\alpha \sum_{\forall l: j \in Net_l} \frac{x_j - x_{cl}}{S_l} = 0$$

$$\sum_{\forall l: j \in Net_l} \frac{x_j}{S_l} - \sum_{\forall l: j \in Net_l} \frac{x_{cl}}{S_l} = 0$$

$$\sum_{\forall l: j \in Net_l} \frac{x_j}{S_l} - \sum_{\forall l: j \in Net_l} \frac{x_{cl}}{S_l} = 0$$

$$x_j \sum_{\forall l: j \in Net_l} \frac{1}{S_l} = \sum_{\forall l: j \in Net_l} \frac{x_{cl}}{S_l}$$

$$x_j = \frac{1}{\sum_{\forall l: j \in Net_l} 1 / S_l} \sum_{\forall l: j \in Net_l} \frac{x_{cl}}{S_l}$$

In the previous equation, $x_{cl} = \frac{1}{k_l} \sum_{\forall i \in Net_l} x_i$ and $S_l = \sqrt{\sum_{\forall i \in Net_l} x_i^2 - k_l x_{cl}^2 + 1}$. Now, by putting the iteration number into these equations we obtain the Jacobi iteration for the placement problem:

$$x_{cl}^{(k)} = \frac{1}{k_l} \sum_{\forall i \in Net_l} x_i^{(k)}$$

$$S_l^{(k)} = \sqrt{\sum_{\forall i \in Net_l} (x_i^{(k)})^2 - k_l (x_{cl}^{(k)})^2 + 1}$$

$$x_j^{(k+1)} = \frac{1}{\sum_{\forall l: j \in Net_l} 1 / S_l^{(k)}} \sum_{\forall l: j \in Net_l} \frac{x_{cl}^{(k)}}{S_l^{(k)}}$$

To implement the Gauss-Seidel iteration, we need to update x_{cl} and S_l immediately after x_j moves from $x_j^{(k)}$ to $x_j^{(k+1)}$. Fortunately, both x_{cl} and S_l can be updated in a constant time (Section 3.2). This feature makes it possible to build a time-efficient Gauss-Seidel method for FPGA placement problem based on the Star+ model. For the sake of

simplicity, we introduce two new variables U_l and V_l, and let $U_l = \sum_{\forall i \in Net_l} x_i^{\,2}$ and $V_l = \sum_{\forall i \in Net_l} x_i$. The Gauss-Seidel iteration for the placement problem is defined as:

$$U_l = \sum_{\forall i \in Net_l} (x_i^{\,(0)})^2$$

$$V_l = \sum_{\forall i \in Net_l} x_i^{\,(0)}$$

$$x_{cl} = \frac{1}{k_l} V_l$$

$$S_l = \sqrt{U_l - k_l x_{cl}^{\,2} + 1}$$

$$x_j^{\,(k+1)} = \frac{1}{\sum_{\forall l: j \in Net_l} 1/S_l} \sum_{\forall l: j \in Net_l} \frac{x_{cl}}{S_l} \qquad \text{(Equation 6.4)}$$

$$U_l = U_l + (x_j^{(k+1)})^2 - (x_j^{(k)})^2$$

$$V_l = V_l + x_j^{(k+1)} - x_j^{(k)}$$

To advance to the SOR iteration from the Gauss-Seidel iteration, we introduce a relaxation factor ω into Equation 6.4. The SOR iteration for placement is summarized as follows:

$$U_l = \sum_{\forall i \in Net_l} (x_i^{\,(0)})^2$$

$$V_l = \sum_{\forall i \in Net_l} x_i^{\,(0)}$$

$$x_{cl} = \frac{1}{k_l} V_l$$

$$S_l = \sqrt{U_l - k_l x_{cl}^{\,2} + 1}$$

$$x_j^{\,(k+1)} = (1-\omega) x_j^{\,(k)} + \frac{\omega}{\sum_{\forall l: j \in Net_l} 1/S_l} \sum_{\forall l: j \in Net_l} \frac{x_{cl}}{S_l}$$

$$U_l = U_l + (x_j^{(k+1)})^2 - (x_j^{(k)})^2$$

$$V_l = V_l + x_j^{(k+1)} - x_j^{(k)}$$

6.3 Improving SOR for FPGA Placement

In practice, the performance of SOR may be affected by the order in which the x_i variables are calculated. This is illustrated in the following example.

Assume a simple equation system with only three variables and three equations:

$$x_1 - \frac{1}{2}x_2 = 0$$
$$x_2 - \frac{1}{2}x_1 - \frac{1}{2}x_3 = 0$$
$$x_3 - \frac{1}{2}x_2 = 2$$

Prepare to calculate the first SOR iteration (assuming, for the sake of simplicity, ω=1):

$$x_1 = \frac{1}{2}x_2$$
$$x_2 = \frac{1}{2}x_1 + \frac{1}{2}x_3$$
$$x_3 = \frac{1}{2}x_2 + 2$$

Now suppose that we start from (initial guess) $x_1^{(0)} = 0, x_2^{(0)} = 0, x_3^{(0)} = 0$, and then perform several SOR iterations to solve the equation system. We have:

$$x_1^{(1)} = 0, x_2^{(1)} = 0, x_3^{(1)} = 2$$
$$x_1^{(2)} = 0, x_2^{(2)} = 1, x_3^{(2)} = 2.5$$
$$x_1^{(3)} = 0.5, x_2^{(3)} = 1.5, x_3^{(3)} = 2.75$$

M

However, if we change the order of calculating x_1, x_2 and x_3 as shown below:

$$x_3 = \frac{1}{2}x_2 + 2$$
$$x_2 = \frac{1}{2}x_1 + \frac{1}{2}x_3$$
$$x_1 = \frac{1}{2}x_2$$

And assuming we still start from (with the same initial guess) $x_1^{(0)} = 0, x_2^{(0)} = 0, x_3^{(0)} = 0$, and perform several SOR iterations, we get:

$$x_3^{(1)} = 2, x_2^{(1)} = 1, x_1^{(1)} = 0.5$$
$$x_3^{(2)} = 2.5, x_2^{(2)} = 1.5, x_1^{(2)} = 0.75$$
$$x_3^{(3)} = 2.75, x_2^{(3)} = 1.75, x_1^{(3)} = 0.875$$

We can now see that the latter sequence results in faster convergence than the former sequence. In fact, for a given starting point the difference can be as big as *n-1* iterations, where *n* is the number of blocks. The time required to perform *n-1* iterations of SOR is $O(n^2)$. In order to try and speed up the rate of convergence, we employ a novel heuristic to pre-determine (sort) the sequence (order) in which the x_i variables should be processed. This heuristic is described next.

6.3.1 Ordering Heuristic

Recall that prior to performing placement, the Shrubbery algorithm (described in Chapter 5) is used pre-place all of the I/O pads. (It is necessary to perform this pre-placement to avoid trivial solutions being found (i.e., f(x) =0).) The proposed ordering heuristic is based on the idea that the position of blocks that have more connectivity with I/O pads should be determined ahead of blocks that have less connectivity with I/O pads.

To determine (sort) the sequence for calculating each block's x-coordinates x_i, we use a modified version of Dijkstra's algorithm. First, a graph with *n* vertices is built in a

way similar to that used when pre-placing I/O pads (see Chapter 5). Each vertex represents a block. Each edge represents a connection with an edge weight, which is the "closeness" of the connection. The closeness is computed as the reciprocal of the cardinality of the corresponding net.

For example, using the graph in Fig 5.4 as a starting point, we obtain the "closeness" graph shown in Fig 6.4, which has the same vertices and edges, but each new edge weight is now the reciprocal of the original edge weight (also the reciprocal the cardinality of the corresponding net). All blocks that have been pre-placed using Shrubbery (see Chapter 5) are put into the *source* pool. In this case, I/O pads *J*, *K*, *L*, *M*, *N* and *O* are in the source pool. Any other vertex, which is not in the source pool but has an edge (or edges) connecting to any vertex (or vertices) in the source pool, is associated to the source with a "closeness" that equals the sum of all the edge weights. For example, in Fig 5.4, block *e* has only one edge connecting it to *J* (a vertex in the source pool); therefore the "closeness" of *e* is the edge weight, 1; block *a* has two edges connecting to *K* and *N*, respectively, and hence its "closeness" is 2 (the sum of the two edge weights). For those blocks that have no edge connecting to the source pool, their "closeness" is 0 and will not be considered (sorted) at this time. The blocks with nonzero closeness are sorted in descending order of their closeness: *a(2)*, *e(1)*, *c(1)*, *d(1)*, *f(1)*, and are also put into the source pool. (Note: the numbers inside the brackets denotes the closeness of a block. Blocks that have the same closeness value are randomly ordered.) All of the blocks that are sorted in order are put into the source pool, and closeness of vertices not in the source pool is re-calculated (see Fig 6.5). This time, the closeness of all the vertices that have edges connecting to the source pool are: *b(3)*, *h(1.5)*, *g(1)*, *i(0.5)*. Finally, by connecting sequence *a*, *e*, *c*, *d*, *f* and *b*, *h*, *g*, *i*, we get the entire sequence *a*, *e*, *c*, *d*, *f*, *b*, *h*, *g*, *i*. This sequence defines the order of calculating the coordinates of the blocks.

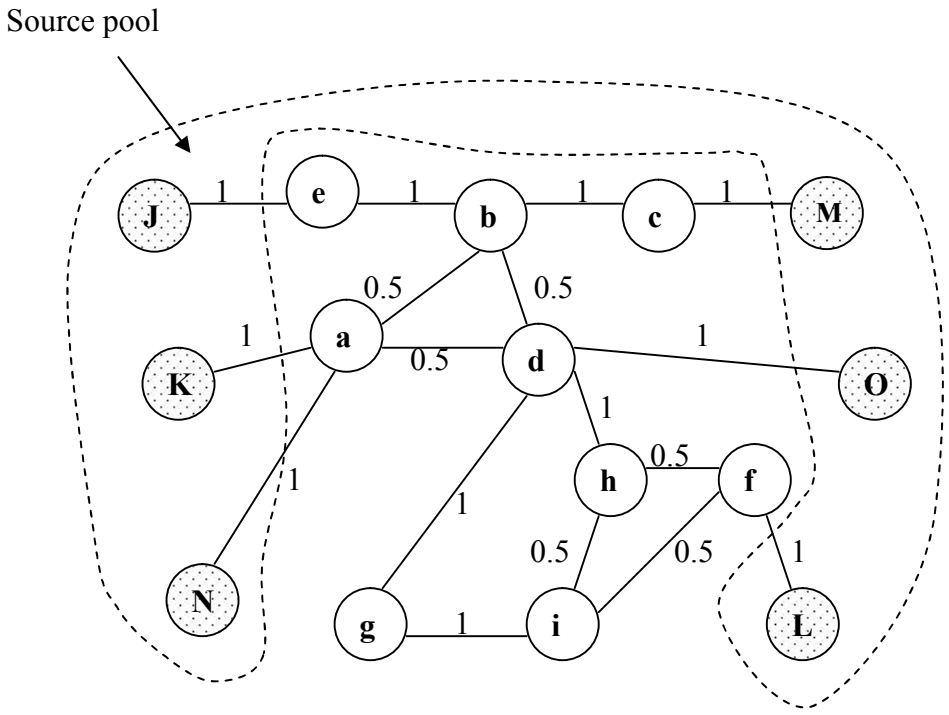


Figure 6.4 The corresponding graph

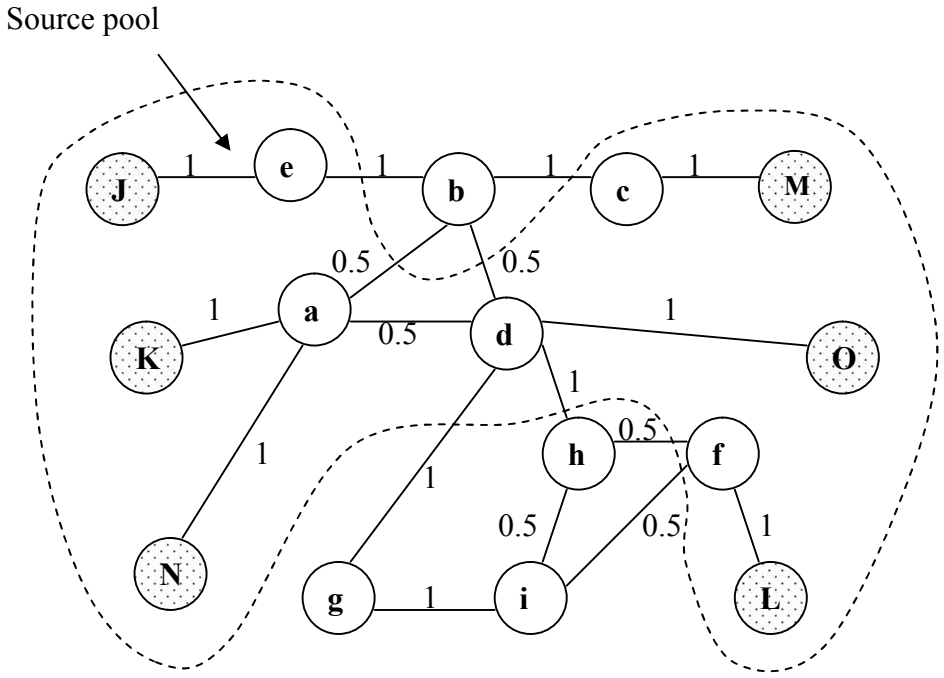


Figure 6.5 The graph after a, e, c, d, f are in the source pool

Figure 6.6 shows a simple circuit:

Three logic blocks with x-coordinates x_1, x_2 and x_3 respectively;

Two I/O pads b_1 and b_2 that are pre-placed at the positions of 0 and 4 (x-coordinates);

And four nets: b_1-x_1, $x_1 - x_2$, $x_2 - x_3$, and x_3-b_2.

Since each of the nets connects two blocks (including I/O pads), the closeness of each corresponding connection is the inverse of two (i.e., 0.5).

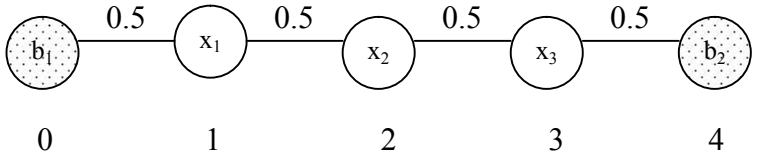


Figure 6.6 Sort the equations

The corresponding equation system is:

$$x_3 = \frac{1}{2}x_2 + 2$$
$$x_2 = \frac{1}{2}x_1 + \frac{1}{2}x_3$$
$$x_1 = \frac{1}{2}x_2$$

Initially, x_1, x_2 and x_3 are set to zero, which is the position of b_1. The source pool has only one vertex b_2. Then, we select a vertex that is outside the source pool and has the largest "closeness" connecting all vertices in the source pool. At this time, x_3 is picked and put into the source pool. Then the next vertex that has the largest closeness is x_2 and x_2 is put into the source pool. Eventually, x_1 is picked and put into the source pool. After all blocks have been put into the source pool, we obtain the sequence of calculating

the variables, which is actually the exact sequence of the corresponding blocks being put into the source pool. A formal description of the ordering heuristic is presented in Fig. 6.7.

[1] convert the circuit into a graph:
a block → a vertex
a net → a clique
[2] **for** any edge e_{ij}, calculate w_{ij} as the reciprocal of the cardinality of the corresponding net
[3] set the "closeness" c_i of block i to 0
[4] *Source_pool* = {}
[5] **for** each I/O pad i
[6] $Source\_pool = Source\_pool \cup \{i\}$
[7] $\forall e_{ij} : j \notin Source\_pool, c_j = c_j + w_{ij}$
[8] **do**
[9] Let $i \notin Source\_pool$ and has the max closeness c_i
[10] $Source\_pool = Source\_pool \cup \{i\}$
[11] $\forall e_{ij} : j \notin Source\_pool, c_j = c_j + w_{ij}$
[12] **until** *Source_pool* contains all *m* terminals
[13] The sequence of blocks being put into Source_pool is the sequence of calculating the positions of the blocks.

Figure 6.7: Ordering heuristic

Figure 6.8 gives the pseudo code of the entire SOR placement method. The first step initializes all x-coordinates using Shrubbery pre-placement. Step 2 uses the aforementioned ordering heuristic to determine the order that will be used to update all of the x_is. Steps 3-9 initialize the middle variables U_l and V_l, the center of gravity x_{cl}, and compute the Star+ estimate S_l for each net *l*. Within the while loop (steps 11-25), each iteration calculates a new *x*, and updates U_l, V_l, x_{cl} and S_l for each affected net *l*. The iterations repeat *number_SOR* times. (The value of *number_SOR* is determined in a similar way to how *number_CG* is determined. See Sections 5.3 and 5.5 regarding how to compute *number_CG*.)

[1] Initialize all x_i
[2] Sort the order of calculating all the x_is
[3] **For** each net l
[4] {
[5] $U_l = \sum_{\forall i \in Net_l} (x_i^{(0)})^2$
[6] $V_l = \sum_{\forall i \in Net_l} x_i^{(0)}$
[7] $x_{cl} = \frac{1}{k_l} V_l$
[8] $S_l = \sqrt{U_l - k_l x_{cl}^2 + 1}$
[9] }
[10] $i = 0$

[11] **While** $i <$ *number_SOR*
[12] {

[13] **For** each block j
[14] {
[15] $x_j^{(k+1)} = (1-\omega) x_j^{(k)} + \frac{\omega}{\sum_{\forall l: j \in Net_l} 1/S_l} \sum_{\forall l: j \in Net_l} \frac{x_{cl}}{S_l}$
[16] For each net l that $j \in l$
[17] {
[18] $U_l = U_l + (x_j^{(k+1)})^2 - (x_j^{(k)})^2$
[19] $V_l = V_l + x_j^{(k+1)} - x_j^{(k)}$
[20] $x_{cl} = \frac{1}{k_l} V_l$
[21] $S_l = \sqrt{U_l - k_l x_{cl}^2 + 1}$
[22] }
[23] }
[24] $i = i + 1$
[25] } //end of while

Figure 6.8: Pseudo-code of SOR placement algorithm

The time complexity of each iteration (steps 13-23) in the SOR placement algorithm is $O(n)$, where n is the number of blocks. Because of the physical limit of the FPGA architecture, each block can only connect to a certain number of nets. This number is a constant in the algorithm implementation. That means in the algorithm described in Figure 6.8, the runtime for step 15 and steps 16-22 is also constant. Therefore, the runtime for the inner loop (steps 13-23) is linear to the number of blocks, which is $O(n)$.

6.3.2 Effect of Ordering Heuristic

In this subsection, we examine the overall effectiveness of the ordering-heuristic proposed in Section 6.3. Table 6.1 compares the quality of the placements obtained when using SOR with and without variable ordering. Column 1 identifies each benchmark by name. Columns 2 and 3 indicate the wirelength and runtime of SOR with variable ordering, while columns 4 and 5 give the same information but this time without variable ordering.

For 14 of the 20 benchmarks, using variable ordering results in a lower wirelength estimate. The average improvement in these thirteen cases is 3.8 percent. In the 6 cases where variable ordering does not help, it does not hurt either. The average difference in solution quality for these 6 cases is only 0.5 percent. Overall, the results in Table 6.1 show that ordering heuristics improves the quality by 2.7% on average, while requiring only 0.4% additional runtime. Given these facts, it is clear that there is a definite benefit to employing variable ordering when using the SOR-based placement method.

Table 6.1: With ordering vs. without ordering

	With Ordering		Without Ordering	
	Star+	**Time**	**Star+**	**Time**
Tseng	11288	0.34375	11884	0.342
Ex5p	18875	0.34375	19982	0.34275
Apex4	21484	0.4375	22012	0.4375
Misex3	22033	0.5	22057	0.4965
Diffeq	17339	0.5625	18595	0.5585
alu4	20990	0.5625	21252	0.55775
Seq	28485	0.6875	28627	0.68725
Apex2	31771	0.765625	31686	0.7585
S298	22161	0.765625	23553	0.75875
Dsip	21929	0.828125	21602	0.82175
Bigkey	25264	0.953125	26348	0.9475
Frisc	61315	2.078125	64210	2.0745
Elliptic	53487	2.0625	54540	2.0515
Spla	73681	2.171875	73421	2.155
Des	32974	1.109375	32838	1.1065
Ex1010	74126	2.921875	79055	2.9215
Pdc	103418	3.03125	103384	3.00125
S38417	72645	4.859375	75675	4.83125
S38584.1	72761	4.90625	72320	4.893
Clma	156409	8.21875	165531	8.19975
Total	**942435**	**38.10938**	**968572**	**37.943**

6.3.3 Choosing the Value of Relaxation Factor ω

In Section 6.2, we mentioned that, in general, the relaxation factor (ω) affects the convergence properties of SOR. For example, if ω=1, the SOR method simplifies to the Gauss-Seidel method. Moreover, Kahan's theorem [112] shows that SOR does not guarantee convergence if ω is not between 0 and 2. Typically, values of ω>1 are used to speedup convergence, while values of ω<1 are often used to establish convergence of a diverging iterative process. Generally, it is not possible to compute a priori the value of ω that is optimal with respect to the rate of convergence of SOR [33]. Even when it is possible to compute the optimal value of ω, the expense of such computation is usually prohibitive [33]. In this subsection, we perform a series of experiments to explore the effect of ω on the convergence of our SOR-based placement method.

In the experiments that follow, SOR was run on all 20 MCNC benchmarks, while the value of ω was varied from 0.2 to 2. For each benchmark and each value of ω, SOR was executed for the exact same number of iterations as the Conjugate Gradient Placement algorithm (see Section 5.3 and Section 5.5 about how to determine the number of iterations in CG placement). The experimental results are summarized in Figure 6.9. The x-axis shows the value of ω from 0.2 to 2 with an incremental step of 0.2. The y-axis gives the total wirelength estimate of the placement solutions obtained by running SOR on 20 MCNC benchmarks, with the corresponding value of ω. Clearly, there is an obvious trend showing that as ω increases the total number of estimated wire segments decreases. In particular, when ω=2 the total estimated number of wire segments is smallest (942435). Based upon these results, we choose to set ω=2 in our implementation. However, it is worthwhile to note that using the smallest ω value (ω=0.2) results in a wirelength estimate only (approximately) 1% worse than when using ω=2 (952746 versus 942435). Thus, the proposed SOR method is quite stable and robust, regardless of what value of ω is used (i.e., 0.2 to 2).

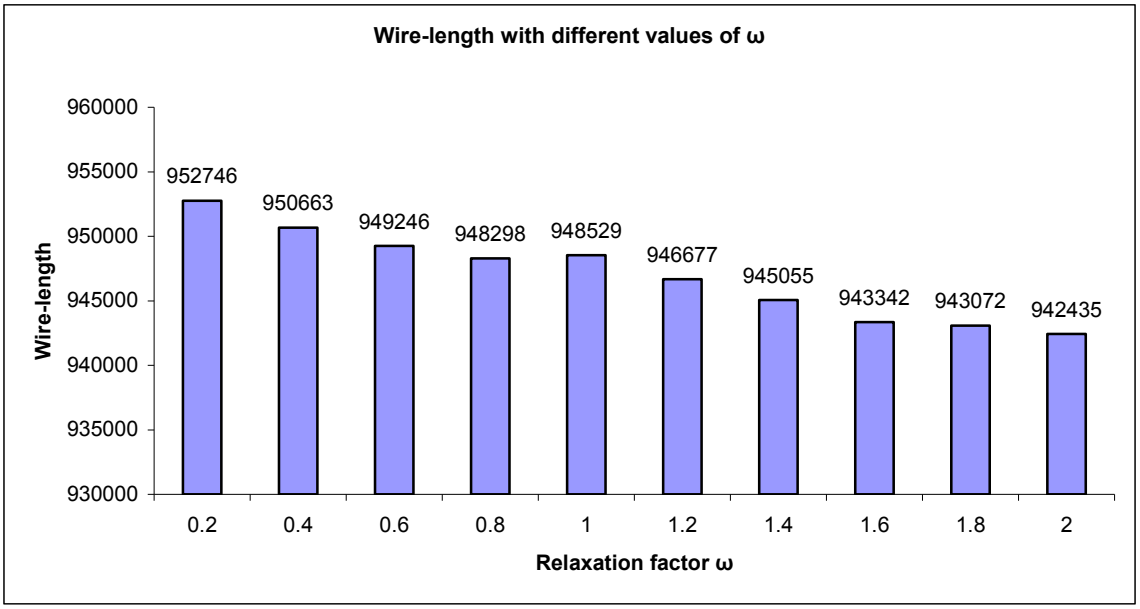


Figure 6.9: Wirelength with different values of ω

6.4 Experimental Results

Having verified that the ordering heuristic is effective, and having determined an appropriate value for the relaxation factor, we now turn our attention to the overall effectiveness of the SOR method compared with the CG method presented in Chapter 4, and the state-of-the-art academic place and route tool, VPR. More specifically, in Section 6.4.1, we compare the SOR-placement head-to-head with the CG-placement with respect to both runtime and estimated wirelength quality. Then, in Section 6.4.2, we compare SOR placement with VPR, this time with respect to runtime, critical-path delay, and wirelength following routing.

6.4.1 SOR versus CG

We begin with a head-to-head comparison of the two new analytic placement methods proposed in this thesis: Conjugate Gradient placement and Successive Over-Relaxation placement. As both of these analytical methods employ the Star+ net model described in Chapter 3, and later adapted for use in analytical methods (Chapter 4) both algorithms are compared on the basis of estimated wirelength (i.e., no actual routing is performed at this point).

Table 6.2 shows the results of the comparison. Column 1 identifies the benchmarks by name. Columns 2 and 3 show the estimated wirelength and run time for SOR placement, while Columns 4 and 5 show similar information for CG placement. The total wirelength and average runtime for both analytic methods is given in the last row of the table.

The results show that with regards to estimated wirelength, for 15 of the 20 benchmarks, SOR is able to find a placement with lower estimated wirelength. The biggest improvement SOR obtains over CG is for the DSIP benchmark where SOR obtains a wirelength estimate that is 10% lower compared with CG. However, it should

be noted that, overall, SOR obtains a 2.6 percent improvement in estimated wirelength compared with CG. Thus, both analytic algorithms perform similarly with respect to solution quality. With regards to runtime, however, there is a significant difference between the two algorithms. For every benchmark, SOR is always faster than CG. Moreover, on average, SOR is approximately 7x faster than CG. The reason why SOR is so much faster than CG can be explained by the fact that the target equation system is non-linear. When using CG to solve a non-linear problem, the search directions on each iteration can quickly lose conjugacy [32], and hence are not as helpful to the convergence of the algorithm. However, calculating conjugate directions is computationally expensive. Therefore, for this specific type of non-linear system, SOR outperforms CG in terms of runtime.

Table 6.2: Comparisons between SOR and CG

	SOR		**CG**	
	Wirelength	**Time** (s)	**Wirelength**	**Time** (s)
alu4	20990	0.5625	21702	2.40625
apex2	31771	0.765625	31744	3.53125
apex4	21484	0.4375	22498	1.71875
Bigkey	25264	0.953125	25703	5.25
Clma	156409	8.21875	151243	73.32813
Des	32974	1.109375	35345	7.484375
Diffeq	17339	0.5625	18410	2.421875
Dsip	21929	0.828125	24160	4.78125
Elliptic	53487	2.0625	55286	12.42188
Ex1010	74126	2.921875	74718	18.10938
Ex5p	18875	0.34375	19629	1.3125
Frisc	61315	2.078125	64443	11.51563
misex3	22033	0.5	22782	2.0625
Pdc	103418	3.03125	104615	18.625
S298	22161	0.765625	21811	3.53125
S38417	72645	4.859375	77707	38.17188
S38584.1	72761	4.90625	73415	41.01563
Seq	28485	0.6875	29966	3.078125
Spla	73681	2.171875	71575	12.34375
Tseng	11288	0.34375	11156	1.40625
Total	**942435**	**38.10938**	**957908**	**264.5156**

6.4.2 SOR versus VPR

Having established that SOR placement is superior to CG placement, both with respect to solution quality and runtime, we now compare SOR to VPR [28] – the state-of-the-art academic place and route tool.

Table 6.3 compares the running time (in seconds) of SOR placement with that of VPR with `inner_num` set to 1 and 10, respectively. Both placement tools were tested using all 20 MCNC benchmarks. The last row of the table shows the total running time to place all 20 benchmarks. The results show that regardless of the value of `inner_num`, SOR always finds a solution faster than VPR for all 20 benchmarks. Moreover, the total runtime for SOR is 38 seconds compared with 169.5 seconds for VPR when run in its fastest mode with inner_num =1. This means that SOR is 4x faster than VPR when run in its fastest mode of operation. Moreover, SOR is 40 times faster than VPR when VPR is run with inner_num =10. Thus, we can see that SOR is significantly faster than VPR, even when VPR is run in its fastest mode.

We now turn our attention to solution quality as measured by critical-path delay. Recall that SOR and VPR use two different net models for estimating wirelength: Star+ and HPWL, respectively. Consequently, any comparison between SOR and VPR, with regards to solution quality, must be performed *after* routing. When using VPR's router, the router is configured to perform timing-driven routing which attempts to improve circuit speed by reducing critical-path delay.

Table 6.4 compares the critical-path delays found when using SOR and VPR with `inner_num=1` and `inner_num=10`, respectively. (Note: the results presented for VPR are the average of 10 independent runs.) The results show that when VPR is run with inner_num=1 (fastest option), SOR finds lower critical-path delays for 15 of the 20 cases. The average reduction for these 15 cases is 14 percent. In the 5 cases that SOR fails to find a lower critical-path delay, the average increase in delay is 7.8 percent.

Overall, SOR finds an 8.8 percent reduction in critical-path delay compared with VPR, and does so typically 4x faster than VPR.

Table 6.3: Running time of SOR and VPR in Seconds

	SOR	VPR	
		inner_num=1	inner_num=10
Tseng	0.34375	1.86	18.6
Ex5p	0.34375	1.83	18.3
Apex4	0.4375	2.2	22
Misex3	0.5	2.53	25.3
Diffeq	0.5625	2.95	29.5
alu4	0.5625	2.66	26.6
Seq	0.6875	3.45	34.5
Apex2	0.765625	3.86	38.6
s298	0.765625	3.45	34.5
Dsip	0.828125	2.8	28
bigkey	0.953125	3.83	38.3
Frisc	2.078125	9.59	95.9
Elliptic	2.0625	10.11	101.1
Spla	2.171875	9.84	98.4
Des	1.109375	3.47	34.7
ex1010	2.921875	13.55	135.5
Pdc	3.03125	13.24	132.4
S38417	4.859375	22.69	226.9
S38584.1	4.90625	22.3	223
Clma	8.21875	33.3	333
Total	**38.11**	**169.51**	**1695.1**

Not surprisingly, when VPR is run with `inner_num=10`, VPR performs much better with respect to critical-path delay. SOR finds a better solution for 11 of the 20 cases. Moreover, the overall average improvement in critical-path delay is reduced to 0.9%. However, it is important to remember that VPR now requires 40x as much runtime as SOR, and even then, SOR finds solutions with lower-critical path delay, on average.

Table 6.4: Critical path delays (SOR vs. VPR)

	CW	SOR	VPR	
			inner_num=1	inner_num=10
Alu4	11	106.857	120.331	113.6717
Apex2	13	116.394	128.77	125.1346
Apex4	14	122.411	127.922	122.6053
bigkey	8	73.6277	100.935	100.0536
Clma	15	268.531	264.999	252.9958
Des	12	124.301	123.01	136.5118
Diffeq	9	97.553	106.112	90.33062
Dsip	8	64.4369	91.0482	93.37907
elliptic	13	180.584	257.387	206.6148
Ex1010	13	191.122	205.552	202.9452
Ex5p	15	102.372	116.071	125.2613
Frisc	15	203.948	227.362	189.0848
Misex3	13	115.636	108.431	105.6976
Pdc	21	236.667	254.422	217.5874
S298	9	203.804	240.983	203.189
S38417	9	159.145	196.969	163.1709
S38584.1	10	140.828	123.888	119.709
Seq	13	96.3664	123.035	118.0495
Spla	18	238.615	205.085	188.0682
Tseng	8	80.6322	81.7572	75.83124
Total		**2924**	**3204**	**2950**

We now turn our attention to wirelength. Table 6.5 compares the placements produced by SOR and VPR (with `inner_num=1` and `inner_num=10`, respectively) with respect to wirelength following routing. For this comparison, the router is told to use a breadth-first strategy. The reason for using a breadth-first strategy is because a breadth-first routing strategy seeks to find a successful routing by minimizing the number of required wire segments to make all connections.

In Table 6.5, Column 1 identifies the benchmark by name. Column 2 indicates the channel width used by the router. The third column shows the actual wirelength required when using SOR. The fourth and fifth columns show the total wirelength required by VPR (with `inner_num`=1 and `inner_num`=10, respectively). All data for VPR is the average of 10 independent runs.

Table 6.5 Wirelength (SOR vs. VPR)

	CW	SOR	VPR	
			inner_num=1	inner_num=10
alu4	11	21652	22038	21016
Apex2	13	31969	32546	30638
Apex4	14	21666	22865	21848
bigkey	8	24029	22396	18505
Clma	15	150305	142509	133592
Des	12	33778	29161	24758
Diffeq	9	15990	16263	14676
Dsip	7	20468	17171	14582
elliptic	13	50419	53811	45912
Ex1010	13	74397	72613	70864
Ex5p	15	20239	19924	18648
Frisc	14	59043	59957	55274
Misex3	13	22417	22700	21871
Pdc	19	105148	104298	99046
S298	9	19960	22703	21346
S38417	9	66750	66586	61764
S38584.1	10	64567	63515	57099
Seq	13	28186	29611	28059
Spla	16	72060	71194	67362
Tseng	8	9835	10420	9423
Total		**912878**	**902282**	**836281**

The results show that when VPR is run with inner_num=1, SOR uses less wirelength for 10 of the 20 cases; for these 10 cases, the average improvement of SOR over VPR is 4.2%; for the 10 cases where VPR gets a better result, the average improvement of VPR over SOR is 4.4%; the overall improvement of VPR over SOR in terms of wire-length is 1.1%. When `inner_num`=10, SOR only outperforms VPR in 2 of the 20 benchmarks; for these two cases, the average improvement of SOR over VPR is 3.7%; for the 18 cases where VPR gets a better result, the average improvement of VPR over SOR is 11%; the overall improvement of VPR over SOR in terms of wirelength is 9.1%. It should be emphasized, however, that as all of the circuits are routable, the primary objective is not minimizing wirelength, but maximizing the speed of the final circuit. Although SOR does not perform as well as VPR with respect to wirelength, SOR does find solutions with 1-9 percent less critical-path delay.

To summarize, SOR is more efficient than VPR. When VPR is run with `inner_num`=1, SOR is 4x faster and gets solutions with 8.8% less delay while using only 1% more wire segments. When VPR is run with `inner_num`=10, SOR is about 40x faster and gets solutions with 0.9% less delay while using 9% more wire segments.

6.5 Convergence of SOR

It is clear that SOR's primary advantage over VPR is its speed. Figure 6.10 shows the convergence properties of SOR and VPR for the CLMA benchmark (the largest benchmark among MCNC20). In Figure 6.10, the x-axis is the time (in seconds) that has passed since the algorithms started, while the y-axis is the number of wire segments. The blue solid line is the convergence curve of VPR with `inner_num` = 1 (the convergence of VPR with `inner_num`=10 has a similar pattern but is much slower), and the red dash line is the convergence curve of SOR. It shows that SOR converges much faster at the beginning of the search than VPR. This feature shows how SOR is able to achieve high-quality solutions much faster that VPR. (Note: Although Fig. 6.10 is only for a specific benchmark, the behaviour it illustrates is typical of that found when using other benchmarks.)

A close look at Figure 6.10 shows that convergence curve of VPR is divided into three phases. The first phase (roughly from 0 to the 11^{th} second in this case) is the period that VPR (based on simulated annealing) is running at high temperature. During this period, the quality of the placement is improved very slowly since a large portion of non-improvement moves are accepted. During the second phase (from the 11^{th} second to the 25^{th} second in this case), most of the improvement in solution quality is made as the temperature goes lower and VPR accepts fewer non-improving moves. During the third phase (after the 25^{th} second in this case), solution quality only improves slightly, since only a few improving moves can be found due to the low temperature of simulated annealing.

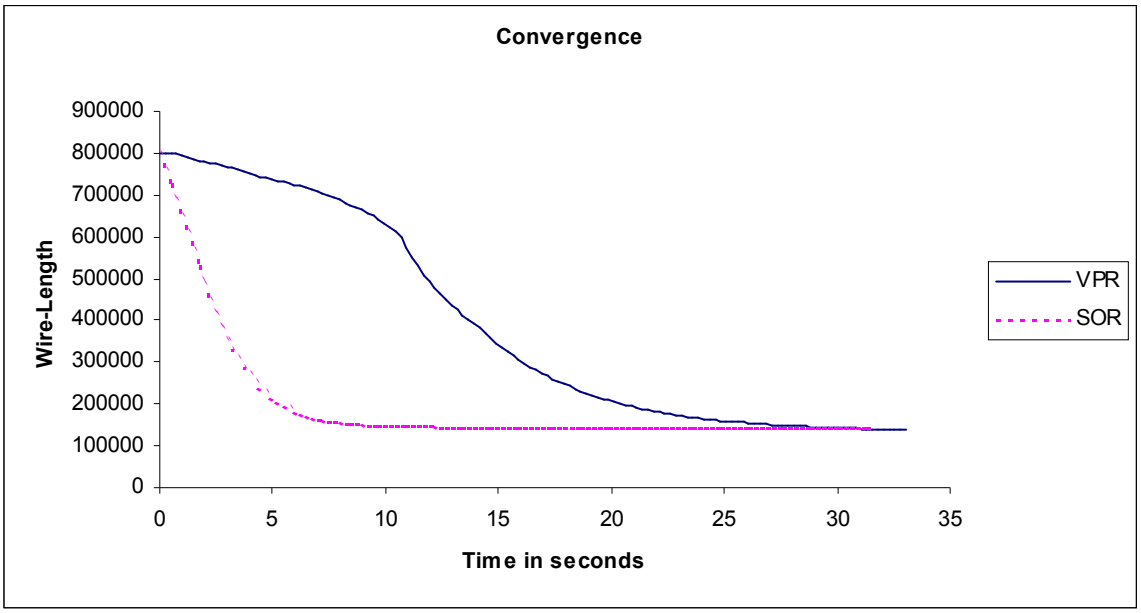


Figure 6.10: Convergence of SOR and VPR

In contrast, the convergence curve for SOR has only two phases. SOR improves the quality drastically during the first phase (from 0 to the 6th second in this case). During the second phase (after the 6th second in this case), the solution quality is still improved but very slowly, and finally the curve flattens out where little further improvement is made. The ability to improve quality at an early stage in the search is essential when a fast solution is required. In this respect, SOR may be more attractive to users, since VPR first spends two thirds of its runtime at high and low temperatures where solution quality improves slowly. SOR, on the other hand, rapidly improves the quality finding a high-quality solution in small amounts of actual runtime.

6.6 Hybrid Approach

Given that SOR is able to find high-quality solutions quickly, in this section we consider the effect of combining both SOR and VPR in the context of a hybrid placement strategy. The basic idea is to first run SOR to quickly find a high-quality solution, and then run VPR to further improve the quality of the solution. Two hybrid strategies are considered:

one where VPR uses the Star+ model (and hence is consistent with SOR) and one where VPR uses the traditional HPWL model.

Table 6.6 shows the critical-path delays for both hybrid approaches based upon using a timing-driven routing option. The first column identifies the benchmark by name. The second column shows the channel width used by the router. The third column gives the critical-path delay when using SOR by itself. The fourth and fifth columns give the critical-path delays of VPR (when run by itself) with `inner_num`=1 and `inner_num`=10, respectively. (All of the data for VPR are the average of 10 runs.) Columns 6 gives the critical-path delays when SOR is run first, followed by VPR using the Star+ model and inner_num=1 (which we denote: SOR+VPR(Star+)). Column 7 provides similar information, but for the case where `inner_num=10`. Columns 8 and 9 give the similar information when VPR is run after SOR but using the traditional HPWL model (which we denote: SOR+VPR(HPWL)).

The following observations can be made. First, running SOR followed by VPR using the Star+ model and inner_num=1 results in a reduction in critical-path delay for 11 (of 20) benchmarks compared with running SOR by itself. The average improvement among these 11 cases is 16.5 percent. In the 9 cases that did not result in an improvement, the average increase in critical-path delay is 13.6 percent. The main reason, that the hybrid approach made the original solution provided by SOR worse, is that VPR, which is based on SA, allows for a high-number of non-improving moves during the early part of the search when a high-temperature is allowed. This can cause a good initial solution to be made substantially worse. When inner_num=1, there may not be enough optimization time to find an equivalent or better solution. Not surprisingly, when inner_num=10, and more time is allowed for optimization, the hybrid strategy results in a better solution being found for 15 of 20 benchmarks. Overall, the hybrid strategy SOR+VPR (Star+) is able to improve on the critical-path delay by 3.2 – 6.7 percent over SOR alone. Moreover, the same hybrid strategy is able to improve on critical-path delay by 7.5 – 11.6 percent over VPR alone.

Table 6.6 Critical path delays (hybrid)

	CW	SOR	VPR		SOR + VPR (Star+)		SOR + VPR (HPWL)	
			inner_num=1	inner_num=10	inner_num=1	inner_num=10	inner_num=1	inner_num=10
Alu4	11	106.857	120.331	113.6717	102.287	95.223	118.73	109.308
Apex2	13	116.394	128.77	125.1346	108.733	103.591	127.654	126.604
Apex4	14	122.411	127.922	122.6053	108.733	98.1068	119.912	117.003
bigkey	8	73.6277	100.935	100.0536	92.9461	76.8831	104.838	103.601
Clma	15	268.531	264.999	252.9958	230.864	246.03	243.582	250.261
Des	12	124.301	123.01	136.5118	156.97	117.784	124.977	127.872
Diffeq	9	97.553	106.112	90.33062	98.9624	82.1605	126.119	77.6612
Dsip	8	64.4369	91.0482	93.37907	75.1003	74.5031	77.404	79.1104
elliptic	13	180.584	257.387	206.6148	256.487	204.424	237.39	175.422
Ex1010	13	191.122	205.552	202.9452	190.389	186.379	208.988	196.069
Ex5p	15	102.372	116.071	125.2613	108.519	96.2305	119.34	127.493
Frisc	15	203.948	227.362	189.0848	207.03	185.652	219.742	185.216
Misex3	13	115.636	108.431	105.6976	110.304	102.782	101.673	105.818
Pdc	21	236.667	254.422	217.5874	225.43	225.133	270.005	222.744
S298	9	203.804	240.983	203.189	193.695	216.344	212.701	182.025
S38417	9	159.145	196.969	163.1709	161.73	142.167	194.709	152.599
S38584.1	10	140.828	123.888	119.709	116.485	124.675	123.172	141.023
Seq	13	96.3664	123.035	118.0495	96.6295	111.082	133.099	109.401
Spla	18	238.615	205.085	188.0682	184.265	170.39	203.353	177.867
Tseng	8	80.6322	81.7572	75.83124	7.875	70.0942	97.7821	78.6372
Total		**2924**	**3204**	**2950**	**2833**	**2730**	**3165**	**2846**

Perhaps not surprisingly, the second hybrid strategy that employs SOR followed by VPR using HPWL does not perform as well. When run with inner_num=1, the hybrid strategy only finds a lower critical-path delay for 5 of the 20 benchmarks compared with SOR by itself. Overall, this hybrid strategy increases the critical-path delay by 8.2 percent. The primary reason for the failure of this second hybrid approach is that SOR and VPR use different objective functions based on Star+ and HPWL, respectively, and their optimization effects counteract with each other.

When inner_num=10 is used, and a longer optimization is performed, the hybrid strategy finds a lower critical-path delay compared with SOR for 11 of the 20 benchmarks. Overall, the hybrid strategy reduces the critical-path delay by 2.7 percent compared with SOR.

Table 6.7 shows the wirelength for both hybrid approaches based upon using a breadth-first routing option. The first column identifies the benchmark by name. The second column shows the channel width used by the router. The third column gives the wirelength when using SOR by itself. The fourth and fifth columns give the wirelength of VPR with `inner_num`=1 and `inner_num`=10, respectively. (All of the data for VPR are the average of 10 runs.) Columns 6 gives the wirelength when SOR is run first, followed by VPR using the Star+ model and inner_num=1. Column 7 provides similar information, but for the case where `inner_num=10`. Columns 8 and 9 give the similar information when VPR is run after SOR but using the traditional HPWL.

Table 6.7 Wirelength (hybrid)

	CW	**SOR**	**VPR**		**SOR + VPR** (Star+)		**SOR + VPR** (HPWL)	
			inner_num=1	inner_num=10	inner_num=1	inner_num=10	inner_num=1	inner_num=10
alu4	11	21652	22038	21016	21639	21058	22684	21315
apex2	13	31969	32546	30638	30622	30004	31816	31501
apex4	14	21666	22865	21848	22278	21984	22471	21770
bigkey	8	24029	22396	18505	22921	20048	21714	17562
clma	15	150305	142509	133592	148859	140725	146427	141009
des	12	33778	29161	24758	35343	29471	29119	25774
diffeq	9	15990	16263	14676	15412	14379	15839	15281
dsip	7	20468	17171	14582	18236	14090	16947	13416
elliptic	13	50419	53811	45912	52704	46629	58153	47273
Ex1010	13	74397	72613	70864	68554	66683	72087	71667
Ex5p	15	20239	19924	18648	20402	19632	20042	18971
frisc	14	59043	59957	55274	59894	56885	60402	55763
misex3	13	22417	22700	21871	21916	21689	22467	22974
pdc	19	105148	104298	99046	104078	101859	107686	104080
S298	9	19960	22703	21346	20961	21053	21424	20623
S38417	9	66750	66586	61764	67447	64067	67931	62438
S38584.1	10	64567	63515	57099	63077	56885	64386	57048
seq	13	28186	29611	28059	28153	26980	29121	28682
spla	16	72060	71194	67362	71050	70169	74162	69143
tseng	8	9835	10420	9423	10274	9530	11120	9000
Total		**912878**	**902282**	**836281**	**903820**	**853820**	**915998**	**855290**

The following observations can be made. Running SOR followed by VPR using the Star+ model and inner_num=1 results in a reduction in wirelength for 12 (of 20) benchmarks compared with running SOR by itself. The average improvement among

these 12 cases is 3.3 percent. In the 8 cases that did not result in an improvement, the average increase in wirelength is 3.1 percent. When inner_num=10, and more time is allowed for optimization, the hybrid strategy results in a better solution being found for 18 of 20 benchmarks. Overall, the hybrid strategy SOR+VPR (Star+) is able to improve on wirelength by 1 – 6.5 percent over SOR alone.

The second hybrid strategy employs SOR followed by VPR using HPWL. When run with inner_num=1, the hybrid strategy finds shorter wirelength for 9 of the 20 benchmarks compared with SOR by itself. Overall, this hybrid strategy increases the wirelength by 0.3 percent. When inner_num=10 is used, the hybrid strategy is able to find less wirelength compared with SOR for 16 of the 20 benchmarks. Overall, the hybrid strategy reduces the wirelength by 6.3 percent compared with SOR.

In summary, using a hybrid strategy based on Star+ always helps. Using a hybrid strategy based on HPWL may result in even worse solutions unless a long-enough time (inner_num=10) is allowed for optimization. However, even then, there is no guarantee that an improved solution will be found.

6.7 Conclusion

In this chapter, we presented an analytic placement algorithm based on Successive Over-Relaxation (SOR). We began by introducing the necessary background for understanding SOR and showed that SOR was a logical extension of the Gauss-Seidel method. We then showed how an SOR placement method could be developed based on the Star+ wirelength model first introduced in Chapter 3.

We then showed how the performance of the SOR-placement method could be improved by employing a novel heuristic to pre-determine the order in which variables appearing in the non-linear equation system should be processed. Our results showed that on average a 2.7% improvement in solution quality (wirelength) could be obtained at the

cost of a 0.4% increase in runtime. We also showed that using a relaxation factor 2 causes the algorithm to converge slightly faster compared with other, smaller relaxation values.

Both SOR placement and CG placement were compared head-to-head using all 20 MCNC benchmarks. The results revealed that although both analytic methods find solutions with similar quality, SOR finds solutions, on average, 7x faster. The superior performance of SOR is attributable to the fact that when CG is used to solve a non-linear system, the search directions quickly loose conjugacy [32], thus slowing the convergence rate of the algorithm.

Due to its superior run-time performance, SOR was compared head-to-head with VPR, the state-of-the-art academic placement and routing tool, using all 20 MCNC benchmarks. The results showed that SOR is approximately 4x-40x faster than VPR, while obtaining solutions with approximately 1-8.8 percent less critical-path delay. However, the solutions found by SOR required approximately 1-8.8 percent more wire segments to implement.

Due to the excellent convergence properties of SOR placement, we also considered a hybrid approach where SOR was used to quickly find a fast initial placement, and then VPR was used to improve the quality of the placement. Our results showed that the hybrid approach was able to further improve critical-path delay by approximately 3-7 percent and wirelength by 1-7 percent, compared with SOR alone, at a cost of 4-40x more runtime.

Chapter 7

Conclusions and Future Work

7.1 Contributions

This work has contributed two analytic methods based on a new net model for FPGA placement. The new net model and the placement methods developed in this research are briefly summarized in Table 7.1.

In Chapter 3, we presented a novel model, called *Star+*, for estimating wirelength during FPGA placement. Unlike the traditional HPWL model [30][31], employed by many FPGA placement tools including VPR [27][28], the Star+ model is continuously differentiable and hence suitable for use with analytic methods. However, as was shown in Chapter 3, the Star+ model can also be employed effectively in iterative-improvement placement methods, like those based on Simulated Annealing [54].

Table 7.1: Summary of contributions

Contributions	Features
Star+ model	• Differentiable • Outperforms HPWL 6-9% in terms of critical-path delay • Constant updating time
Shrubbery pre-placement	• Outperforms random pre-placement by 1.2% • Time complexity $O(\|E\|log\|V\|)$
CG placement method	• $O(n)$ computation time of each iteration • Similar quality compared with VPR
SOR placement method	• 4 – 40x faster than VPR • Improves critical-path delay by 1 – 8.8% over VPR
Hybrid placement method	• Improves critical-path delay by 3.2 – 6.7% over SOR • Improves wire-length by 1 – 6.5% over SOR • Costs 4 – 40x extra runtime.
Timing-Driven placement method based on SOR	• Optimizes wirelength and critical-path delay

To establish the effectiveness of the Star+ model, an empirical comparison of Star+ and HPWL [30][31] was made by first incorporating both wirelength estimation models into VPR [27][28] – the state-of-the-art academic placement and routing tool based on simulated annealing – then using VPR to place and route all 20 MCNC benchmarks [62]. Overall, the results showed that VPR has similar and, in some instances, superior performance when using Star+ compared with HPWL. More specifically, the results show that the Star+ model achieves a 6-9% reduction in critical-path delay compared with HPWL, while producing similar quality results with respect to routability (measured in terms of channel width) and total number of wire segments.

An important characteristic of the Star+ model is that the time to calculate incremental changes in cost from moving/swapping blocks can always be computed in *O(1)* time. Moreover, it was shown that as the size (cardinality) of a net increases, Star+ significantly outperforms HPWL with respect to the time required to re-compute the wirelength estimate following an improving move or swap.

In Chapter 4, we presented an analytic placement algorithm based upon the conjugate gradient (CG) method. Unlike previous analytic placement methods that try to optimize an objective function based on quadratic distance, the CG-based placement algorithm seeks to minimize an objective function based on the (near-linear) Star+ model. Due to the linear nature of distance, the Star+ model is theoretically more accurate than quadratic distance. The accuracy of the Star+ model, however, comes at the expense of the complexity of the analytic algorithm that employs it. To minimize quadratic distance, only a *linear* equation system must be solved. To minimize the Star+ model a *non-linear* equation system must be solved, which is usually much harder and hence more time-consuming. Another important feature of the algorithm is that the computation complexity of each iteration is O(n) even though the target system is not sparse. Due to the high performance and accuracy of the Star+ model, our method is able to produce placements with similar quality compared with VPR [27].

In Chapter 5, we developed a pre-placement algorithm, called Shrubbery, to guarantee that the algorithm produces non-trivial placements. The *goal* of the pre-placement is to place the I/O pads in such a way that the I/O pads with higher connectivity are placed closer together than the I/O pads with lower connectivity. Also, this pre-placement provisionally locates components on the periphery of the FPGA, which causes other components to be distributed throughout the chip. We demonstrated that Shrubbery is able to outperform a random pre-placement strategy that seeks to find the best (initial) placement by generating a pool of random placements, both with respect to solution quality and runtime. Unlike the random pre-placement strategy, Shrubbery only requires a single application of CG, which is much faster than the multiple applications of CG that are required to create a pool of random pre-placements. Most importantly, we were able to show, both theoretically and empirically, that the running time of Shrubbery is extremely small. In Chapter 5, We also showed that CG conclude that CG is competitive with VPR in its fast mode (`inner_num`=1), but produces slightly lower quality solutions than VPR when `inner_num`=10. By reducing the runtime of each iteration from $O(n^2)$ to $O(n)$, we have improved the speed of CG significantly and made it about 5 times faster than VPR when `inner_num`=10.

In Chapter 6, we introduced an analytic placement algorithm that uses Successive Over-Relaxation (SOR) method and the Star+ model. An advantage of using the Star+ model is that the wire-length estimate change of a net caused by moving a single block can always be calculated in a constant time. Since moving a block is the mostly used operation in SOR method, the time for re-computing the cost after a movement counts a major part of the total running time. Therefore, constant time for updating the net wire-length estimate is critical to build a time-efficient SOR placement method. We demonstrated that SOR method based on the Star+ model is 4 – 40 times faster than VPR, and outperforms VPR by 1 – 8.8% with respect to critical-path delay. Due to the excellent convergence properties of SOR placement, we also considered a hybrid approach where SOR was used to quickly find a fast initial placement, and then VPR was used to improve the quality of the placement. Our results showed that the hybrid approach was able to further improve critical-path delay by approximately 3-7 percent and wirelength by 1-7 percent, compared with SOR alone, at a cost of 4-40x more runtime.

7.2 Future Work

There are five ways to enhance this work: 1) applying analytic placement in a multilevel optimization context to further improve the performance and solution quality; 2) designing a router based on Steiner heuristics to improve routing results; 3) acceleration via multi-core and/or re-configurable computing; 4) extending the model to include timing and congestion; and 5) extending the model to handle modern FPGA architectures.

7.2.1 Multilevel Optimization

As FPGAs continue to increase in logic capacity and functionality, so do the designs mapped to them. Multilevel methods are able to reduce problem complexity. These methods construct a hierarchy of successively coarser problems from the bottom up by recursive aggregation. They employ iterative improvement at each of the resulting levels,

transfer these improvements up and down the hierarchy, and eventually terminate with a solution at the original, finest level. Although multilevel placement has become a very active research topic, with several high-quality placers developed for standard cell-based designs, little work has been done on multilevel placement for FPGAs. Recently, Areibi et. al. [44] have shown the benefit of multilevel optimization by applying multilevel to FPGA placement. However, it is likely that many of the multilevel placement techniques developed for standard cells can be applied to further enhance the quality of the work proposed in this thesis. An important concept in multilevel is to use *different* placement algorithms at different levels of the hierarchy. At the higher level of the hierarchy, we can use the proposed analytic method to quickly achieve a globally good placement of clusters, and use other techniques (e.g., local improvement and simulated annealing) at the low level of the hierarchy to fine-tune the solution quality.

7.2.2 FPGA Routing

Routing is a time-consuming step that determines the actual interconnects (wire-length, congestion, delay) and plays a critical role in the overall FPGA system performance. Given the similarity between FPGA and standard-cell global routing, many FPGA routers are based on algorithms developed for routing ASICs, including VPR. The basic framework of current FPGA router is based on *iterative* routing, where on each iteration, the routing of each net is based on maze expansion of a multi-pin net in the routing graph. Instead of maze expansion one may use *graph-based Steiner heuristics* to construct a near-optimal Steiner tree in the graph. However, to the best of my knowledge, no one has tried replacing the maze expansion engine in VPR with a graph-Steiner-based algorithm. We proposed two graph-based Steiner heuristics, *Shrubbery* and *Pole-Center*, in [111]. The experiments done in [111] suggest that using these algorithms to replace maze expansion in VPR may drastically improve the routing runtime, while maintaining the solution quality.

7.2.3 Algorithm Acceleration via Multi-Core and/or Re-configurable Computing

An important means to achieve highly scalable placement and routing algorithms is to make the best use of multi-core computing systems and hardware acceleration available through re-configurable computing. These approaches can be investigated in three aspects: 1) implement the algorithm directly in hardware using an FPGA; 2) employ a hardware/software co-design approach where time-critical bottlenecks within the algorithm are implemented directly in hardware, with the remainder of the algorithm implemented in software running on a soft core; and 3) develop an Application Specific Instruction Set Processor (ASIP) to run the algorithm.

7.2.4 Timing and Congestion

Today, it is important to perform timing [13][23], congestion [71], and even low-power [11] optimization when performing placement. In Appendix A, we provide a timing-driven model that carries out timing optimization by including an additional timing cost term to the objective function. The timing cost is the summation of the delay times and the timing criticality over all connections in the design, where the critically of a connection depends on its timing slack. Two important issues resulting from this model will need to be considered: 1) When to update the slacks and 2) how to optimize two very different objectives. With regards to the former issue, it is suggested that slack values be evaluated after every iteration, thus leading to shorter runtime and more effective optimization. With regards to the latter issue, optimizing two objectives will require careful scaling to ensure that the relative importance of both wirelength and timing are independent of their actual values.

7.2.5 Modern FPGA Architectures

Today's modern FPGA architectures contain a variety of features including hardwired macro blocks such as embedded memories, multipliers, DSP blocks, just to name a few. An important consideration will be how best to extend an analytical method to handle blocks of different types. Given the ease with which simulated annealing can swap blocks (or groups of blocks representing more complex structures), it may be beneficial to consider using a hybrid analytic/simulated-annealing based algorithm for these types of architectures.

Appendix A

Timing-Driven Placement

In Chapters 4 – 6, we introduced analytic methods based on the Star+ model that minimize wirelength. In this appendix, we will show that the Star+ model can also be used in analytic timing-driven placement algorithms, by presenting an SOR timing-driven placement algorithm that is based on the Star+ model. This appendix is organized as follows. Section A.1 gives some background of timing analysis. Section A.2 illustrates the timing-driven placement algorithm.

A.1 Background

The objective of timing-driven placement is to place logic blocks that are on the critical path into CLBs that are close to each other and therefore to minimize the amount of wire segments and interconnects that the critical signals must travel.

Timing-driven placement algorithms can be roughly divided into two categories: path-based and net-based. A path-based algorithm computes the delays of all paths and directly minimizes the maximum delay (critical path delay). In contrast, a net-based algorithm uses static timing analysis to assign each net a criticality weight; higher weights are assigned to nets that are more timing critical. A path-based algorithm usually gives a more accurate timing inspection. However, this type of algorithm suffers high computation complexity due to the exponential number of paths that have to be simultaneously considered.

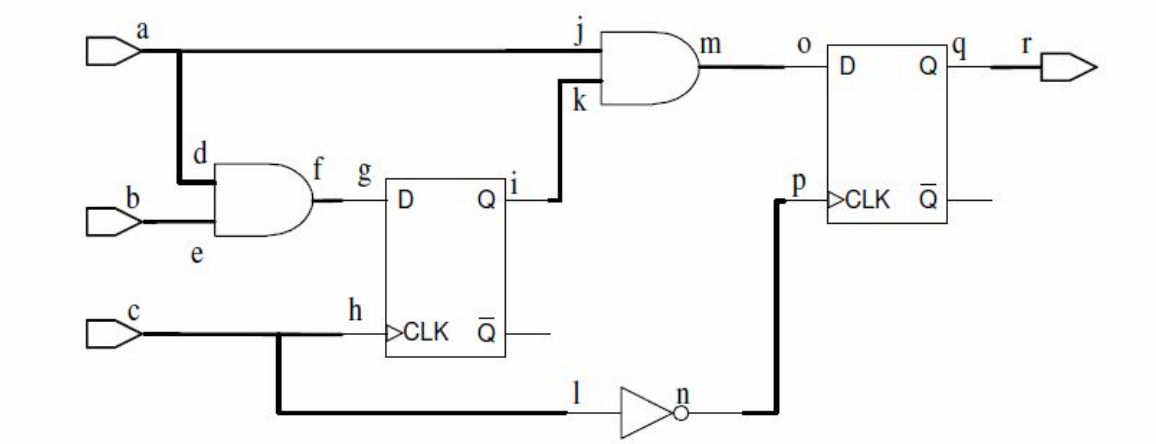
A.1.1 Timing Analysis

To perform timing analysis, a circuit is first converted into a directed graph $G(V, E)$. Each wire and each logic block pin becomes a node in the graph, where a pin comes from a look-up table (LUT), a register, or an I/O pad. A node is a *source* if the original pin is an input pad or a register output. A node is a *sink* if the original pin is an output pad or a register input. Each switch becomes a directed edge connecting two corresponding nodes. Each edge is assigned a weight that represents the physical delay between the nodes. Figure A.1 gives an example of a circuit and the corresponding timing analysis graph. A path starts at a source and ends at a sink. Give a node j, the arrival time, $T_{arrival}(j)$ is the time at which the signal at node j settles to its final value. The signal at node j becomes stable a delay time after when the signal at all input nodes to node j settle to the final value. To determine the arrival time of all the nodes, a breadth first traversal is performed on the graph, which starts from the sources. The arrival time $T_{arrival}$ of each node j can iteratively calculated with the following equation:

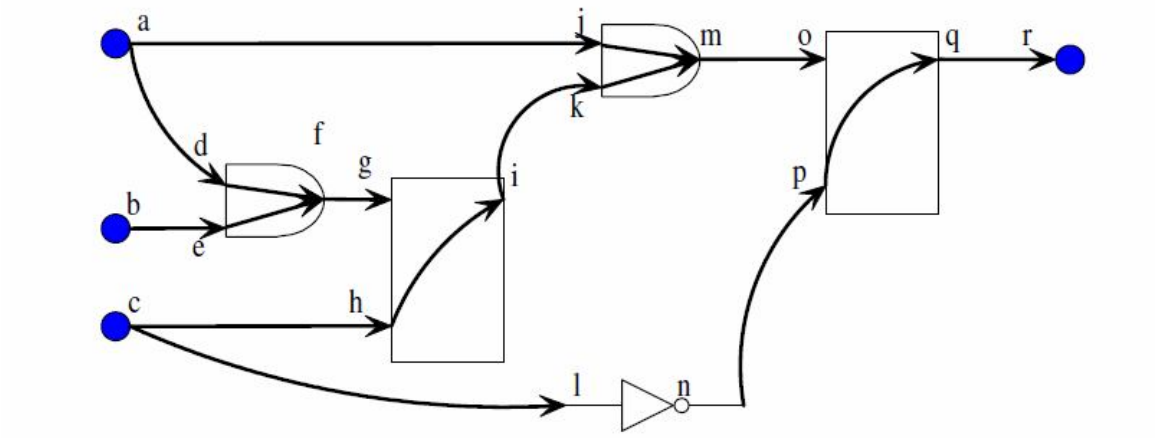
$$T_{arrival}(j) = \begin{cases} 0, & j \in sources \\ \max_{\forall i \in fanin(j)} \{T_{arrival}(i) + delay(i, j)\}, & (i, j) \in E \end{cases}$$

where *delay(i,j)* is the delay value of the edge connecting the node i to node j. The maximum arrival time of all the nodes determines the delay of the circuit and is labeled as D_{max}, and is calculated as:

$$D_{\max} = \max\{T_{arrival}(j)\}, \qquad j \in sinks$$



(a) Circuit.



(b) Timing analysis graph.

Figure A.1: Timing analysis graph

Our goal is to minimize D_{max}. However, simply re-placing the blocks on the path from the source to the node with the largest $T_{arrival}$ for the purpose of reducing D_{max} just does not work, since reducing the delay on one path will inevitably add delays to the connections on some other paths that may become critical thereafter. Accordingly, it is useful to find out how much delay can be added to each connection before the corresponding path becomes critical. This amount of delay is called the *slack*.

To determine the slack of each connection, it is necessary to know the maximum required arrival time of every node with the condition that D_{max} is not increased. Obviously, the maximum required arrival time of all output pads and register inputs is D_{max}. The maximum required arrival time of other nodes is computed as:

$$T_{required}(i) = \min_{\forall j \in fanout(i)} \{T_{required}(j) - delay(i,j)\}$$

where *i* is the source of a net; *j* is a sink of the net; and the *delay(i,j)* is the delay value of the connection from node *i* to node *j*. The slack of connection (*i,j*) is defined as:

$$slack(i,j) = T_{required}(j) - T_{arrival}(i) - delay(i,j)$$

A.1.2 Criticality and Cost

The criticality of the connection from the source *i* to a sink *j* is defined as:

$$crit(i,j) = 1 - \frac{slack(i,j)}{D_{\max}}$$

As $slack(i,j)$ is always between 0 and $D_{\max}$. The value of $crit(i,j)$ is therefore between 0 and 1. The slack of the connections on a critical path is always 0 and the criticality is always 1.

In VPR, the time cost of a connection is calculated as:

$$Time\_Cost(i,j) = delay(i,j) \cdot crit(i,j)^{Crit\_Expoenet}$$

And the total time cost of a circuit equals the sum of the time cost of all its connections:

$$Time\_Cost = \sum_{\forall i,j \in circuit} Time\_Cost(i,j)$$

VPR uses a trade-off variable λ to determine the weight of time cost and wirelength cost. The change in the combined cost is calculated using the following formula:

$$\Delta C = \lambda \cdot \frac{\Delta Time\_Cost}{previous\_Time\_Cost} + (1-\lambda)\frac{\Delta wirelength}{previous\_wirelength}$$

Nevertheless, the above formula does not work for analytic methods, since analytic methods need a clearly defined objective function like Equation 4.8. For this

purpose, we present the Star+ time-cost model of net *l* in a format similar to the Star+ wirelength model:

$$T_l = \sqrt{\sum_{\forall i \in fanout(l)} [crit(sl,i)(x_i - x_{sl})]^2} \qquad \text{(Equation A.1)}$$

where *sl* is the source (or driver) of net *l*, and x_{sl} is the *x*-coordinate of *sl*. The *crit(sl,i)* is the criticality of connection (sl,i).

By introducing time cost factor into Equation 4.8, we get the total combined cost of wirelength and time for the circuit as:

$$\sum_{\forall l \in circuit} \left\{ \lambda \left(\alpha \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} \right) + (1-\lambda)\alpha \cdot T_l \right\} \qquad \text{(Equation A.2)}$$

where λ is the trade-off parameter. Since $S_l = \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta}$ (Equation 4.9), the total combined cost of a circuit (Equation A.2) can be rewritten as:

$$\sum_{\forall l \in circuit} \{\lambda(\alpha \cdot S_l) + (1-\lambda)\alpha \cdot T_l\} \qquad \text{(Equation A.3)}$$

where $(\alpha \cdot S_l)$ is the wirelength cost of net *l*, and $(\alpha \cdot T_l)$ represents the time cost. As α appears in both parts, minimizing Equation A.3 is equivalent to minimizing the following equations obtained by removing α:

$$f(x) = \sum_{\forall l \in circuit} \{\lambda \cdot S_l + (1-\lambda) T_l\} \qquad \text{(Equation A.4)}$$

The above equation is used as the objective function of the presented analytic timing-driven placement. In the next section, we will introduce how to obtain the recurrence relation of SOR iteration from Equation A.4.

A.2 SOR Timing-driven Placement

As $f(x)$ is positive-definite, it can be minimized by solving the equation system $f'(x)=0$. The gradient $f'(x)$ of $f(x)$ is defined as follows:

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \text{M} \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix},$$

where the *j*th element of $f'(x)$ is the partial derivative of $f(x)$ with respect to x_j:

$$\frac{\partial}{\partial x_j} f(x) = \frac{\partial}{\partial x_j} \sum_{\forall l \in circuit} \{\lambda \cdot S_l + (1-\lambda) T_l\}$$

$$= \frac{\partial}{\partial x_j} \sum_{\forall l: j \in Net_l} \{\lambda \cdot S_l + (1-\lambda) T_l\}$$

$$= \sum_{\forall l: j \in Net_l} \left\{ \lambda \cdot \frac{\partial S_l}{\partial x_j} + (1-\lambda) \frac{\partial T_l}{\partial x_j} \right\} \qquad \text{(Equation A.5)}$$

From Equation 4.11, we have $\frac{\partial S_l}{\partial x_j} = \frac{x_j - x_{cl}}{S_l}$ when $j \in Net_l$.

From Equation A.1, we have:

$$\frac{\partial T_l}{\partial x_j} = \frac{\partial}{\partial x_j} \sqrt{\sum_{\forall i \in fanout(l)} [crit(sl,i)(x_i - x_{sl})]^2}$$

$$= \frac{1}{2\sqrt{\sum_{\forall i \in fanout(l)} [crit(sl,i)(x_i - x_{sl})]^2}} \frac{\partial}{\partial x_j} \left\{ \sum_{\forall i \in fanout(l)} [crit(sl,i)(x_i - x_{sl})]^2 \right\}$$

$$= \frac{1}{2\sqrt{\sum_{\forall i \in fanout(l)} [crit(sl,i)(x_i - x_{sl})]^2}} \left\{ \sum_{\forall i \in fanout(l)} \frac{\partial}{\partial x_j} [crit(sl,i)(x_i - x_{sl})]^2 \right\}$$

$$= \frac{1}{2\sqrt{\sum_{\forall i \in fanout(l)} [crit(sl,i)(x_i - x_{sl})]^2}} \left\{ \sum_{\forall i \in fanout(l)} 2[crit(sl,i)^2 (x_i - x_{sl}) \frac{\partial}{\partial x_j} (x_i - x_{sl})] \right\}$$

$$= \frac{1}{\sqrt{\sum_{\forall i \in fanout(l)} [crit(sl,i)(x_i - x_{sl})]^2}} \left\{ \sum_{\forall i \in fanout(l)} [crit(sl,i)^2 (x_i - x_{sl}) (\frac{\partial x_i}{\partial x_j} - \frac{\partial x_{sl}}{\partial x_j})] \right\}$$

$$= \frac{1}{T_l}\left\{\sum_{\forall i \in fanout(l)} [crit(sl,i)^2 (x_i - x_{sl})(\frac{\partial x_i}{\partial x_j} - \frac{\partial x_{sl}}{\partial x_j})]\right\} \qquad \text{(From Equation A.1)}$$

When j is a sink of net l (i.e., $j \neq sl$ or $j \in fanout(l)$), we have $\frac{\partial x_i}{\partial x_j} = 0$ (when $i \neq j$), $\frac{\partial x_j}{\partial x_j} = 1$ and $\frac{\partial x_{sl}}{\partial x_j} = 0$. The above equation can be simplified as:

$$\frac{\partial T_l}{\partial x_j} = \frac{1}{T_l}\left\{\sum_{\forall i \in fanout(l)} [crit(sl,i)^2 (x_i - x_{sl})(\frac{\partial x_i}{\partial x_j} - 0)]\right\}$$

$$= \frac{1}{T_l}\left\{\sum_{i=j} [crit(sl,i)^2 (x_i - x_{sl})\frac{\partial x_i}{\partial x_j}] + \sum_{\forall i \in fanout(l) \wedge i \neq j} [crit(sl,i)^2 (x_i - x_{sl})\frac{\partial x_i}{\partial x_j}]\right\}$$

$$= \frac{1}{T_l}\left\{[crit(sl,j)^2 (x_j - x_{sl})\frac{\partial x_j}{\partial x_j}] + \sum_{\forall i \in fanout(l) \wedge i \neq j} [crit(sl,i)^2 (x_i - x_{sl}) \cdot 0]\right\}$$

$$= \frac{crit(sl,j)^2 (x_j - x_{sl})}{T_l}$$

When j is the source (or driver) of net l (i.e., $j = sl$ or $j \in fanin(l)$), we have $\frac{\partial x_{sl}}{\partial x_j} = 1$ and all other $\frac{\partial x_i}{\partial x_j} = 0$ (because $i \in fanout(l)$ and hence $i \neq j$). The $\frac{\partial T_l}{\partial x_j}$ then can be simplified as:

$$\frac{\partial T_l}{\partial x_j} = \frac{1}{T_l}\left\{\sum_{\forall i \in fanout(l)} [crit(sl,i)^2 (x_i - x_{sl})(0-1)]\right\}$$

$$= \frac{1}{T_l}\left\{\sum_{\forall i \in fanout(l)} [crit(sl,i)^2 (x_{sl} - x_i)]\right\}$$

$$= \frac{1}{T_l}\left\{\sum_{\forall i \in fanout(l)} [crit(j,i)^2 (x_j - x_i)]\right\}$$

To summarize,

$$\frac{\partial T_l}{\partial x_j} = \begin{cases} \frac{1}{T_l}\left\{\sum_{\forall i \in fanout(l)} [crit(j,i)^2 (x_j - x_i)]\right\}, & \text{if } j \in fanin(l) \\ \frac{crit(sl,j)^2 (x_j - x_{sl})}{T_l}, & \text{if } j \in fanout(l) \end{cases} \qquad \text{(Equation A.6)}$$

By replacing Equation 4.11 and Equation A.6 in Equation A.5, we have:

$$\frac{\partial}{\partial x_j} f(x) = \sum_{\forall l: j \in Net_l} \left\{ \lambda \cdot \frac{\partial S_l}{\partial x_j} + (1-\lambda) \frac{\partial T_l}{\partial x_j} \right\}$$

$$= \sum_{\forall l: j \in Net_l} \left\{ \lambda \cdot \frac{\partial S_l}{\partial x_j} \right\} + \sum_{\forall l: j \in Net_l} \left\{ (1-\lambda) \frac{\partial T_l}{\partial x_j} \right\}$$

$$= \sum_{\forall l: j \in Net_l} \left\{ \lambda \cdot \frac{\partial S_l}{\partial x_j} \right\} + \sum_{\forall l: j \in fanin(l)} \left\{ (1-\lambda) \frac{\partial T_l}{\partial x_j} \right\} + \sum_{\forall l: j \in fanout(l)} \left\{ (1-\lambda) \frac{\partial T_l}{\partial x_j} \right\}$$

$$= \sum_{\forall l: j \in Net_l} \left\{ \lambda \cdot \frac{x_j - x_{cl}}{S_l} \right\} + \sum_{\forall l: j \in fanin(l)} (1-\lambda) \left\{ \frac{1}{T_l} \sum_{\forall i \in fanout(l)} [crit(j,i)^2 (x_j - x_i)] \right\}$$
$$+ \sum_{\forall l: j \in fanout(l)} \left\{ (1-\lambda) \frac{crit(sl,j)^2 (x_j - x_{sl})}{T_l} \right\}$$

By making $\frac{\partial}{\partial x_j} f(x) = 0$, we get the equation:

$$\sum_{\forall l: j \in Net_l} \left\{ \lambda \cdot \frac{x_j - x_{cl}}{S_l} \right\} + \sum_{\forall l: j \in fanin(l)} (1-\lambda) \left\{ \frac{1}{T_l} \sum_{\forall i \in fanout(l)} [crit(j,i)^2 (x_j - x_i)] \right\}$$
$$+ \sum_{\forall l: j \in fanout(l)} \left\{ (1-\lambda) \frac{crit(sl,j)^2 (x_j - x_{sl})}{T_l} \right\} = 0$$

$$\sum_{\forall l: j\in Net_l}\left\{\lambda\cdot\frac{x_j}{S_l}\right\}-\sum_{\forall l: j\in Net_l}\left\{\lambda\cdot\frac{x_{cl}}{S_l}\right\}+\sum_{\forall l: j\in fanin(l)}(1-\lambda)\left\{\frac{1}{T_l}\sum_{\forall i\in fanout(l)}[crit(j,i)^2 x_j]\right\}$$

$$-\sum_{\forall l: j\in fanin(l)}(1-\lambda)\left\{\frac{1}{T_l}\sum_{\forall i\in fanout(l)}[crit(j,i)^2 x_i]\right\}+\sum_{\forall l: j\in fanout(l)}\left\{(1-\lambda)\frac{crit(sl,j)^2 x_j}{T_l}\right\}$$

$$-\sum_{\forall l: j\in fanout(l)}\left\{(1-\lambda)\frac{crit(sl,j)^2 x_{sl}}{T_l}\right\}=0$$

$$x_j\sum_{\forall l: j\in Net_l}\frac{\lambda}{S_l}+x_j\sum_{\forall l: j\in fanin(l)}\left(\frac{1-\lambda}{T_l}\sum_{\forall i\in fanout(l)}crit(j,i)^2\right)+x_j\sum_{\forall l: j\in fanout(l)}\left(\frac{1-\lambda}{T_l}crit(sl,j)^2\right)$$

$$-\sum_{\forall l: j\in Net_l}\frac{\lambda x_{cl}}{S_l}-\sum_{\forall l: j\in fanin(l)}\left[\frac{1-\lambda}{T_l}\sum_{\forall i\in fanout(l)}\left(crit(j,i)^2 x_i\right)\right]-\sum_{\forall l: j\in fanout(l)}\left[\frac{1-\lambda}{T_l}crit(sl,j)^2 x_{sl}\right]=0$$

$$x_j\left\{\sum_{\forall l: j\in Net_l}\frac{\lambda}{S_l}+\sum_{\forall l: j\in fanin(l)}\left(\frac{1-\lambda}{T_l}\sum_{\forall i\in fanout(l)}crit(j,i)^2\right)+\sum_{\forall l: j\in fanout(l)}\left(\frac{1-\lambda}{T_l}crit(sl,j)^2\right)\right\}=$$

$$\sum_{\forall l: j\in Net_l}\frac{\lambda x_{cl}}{S_l}+\sum_{\forall l: j\in fanin(l)}\left[\frac{1-\lambda}{T_l}\sum_{\forall i\in fanout(l)}\left(crit(j,i)^2 x_i\right)\right]+\sum_{\forall l: j\in fanout(l)}\left[\frac{1-\lambda}{T_l}crit(sl,j)^2 x_{sl}\right]$$

$$x_j=\left\{\sum_{\forall l: j\in Net_l}\frac{\lambda}{S_l}+\sum_{\forall l: j\in fanin(l)}\left(\frac{1-\lambda}{T_l}\sum_{\forall i\in fanout(l)}crit(j,i)^2\right)+\sum_{\forall l: j\in fanout(l)}\left(\frac{1-\lambda}{T_l}crit(sl,j)^2\right)\right\}^{-1}$$

$$\left\{\sum_{\forall l: j\in Net_l}\frac{\lambda x_{cl}}{S_l}+\sum_{\forall l: j\in fanin(l)}\left[\frac{1-\lambda}{T_l}\sum_{\forall i\in fanout(l)}\left(crit(j,i)^2 x_i\right)\right]+\sum_{\forall l: j\in fanout(l)}\left[\frac{1-\lambda}{T_l}crit(sl,j)^2 x_{sl}\right]\right\}$$

In above equation, $x_{cl}=\frac{1}{k_l}\sum_{\forall i\in Net_l}x_i$, $S_l=\sqrt{\sum_{\forall i\in Net_l}x_i^2-k_l x_{cl}^2+1}$ and $T_l=\sqrt{\sum_{\forall i\in fanout(l)}[crit(sl,i)(x_i-x_{sl})]^2}$. Now, by putting the iteration number into these equations we get the Jacobi iteration for placement:

$$x_{cl}^{(k)}=\frac{1}{k_l}\sum_{\forall i\in Net_l}x_i^{(k)}$$

$$S_l^{(k)}=\sqrt{\sum_{\forall i\in Net_l}(x_i^{(k)})^2-k_l(x_{cl}^{(k)})^2+1}$$

$$T_l^{(k)}=\sqrt{\sum_{\forall i\in fanout(l)}[crit(sl,i)(x_i^{(k)}-x_{sl}^{(k)})]^2}$$

$$x_j^{(k+1)} = \left\{ \sum_{\forall l: j \in Net_l} \frac{\lambda}{S_l^{(k)}} + \sum_{\forall l: j \in fanin(l)} \left(\frac{1-\lambda}{T_l^{(k)}} \sum_{\forall i \in fanout(l)} crit(j,i)^2 \right) + \sum_{\forall l: j \in fanout(l)} \left(\frac{1-\lambda}{T_l^{(k)}} crit(sl,j)^2 \right) \right\}^{-1}$$
$$\left\{ \sum_{\forall l: j \in Net_l} \frac{\lambda x_{cl}^{(k)}}{S_l^{(k)}} + \sum_{\forall l: j \in fanin(l)} \left[\frac{1-\lambda}{T_l^{(k)}} \sum_{\forall i \in fanout(l)} \left(crit(j,i)^2 x_i^{(k)} \right) \right] + \sum_{\forall l: j \in fanout(l)} \left[\frac{1-\lambda}{T_l^{(k)}} crit(sl,j)^2 x_{sl}^{(k)} \right] \right\}$$

To implement the Gauss-Seidel iteration, we need to update x_{cl}, S_l and T_l immediately after x_j moves from $x_j^{(k)}$ to $x_j^{(k+1)}$. From Section 3.2 in Chapter 3, we know x_{cl} and S_l can be updated in a constant time. Fortunately, T_l can also be updated in a constant time. This feature makes it possible to build a time-efficient Gauss-Seidel method for SOR timing-driven placement. For the sake of simplicity, we introduce five new variables U_l, V_l, P_l, Q_l and R_l, and let $U_l = \sum_{\forall i \in Net_l} x_i^2$, $V_l = \sum_{\forall i \in Net_l} x_i$, $P_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_i^2$, $Q_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_i$ and $R_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2$. Then, Equation A.1 can be transformed into:

$$T_l = \sqrt{\sum_{\forall i \in fanout(l)} [crit(sl,i)(x_i - x_{sl})]^2}$$
$$= \sqrt{\sum_{\forall i \in fanout(l)} [crit(sl,i)^2 x_i^2 - 2crit(sl,i)^2 x_i x_{sl} + crit(sl,i)^2 x_{sl}^2]}$$
$$= \sqrt{\sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_i^2 - \sum_{\forall i \in fanout(l)} 2crit(sl,i)^2 x_i x_{sl} + \sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_{sl}^2}$$
$$= \sqrt{\sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_i^2 - 2x_{sl} \sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_i + x_{sl}^2 \sum_{\forall i \in fanout(l)} crit(sl,i)^2}$$
$$= \sqrt{P_l - 2x_{sl}Q_l + x_{sl}^2 R_l} \qquad \text{(By the definition of } P_l,\ Q_l \text{ and } R_l\text{)}$$

Therefore, the Gauss-Seidel iteration for placement is defined as:

$$U_l = \sum_{\forall i \in Net_l} (x_i^{(0)})^2$$
$$V_l = \sum_{\forall i \in Net_l} x_i^{(0)}$$
$$P_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2 (x_i^{(0)})^2$$

$$Q_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_i^{(0)}$$

$$R_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2$$

$$x_{cl} = \frac{1}{k_l} V_l$$

$$S_l = \sqrt{U_l - k_l x_{cl}^2 + 1}$$

$$T_l = \sqrt{P_l - 2x_{sl} Q_l + x_{sl}^2 R_l}$$

$$x_j^{(k+1)} = \left\{ \sum_{\forall l: j \in Net_l} \frac{\lambda}{S_l} + \sum_{\forall l: j \in fanin(l)} \left(\frac{1-\lambda}{T_l} \sum_{\forall i \in fanout(l)} crit(j,i)^2 \right) + \sum_{\forall l: j \in fanout(l)} \left(\frac{1-\lambda}{T_l} crit(sl,j)^2 \right) \right\}^{-1}$$
$$\left\{ \sum_{\forall l: j \in Net_l} \frac{\lambda x_{cl}}{S_l} + \sum_{\forall l: j \in fanin(l)} \left[\frac{1-\lambda}{T_l} \sum_{\forall i \in fanout(l)} \left(crit(j,i)^2 x_i \right) \right] + \sum_{\forall l: j \in fanout(l)} \left[\frac{1-\lambda}{T_l} crit(sl,j)^2 x_{sl} \right] \right\}$$

(Equation A.7)

$$U_l = U_l + (x_j^{(k+1)})^2 - (x_j^{(k)})^2$$

$$V_l = V_l + x_j^{(k+1)} - x_j^{(k)}$$

$$P_l = P_l + crit(sl,j)^2 \left\{ (x_j^{(k+1)})^2 - (x_j^{(k)})^2 \right\} \qquad (\forall l : j \in fanout(l))$$

$$Q_l = Q_l + crit(sl,j)^2 \left\{ (x_j^{(k+1)}) - (x_j^{(k)}) \right\} \qquad (\forall l : j \in fanout(l))$$

To advance to SOR iteration from the Gauss-Seidel iteration, we introduce a relaxation factor ω into Equation A.7. The SOR iteration for placement is summarized as following:

$$U_l = \sum_{\forall i \in Net_l} (x_i^{(0)})^2$$

$$V_l = \sum_{\forall i \in Net_l} x_i^{(0)}$$

$$P_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2 (x_i^{(0)})^2$$

$$Q_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_i^{(0)}$$

$$R_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2$$

$$x_{cl} = \frac{1}{k_l} V_l$$

$$S_l = \sqrt{U_l - k_l x_{cl}^2 + 1}$$

$$T_l = \sqrt{P_l - 2x_{sl}Q_l + {x_{sl}}^2 R_l}$$

$$x_j^{(k+1)} = (1-\omega)x_j^{(k)} + \left\{ \sum_{\forall l: j \in Net_l} \frac{\lambda}{S_l} + \sum_{\forall l: j \in fanin(l)} \left(\frac{1-\lambda}{T_l} \sum_{\forall i \in fanout(l)} crit(j,i)^2 \right) + \sum_{\forall l: j \in fanout(l)} \left(\frac{1-\lambda}{T_l} crit(sl,j)^2 \right) \right\}^{-1}$$
$$\left\{ \sum_{\forall l: j \in Net_l} \frac{\lambda x_{cl}}{S_l} + \sum_{\forall l: j \in fanin(l)} \left[\frac{1-\lambda}{T_l} \sum_{\forall i \in fanout(l)} \left(crit(j,i)^2 x_i \right) \right] + \sum_{\forall l: j \in fanout(l)} \left[\frac{1-\lambda}{T_l} crit(sl,j)^2 x_{sl} \right] \right\} \cdot \omega$$

$$U_l = U_l + (x_j^{(k+1)})^2 - (x_j^{(k)})^2$$

$$V_l = V_l + x_j^{(k+1)} - x_j^{(k)}$$

$$P_l = P_l + crit(sl,j)^2 \left\{ (x_j^{(k+1)})^2 - (x_j^{(k)})^2 \right\} \qquad (\forall l : j \in fanout(l))$$

$$Q_l = Q_l + crit(sl,j)^2 \left\{ (x_j^{(k+1)}) - (x_j^{(k)}) \right\} \qquad (\forall l : j \in fanout(l))$$

Figure A.2 gives the pseudo code of SOR timing-driven placement as a summary of the whole procedure. The first line initializes the x-coordinates. The second line sorts the order that will be used to calculate all the x_is. The next four lines initialize the middle variables U_l, V_l, P_l, Q_l, R_l, the center of gravity x_{cl}, the Star+ estimate S_l, and time cost T_l for each net l. Within the while loop, each iteration computes new x, and updates U_l, V_l, P_l, Q_l, x_{cl}, S_l and T_l for each affected net l. The iterations terminate when i reaches the maximum number of iterations.

Initialize all x_i
Sort the order of calculating all the x_is
For each net l {

$$U_l = \sum_{\forall i \in Net_l} (x_i^{(0)})^2,\ V_l = \sum_{\forall i \in Net_l} x_i^{(0)},\ P_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2 (x_i^{(0)})^2$$

$$Q_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2 x_i^{(0)},\ R_l = \sum_{\forall i \in fanout(l)} crit(sl,i)^2$$

$$x_{cl} = \frac{1}{k_l} V_l,\ S_l = \sqrt{U_l - k_l x_{cl}^2 + 1},\ T_l = \sqrt{P_l - 2x_{sl} Q_l + x_{sl}^2 R_l}$$

}
$i = 0$

While $i <$ max {

For each block j {

$$x_j^{(k+1)} = (1-\omega) x_j^{(k)} + \left\{ \sum_{\forall l: j \in Net_l} \frac{\lambda}{S_l} + \sum_{\forall l: j \in fanin(l)} \left(\frac{1-\lambda}{T_l} \sum_{\forall i \in fanout(l)} crit(j,i)^2 \right) + \sum_{\forall l: j \in fanout(l)} \left(\frac{1-\lambda}{T_l} crit(sl,j)^2 \right) \right\}^{-1}$$

$$\left\{ \sum_{\forall l: j \in Net_l} \frac{\lambda x_{cl}}{S_l} + \sum_{\forall l: j \in fanin(l)} \left[\frac{1-\lambda}{T_l} \sum_{\forall i \in fanout(l)} \left(crit(j,i)^2 x_i \right) \right] + \sum_{\forall l: j \in fanout(l)} \left[\frac{1-\lambda}{T_l} crit(sl,j)^2 x_{sl} \right] \right\} \cdot \omega$$

For each net l that $j \in l$ {

$U_l = U_l + (x_j^{(k+1)})^2 - (x_j^{(k)})^2$
$V_l = V_l + x_j^{(k+1)} - x_j^{(k)}$
If $j \in fanout(l)$ {
$P_l = P_l + crit(sl,j)^2 \left\{ (x_j^{(k+1)})^2 - (x_j^{(k)})^2 \right\}$
$Q_l = Q_l + crit(sl,j)^2 \left\{ (x_j^{(k+1)}) - (x_j^{(k)}) \right\}$
}
$x_{cl} = \frac{1}{k_l} V_l$
$S_l = \sqrt{U_l - k_l x_{cl}^2 + 1}$
$T_l = \sqrt{P_l - 2x_{sl} Q_l + x_{sl}^2 R_l}$
}
}
} //end of while

Figure A.2: The pseudo-code of SOR timing-driven placement

Bibliography

[1] C. Dick, “FPGAs for Digital Communications,” DSP World Conference Proceedings, April 2000.

[2] J. Huie, P. D’Antonio, R. Pelt and B. Jentz, “Synthesizing FPGA Cores for Software Defined Radio,” Software Defined Radio Technical Conference and Product Exposition, Orlando, Florida, November 2003.

[3] L. Puker, “Paving Paths to Software Radio Design,” Spectrum Signal Processing, online document http://www.spectrumsignal.com/publications/csd.asp.

[4] R. Andraka and A. Berkun, “FPGAs Make a Radar Signal Processor on a Chip a Reality”, Proceedings of the 33rd Asilomar Conference on Signals, Systems and Computers, Monterey, CA, October 24-27, 1999.

[5] J. Hammes, A.P.W. Bohm, C. Ross, M. Chawathe, B. Draper and W. Najjar, “High Performance Image Processing on FPGAs,” Los Alamos Computer Science Institute Symposium, Santa Fe, NM, 2001.

[6] D. Dalton, V. Bessler, J. Griffiths, A. McCarthy, A. Vadher, R. O'Kane, R. Quigley and D. O'Connor, “APPLES: A Full Gate-Timing FPGA-Based Hardware Simulator,” book chapter in “Field-Programmable Logic and Applications,” Publisher: Springer Berlin / Heidelberg, ISBN 978-3-540-40822-2, pp. 1162-1165, September 2003.

[7] SF. Magruder, “Progress in understanding and using over–the–counter pharmaceuticals for syndromic surveillance of public health,” in Syndromic Surveillance, Reports from a National Conf., 2003.

[8] T. Wollinger, J. Guajardo and C. Paar, “Security on FPGAs: State-of-the-art implementations and attacks,” ACM Trans. Embedded Comput. Syst. 3(3): 534-574, 2004.

[9] M. Bednara, M. Daldrup, J. Teich, J. von zur Gathen and J. Shokrollahi, “Tradeoff analysis of FPGA based elliptic curve cryptography,” IEEE International Symposium

on Circuits and Systems, 2002.

[10] K. Shahookar and P. Mazumder, "Vlsi cell placement techniques," *in ACM Computing Surveys (CSUR)*, Volume 23 Issue 2, June 1991.

[11] J. Anderson and F. Najm, "Low-Power Programmable Routing Circuitry for FPGAs," in IEEE International Conference on Computer-Aided Design, pp. 602–609, November 2004.

[12] J. Rabaey, A. Chandrakasan, and B. Nikolic, "DIGITAL INTEGRATED CIRCUITS," Pearson Education Publishing Company, Inc, 2003.

[13] M. Hutton, K. Adibsamii, and A. Leaver, "Timing-Driven Placement for Hierarchical Programmable Logic Devices," Proc. of The 9th ACM/SIGDA Intl. Symposium on FPGAs, pp. 3–11, 2001.

[14] D. Huang and A. Kahng, "Partitioning-Based Standard-Cell Global Placement with an Exact Objective," ACM symp. on Phsyical Design, pp. 18–25, 1997.

[15] X. Bao and S. Areibi, "Constructive and Local Search Heuristic Techniques for FPGA Placement," in CCECE. Niagra Falls, Canada:IEEE, May 2004, pp. 505–508.

[16] K. Vorwerk and A. Kennings, "An Improved Multi-Level Framework for Force-Directed Placement," *DATE*: 902-907, 2005.

[17] J. Kleinhans, G. Sigl, F. Johannes, and K. Antreigh, "Gordian: VLSI Placement by Quadratic Programming and Slicing Optimization," *IEEE Trans. on CAD,* pp. 356-365, March 1991.

[18] G. Sigl, K. Doll, and F. Johannes, "Analytic Placement: A Linear or a Quadratic Function?" *DAC*, pp. 427-432, 1991.

[19] S. Adya and I. Markov, "Improving Min-cut Placement for VLSI Using Analytical Techniques," *Proc. IBM ACAS Conference, IBM ARL*, 55-62, February 2003.

[20] C. Albert, T. Chan, D. Huang, A. Kahng, I. Markov, P. Mulet, and K. Yan, "Faster Minimization of Linear Wirelength for Global Placement," *ACM Symp. on Physical Design*, pp. 4-11, 1997.

[21] A. Kennings and I. Markov, “Analytical minimization of half-perimeter wirelength,” *ASP-DAC*: 179-184, 2000.

[22] H. Eisenmann and F. Johannes, “Generic Global Placement and Floorplanning,” *DAC*: 269-274, 1998.

[23] B. Riess and G. Ettelt, “Speed: Fast and Efficient Timing Driven Placement,” *IEEE Int. Symp. on Circuits and Systems*, pp. 377-380, 1995.

[24] A. Srinivasan, “An Algorithm for Performance-Driven Initial Placement of Small Cell Ics,” *DAC*, pp. 636-639, 1991.

[25] N. Viswanathan and C. Chu, “FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model,” *ISPD*: 26-33, 2004.

[26] Q. Wang, D. Jariwala, J. Lillis, “A study of tighter lower bounds in LP relaxation based placement.,” *ACM Great Lakes Symposium on VLSI*: 498-502, 2005.

[27] 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.

[28] V. Betz, J. Rose, and A. Marquardt, “Architecture and CAD for Deep-Submicron FPGAs,” *Kluwer Academic Publishers*, ISBN 0-7923-8460-1, February 1999.

[29] S. Kirkpatrick, C. Gelatt, and M. Vecchi, “Optimization by Simulated Annealing,” *Science*, pp.671-680, May 1983.

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

[31] C. E. Cheng, “RISA: Accurate and Efficient Placement Routability Modeling,” *DAC*, pp. 690 – 695, 1994.

[32] J. Shewchuk, “An Introduction to the Conjugate Gradient Method Without the Agonizing Pain,” Technical Report, UMI Order Number: CS-94-125, Carnegie Mellon University, 1994.

[33] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Dongarra, V. Eijkhout, R. Pozo, C.

Romine, and H. Van der Vorst, "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods," *SIAM*, Philadelphia, PA, 1994.

[34] http://en.wikipedia.org/wiki/Successive_over-relaxation, "Successive over-relaxation"

[35] G. Grewal and M. Xu, "An Efficient Graph-Based Steiner Tree Heuristic for the Global Routing of Macro Cells," *IEEE Canadian Journal of Electrical and Computer Engineering*, Vol. 31, No. 4, 2006.

[36] P. Douglas, "VHDL," McGraw-Hill Professional, eISBN: 0071409548, May 2002.

[37] D. Thomas and P. Moorby, "The Verilog Hardware Description Language," Springer, ISBN 1402070896, 2002.

[38] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose, "Synthesis Methods for Field-Programmable Gate Arrays," *Proceedings of the IEEE*, pp. 1057 – 1083, July 1993.

[39] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," *Proceedings of the IEEE*, pp. 264 – 300, Feb. 1990.

[40] A. Dunlop and B. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Trans. on CAD*, pp. 92 – 98, Jan. 1985.

[41] J. Rose, W. Snelgrove, and Z. Vranesic, "ALTOR: An Automatic Standard Cell Layout Problem," *Canadian Conf. on VLSI*, pp. 169-173, 1985.

[42] W. Sun and C. Sechen, "Efficient and Effective Placement for Very Large Circuits," *IEEE Trans. on CAD*, pp. 349-359, March 1995.

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

[44] P. Du, G. Grewal, S. Areibi, and D. K. Banerji, "A Fast Hierarchical Approach to FPGA Placement," *ESA/VLSI,* pp. 497-503, 2004.

[45] C. Alpert, J. Huang, and A. Kahng, "Multilevel circuit partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 655

– 667, 1998.

[46] M. Romesis and J. Cong, "Performance-Driven Multi-Level Clustering with Application to Hierarchical FPGA Mapping," *DAC*, pp. 389-394, 2001.

[47] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, “Parallel algorithms for FPGA placement,” in the Great-Lakes Symposium on VLSI, (Chicago), March 2000.

[48] C. Fobel, G. Grewal, A. Morton, “A Comparison of Hardware-Accelerated Local Search Methods for FPGA Placement,” IEEE International Midwest Symposium on Circuits and Systems, August 5-8, 2007.

[49] C. Fobel, G. Grewal, and A. Morton, “A Hardware Accelerated Search Algorithm for FPGA Placement,” IEEE Canadian Conference on Electrical and Computer Engineering, Vancouver, April 11-16, 2007.

[50] Michael G. Wrighton and Andr´e M. Dehon. Hardware-assisted simulated annealing with application for fast FPGA placement. In FPGA ’03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays, pages 33–42, New York, NY, USA, 2003. ACM Press.

[51] Pak K. Chan and M. D. F. Schlag, “Parallel FPGA Placement with Symmetric Multiprocessors (SMPs) and Vector Functional Units, http://www.soe.ucsc.edu/~pak/chanschlagsmp.pdf.

[52] C. Fiduccia and R. Mattheyses, “A Linear Time Heuristic for Improving Network Partitions,” *Design Automation Conf.*, pp. 175-181, 1984.

[53] B. Kernighan and S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs,” *The Bell System Technical Journal*, pp. 49(2): 291-307, 1970.

[54] C. Sechen, “VLSI Placement and Global Routing Using Simulated Annealing,” *Kluwer Academic Publishers*, New York, 1988.

[55] J. Cong and J. Shinnerl, “Multilevel Optimization in VLSICAD,” *Kluwer Academic Publishers,* 2003.

[56] D. Mitra, R. Romeo, and A. Sangiovanni-Vincentelli, “Convergence and Finite-Time

Behavior of Simulated Annealing," Advances in Applied Probability, vol 18, No3, pp. 747-771, 1986.

[57] 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.

[58] M. Huang, F. Romeo, and A. Sangiovanni-Vincetelli, "An Efficient General Cooling Schedule for Simulated Annealing, "ICCAD, pp. 381-384, 1986.

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

[60] J. Lam, J. Delsome, and C. Sechen, "Performance of a New Annealing Schedule,," Proc. 25th DAC conference, pp. 306-311, 1988.

[61] V. Betz, "Architecture and CAD for Speed and Area Optimization of FPGAs," PhD Dissertation, University of Toronto, 1998.

[62] http://www.eecg.toronton.edu/~vaughn/challenge/challenge.html, "The FPGA Place-and-Route Challenge"

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

[64] 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.

[65] W. Sun and C. Sechen, "Efficient and Effective Placement for Very Large Circuits," IEEE Transactions on Computer-Aided Design Automation Conference, 14(3): 349-359, March 1995.

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

[67] C. J. Aplert, J. H. Juang, and A. B. Kahng, "Multilevel Circuit Partitioning," Proc. ACD/IEEE Design Automation Conference, pp. 530-533, 1997.

[68] S. Hur and J. Lillis, "Mongrel: Hybrid Techniques for Standard Cell Placement," IEEE/ACM International Conference on Computer-Aided Design, pp. 165-170, 2000.

[69] M. Wnag, X. Yang, and M. Sarrafzadeh, "Dragon2000: Standard-Cell Placement Tool for Large Indsutry Circuits," IEEE/ACM International Conference on Computer Aided Design, pp.260-263, 2000.

[70] T. Chan, J. Cong, T. Kong, and J. Shinnerl, "Multilevel Optimization for Large-scale Circuit Placement," Proc. IEEE International Conference on Computer Aided Design, San Jose, California, pp. 171-176, November 2000.

[71] C. Chang, J. Cong, D. Pan, and X. Yuan, "Multilevel Global Placement with Congestion Control," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 22, no. 4, pp. 395-409, July 2002.

[72] P. K. Chan and M. D. Schlag, "Parallel placement for Field-programmable gate arrays," 11th International AC/SIGDA Symposium on Filed Programmable Gate Arrays, (Monterey, California), February 2003.

[73] M. Grötschel, A. Martin, and R. Weismantel, "The Steiner Tree Packing Problem in VLSI-Design," *Mathematical Programming*, pp. 165-281, 1997.

[74] G. Grewal, T. Wilson, M. Xu, and D. Banerji, "Shrubbery: A New Algorithm for Quickly Growing High-Quality Steiner Trees," *17th International Conference on VLSI Design*, Mumbai, January 2004.

[75] L. Kou, G. Markowsky, and L. Berman, "A Fast Algorithm for Steiner Trees," *Acta Informatica*, Vol.15, pp.141-145, 1981.

[76] R. Karp, "Reducibility among combinatorial problems," *in Complexity of Computer Computations (E. Miller and J Thatcher, eds.), Plenum Press*, New York, pp. 85-103, 1972.

[77] G. Grewal, M. Xu, T. Wilson, and C. Obimbo, "An Approximate Solution for Steiner

Trees in Multicast Routing," *International Conference on Artificial Intelligence,* Las Vegas, June 2004.

[78] G. Grewal, M. Xu, T. Wilson, and X. Yu, "Generating Diverse Pools of Steiner Trees for VLSI Routing," *IEEE 18th Canadian Conference on Electrical and Computer Engineering*, Saskatoon, Canada, ref. 1568948681, May 2005.

[79] G. Grewal and M. Xu, "An Efficient Graph-Based Steiner Tree Heuristic for the Global Routing of Macro Cells," IEEE Canadian Journal of Electrical and Computer Engineering, Vol. 31, No. 4, 2006.

[80] J.B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, 7(1): 48-50, 1956.

[81] R.C. Prim, "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal*, Vol. 36, pp. 1389–1401, 1957.

[82] D. Chen, J. Cong and P. Pan, "FPGA Design Automation: A Survey," *Foundation and Trends in Electronic Design Automation*, Vol. 1, No. 3, 2006.

[83] B. M. Riess and G. G. Ettelt, "Speed: Fast and Efficient Timing Driven Placement," *Proc. ISCAS*, pp. 377-380, 1995.

[84] http://en.wikipedia.org/wiki/Student's_t-test, "Student's t-test"

[85] E.W. Dijkstra, "A Note on Two Problems in Connection with Graphs", *Numerical Mathematics*, 1:pp269-271, 1959.

[86] J. M. Kleinhans, G. Sigl, F. M. Johannes, "Gordian: A new global optimization/ rectangle dissection method for cell placement", Proc. of ICCAD, pp. 506-509, 1988.

[87] W. Mak and H. Li, "Placement for modern FPGAs," In Proceedings of the 2005 conference on Field Programmable Logic and Applications, 2005, pp. 789-784.

[88] M. Hutton and V. Betz, "FPGA synthesis and physical design," Electronic Design Automaton for Integrated Circuits Handbook," Taylor Francis CRC Press, vol. 1, ch. 13, 2006, pp. 13.1-12.2.

[89] Alisson V. Brito, Matthias Kuhnle, Michael Hubner, Jurgen Becker, Elmar U. K. Melcher, "Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC," isvlsi,pp.35-40, IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07), 2007.

[90] .D. Rao and M. Venkatesan, M, "An Efficient Reconfigurable Architecture and Implementation of Edge Detection Algorithm using Handle-C," In Proceedings of the international Conference on information Technology: Coding and Computing (Itcc'04) Volume 2, pp. 846, 2004.

[91] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, "A system for synthesizing optimized FPGA hardware from MATLAB," In Proceedings of the 2001 IEEE/ACM international Conference on Computer-Aided Design, 314-319.

[92] Michael A. Shanblatt, Brian Foulds, "A Simulink-to-FPGA Implementation Tool for Enhanced Design Flow," mse,pp.89-90, 2005 IEEE International Conference on Microelectronic Systems Education (MSE'05), 2005.

[93] P. Maidee, C. Ababei and K. Bazarga, "Timing-Driven Partitioning-Based Placement for Island Style FPGAs," IEEE Transaction Computer-Aided Design of Integrated Circuits and Systems, 24(3): 395-406, March 2005.

[94] G. Karypis and V. Kumar, "Multilevel Hypergraph Partitioning," In Design Automation Conference, 1997.

[95] P. Gopala Krishnan, X. Li and L. Pilleggi, " Architecture-Aware FPGA Placement Using Metric Embedding," In IEEE/ACM Designer Automation Conference, pp. 460-465, 2006.

[96] S. Ariebi, G. Grewal, D. Banerji and P. Du, "Hierarchical FPGA Placement," IEEE Canadian Journal of Electrical and Computer Engineering Vol. 32, No. 1, pp. 53-64, Winter 2007.

[97] T. Chan, J. Cong and K. Sze, "Multilevel generalized force-directed method for circuit placement," Proceedings of the 2005 international symposium on Physical

design, San Francisco, CA, April 2005.

[98] A. B. Kahng, S. Reda and Q. Wang, “Architecture and details of a high quality, large-scale analytical placer,” In Proceedings of the 2005 IEEE/ACM international Conference on Computer-Aided Design, 891-898, November, 2005.

[99] A. Ludwin, V. Betz and K. Padalia, "High-Quality, Deterministic Parallel Placement for FPGAs on Commodity Hardware," ACM / Sigda Int. Symp. on FPGAs, 2008, pp. 14 - 23.

[100] Y.T. Chang, Y.W. Chang, "An Architecture-Driven Detric for Simultaneous Placement and Global Routing for FPGAs," 37th Conference on Design Automation (DAC'00), 2000, pp.567-572.

[101] S. Nag and R. Rutenbar, “Performance-driven simultaneous place and route for row-based FPGAs,” In Proceedings of the 31st Annual Conference on Design Automation, 1994.

[102] R. Fletcher and C. M. Reeves, “Function minimization by conjugate gradients,” The Computer Journal 1964 7(2): 149-154.

[103] G. Chen, and J. Cong, "Simultaneous Timing Driven Clustering and Placement for FPGAs," *Springer Berlin / Heidelberg*, ISBN 978-3-540-22989-6, August 2004.

[104] Y. Xu and M. A. Khalid, “QPF: efficient quadratic placement for FPGAs,” International Conference on Field Programmable Logic and Applications, 2005, pp. 555-558.

[105] H. Etawil, S. Arebi, and A. Vannelli, “Attractor-repeller approach for global placement,” Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design, pp. 20–24, 1999.

[106] B. Hu and M. Marek-Sadowska, “Far: Fixed-points addition and relaxation based placement,” Proc. Intl. Symp. on Physical Design, pp. 161–166, 2002.

[107] J. Vygen, “Algorithms for large-scale flat placement,” Proc. ACM/IEEE Design Automation Conf., pp. 746–751, 1997.

[108] C. J. Alpert, T. F. Chan, D. J. H. Huang, A. B. Kahng, I. L. Markov, P. Mulet and K. Yan, "Faster Minimization of Linear Wire Length for Global Placement," Proc. ISPD '97, pp. 4-11.

[109] R. Baldick, A. Kahng, A. Kennings and I. Markov, "Function Smoothing with Applications to VLSI Layout," Proc. ASP-DAC `99, pp. 225-228.

[110] http://www.edn.com/article/CA6495296.html, "High noon for FPGAs: Low-cost-versus-high-end showdown".

[111] M. Xu, "Fast Heuristics for Solving Single- and Multiple-Objective Steiner Tree Problems in a Graph," M. Sc. Thesis, University of Guelph, 2004.

[112] W. Kahan, "Gauss-Seidel methods of solving large systems of linear equations," PhD thesis, University of Toronto, 1958.

[113] M. Xu, G. Grewal, S. Areibi, C. Obimbo and D. Banerji, "Near-Linear Wirelength Estimation for FPGA," IEEE 22nd Canadian Conference on Electrical and Computer Engineering, St. John's, Canada, May 2009, pp.1198-1203.