TIMING DRIVEN OPTIMIZATION PROBLEMS IN VLSI CAD

BY

ANMOL MATHUR

B. Tech., Indian Institute of Technology, Delhi, 1990
M.S., University of Illinois at Urbana-Champaign, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

TIMING DRIVEN OPTIMIZATION PROBLEMS IN VLSI CAD

Anmol Mathur, PhD.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1995
C. L. Liu, Advisor

We study several optimization problems that arise in the design of VLSI circuits, with the satisfaction of timing constraints as the primary objective. We focus on problems where the underlying architecture is regular. Field Programmable Gate Arrays (FPGAs), identical standard cell based architectures and WSI arrays of blocks having a regular structure are the major architectures that are regular. The regularity of these architectures allows us to use powerful graph theoretic techniques that would not be possible otherwise.

In this thesis we study problems at several different steps in the FPGA design flow. We address the problems of timing driven technology mapping and placement. We also study the problem of reconfiguring the placement of a circuit on an FPGA in order to tolerate faults in the logic blocks of the FPGA without significant degradation in the circuit delay. This problem is referred to as the *timing driven reconfiguration* problem. Timing driven reconfiguration can be used for off-line reconfiguration for yield enhancement and for on-line reconfiguration for fault tolerance. In a typical design flow, the design may be altered several times after the initial design cycle according to minor changes in the design specification either as a result of design debugging or as a result of changes in engineering requirements. In order to speed up the entire design process it is important to efficiently handle these *engineering changes*. We study the problem of incorporating engineering changes into a design in the presence of timing constraints. We propose a unified approach to solving the timing driven placement, reconfiguration and re-engineering problems using the concept of slack neighborhood graph.

To my parents

# Acknowledgements

Words cannot express my gratitude to Professor Dave Liu, my thesis advisor. Without his enthusiastic support, encouragement and guidance this thesis would not have been possible. I will always cherish his advise on matters academic and otherwise. I would like to thank the other members of my thesis committee: Professor Prithviraj Banerjee and Professor Rajesh Gupta. Their insightful comments have improved this thesis. Rajesh also introduced me to the area of embedded systems and encouraged me to work on the rate analysis problem.

Thanks are also due to Professor Jane Liu for giving me office space in her students' office. This allowed me to interact with some "non-CAD" people and helped me in broadening my perspective on computer science. I would also like to thank Professor Edward Reingold for a fruitful collaboration on the $k$-modal search problem. I would like to thank all the professors who taught me here at the University of Illinois for providing a very challenging and stimulating academic experience. I would like to specially mention Professor Herbert Edelsbrunner, Professor Doug West and Professor Janak Patel for teaching interesting courses.

I would like to thank all the members of the Design Automation group for providing a productive and cordial environment. A big "thank you" to all the group members over the years: Cheng Chang, Xiangfeng Chen, Ki-seok Chung, Sai-keung Dong, Tong Gao, Taewhan Kim, Ran Libeskind-Hadas, Unni Narayanan, Peichen Pan, Chaeryung Park, Srilata Raman, Prashant Saxena and Too-seng Tia. I would like to thank my office-mates Wuchun Feng and Rhan Ha for their cooperation and help. Many thanks to Joyce Woodworth and Kathy Johnson for efficient administrative help.

It is impossible for me to thank all the friends who made my stay in Urbana-Champaign great. I would like to thank Nimish Shah and Vivek Sarin for numerous academic and non-academic discussions. My apartment-mates over the years: Amber Roychowdhury, Srikanth

Thirumalai, George Tharakan, Anil Joshi, Vivek Sarin and Prashant Saxena deserve special thanks for their wonderful cooking and good humor. I would like to express my thanks to all my "baddy buddies": Nimish Shah, Vivek Sarin, Mallikarjun Shankar, G. Harikumar and K. Srinivas for the badminton games at IMPE. They often provided me with the inspiration at the end of an arduous day to go back and work at night.

Finally, I would like to thank my parents for their constant love and support. They instilled in me the respect for knowledge and the desire to explore new ideas that inspired me to go for a doctorate. I owe a debt of gratitude to my sisters and brothers-in-law Renu jiji and Sandeep jijaji, and Ritu and Rajneesh, for their affection. I am grateful to my lovely little nieces: Shubhi, Meha and Surabhi for their charming ways that kept me cheerful. My naniji, mamaji, mainji, Anant and Amit deserve special thanks for their love and hospitality.

It would be futile for me to try to express my gratitude to my wife, Richa. Her love and devotion made it all worthwhile.

# Table of Contents

## Chapter

# List of Figures

# Chapter 1

# Introduction

With advances in VLSI technology, the size of modules in integrated circuits is becoming smaller and the density of modules on a chip is increasing. Consequently, intra-module delays are becoming smaller and total delay in the circuit is being dominated by delays in the interconnections between the modules. The communication bounded nature of total circuit delay, along with more stringent performance requirements due to more aggressive designs, have made *timing driven layout* an important area of study. To meet the needs of a fast expanding electronics industry, high-performance chips must be designed in a short period. Accordingly, a design flow which incorporates timing analysis and verification into the physical design process is desirable. This motivates the development of tools which optimize area and performance simultaneously. This is in contrast to the traditional approach of focusing on area minimization only. Further, it is important to incorporate timing constraints in the initial phases of the design hierarchy. However, obtaining accurate estimates of the interconnection delays is hard at these design phases and it may be necessary to resort to an iterative approach that feeds back delay information from the layout of one iteration to the next iteration.

In this thesis we study several optimization problems with the satisfaction of timing constraints as the primary objective. We will focus on problems where the underlying architecture is regular. Field Programmable Gate Arrays (FPGAs), identical standard cell based architectures and WSI arrays of blocks having a regular structure are the major architectures that are regular. The regularity of these architectures allows us to use powerful graph theoretic techniques that would not be possible otherwise. FPGAs are becoming increasingly important in the design of Application Specific Integrated Circuits (ASICs) since they provide both large scale integration and user-programmability. User-programmability results in shorter turn-around time and lower manufacturing costs. As FPGA manufacturers are making constant technological improvements to achieve higher levels of integration and higher speeds, FPGAs are used more and more often in the final marketed designs of digital systems. This emphasizes the need for timing driven optimization in the implementation of circuits on FPGAs.

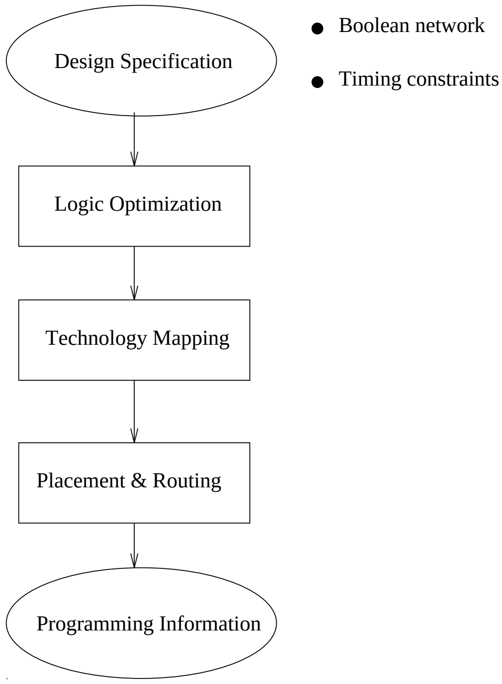


**Figure 1.1**: The typical design flow for implementing a circuit on an FPGA.

The typical design flow for implementing a circuit on an FPGA is shown in Fig. 1.1. The input specification consists of a description of the circuit to be implemented in the form of a Boolean network and the timing constraints that must be satisfied by the implementation of the circuit. The first step is a *technology independent* logic optimization phase in which transformations are applied to simplify the input functions. This is followed by the *technology mapping*, placement of the modules and routing of the interconnections, in that order.

In this thesis we study problems at several different steps in the FPGA design flow. We address the problems of timing driven technology mapping and placement. We also study the problem of reconfiguring the placement of a circuit on an FPGA in order to tolerate faults in the logic blocks of the FPGA without significant degradation in the circuit delay. This problem is referred to as the *timing driven reconfiguration* problem. Timing driven reconfiguration can be used for off-line reconfiguration for yield enhancement and for on-line reconfiguration for fault tolerance. In a typical design flow, the design may be altered several times after the initial design cycle according to minor changes in the design specification either as a result of design debugging or as a result of changes in engineering requirements. In order to speed up the entire design process it is important to efficiently handle these *engineering changes*. We study the problem of incorporating engineering changes into a design in the presence of timing constraints. We propose a unified approach to solving the timing driven placement, reconfiguration and re-engineering problems using the concept of slack neighborhood graph.

## 1.1 Assumptions and Models

In this section we discuss some assumptions that we make about the underlying architecture. The delay model used to compute/estimate the circuit delays is also discussed.

### 1.1.1 Regular Architectures

We focus on timing driven optimization problems in regular architectures. A regular architecture is assumed to have the following two properties:

- **Discreteness**: The slots onto which the modules in the circuit need to be mapped are assumed to be in distinct predefined places. This allows graph based modeling of the layout area.

- **Regularity**: It is assumed that the modules are to be placed in slots that are identical. This symmetry is crucial for several of the graph based algorithms we use to be computationally efficient.

Under these restrictions, the underlying architecture can be viewed as a two dimensional array of identical slots: $S = \{S_1, S_2, \ldots, S_m\}$, along with routing resources (routing channels, preplaced interconnections, programmable switches, etc.). Another characteristic common to all these architectures is that it is easy to estimate interconnection delay without performing detailed routing because of their regularity. Also, in the case of FPGAs, all the routing resources

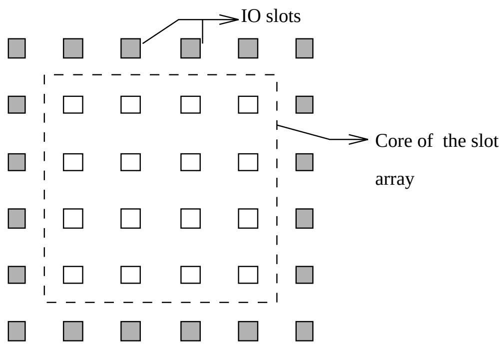


**Figure 1.2**: A regular architecture.

are already present, making the prediction of routing delay easier. We assume that the input and output modules are restricted to the periphery of the grid of slots, while the logic modules are placed in the *core* of the grid (see Fig. 1.2).

### 1.1.2 FPGA Architectures

Most FPGAs consist of a regular array of programmable logic blocks, also referred to as *configurable logic blocks* (CLBs) along with routing resources to interconnect the CLBs. The key feature distinguishing FPGAs from gate arrays is the fact that both the CLBs and the switches in the routing resources can be electrically programmed. This allows for easy customization in order to implement a specific function simply by programming the CLBs and interconnecting switches. This decreases the "time to market" for an ASIC implemented using an FPGA drastically by eliminating the metal layers mask generation step required to customize a gate array.

Typically FPGAs are classified on the basis of the nature of their CLBs and the technology used to implement the switches in the programmable interconnects. On the basis of the CLB architecture FPGAs can be classified into two categories:

- **Look-up Table based FPGAs**: The CLB in a look-up table based FPGA consists of a look-up table that stores the entire truth table of the boolean function to be implemented by the CLB. Such a CLB can implement any boolean function of up to $k$ inputs (where $2^k$ is the size of the look-up table). The use of a look-up table makes the delay of such a CLB independent of the function being implemented. The Xilinx [50] 3000 and 4000 series is the primary example of this type of FPGA. The Xilinx 4000 series CLB has two 4-input look-up tables capable of implementing two 4-input boolean functions. In addition each CLB has two flip-flops to facilitate the implementation of sequential logic. The CLB also has user programmable multiplexors that can be used to combine the outputs of the two look-up tables to implement more complex boolean functions.

- **Multiplexor based FPGAs**: Each logic block in such an architecture consists of a set of cascaded multiplexors. By appropriately controlling the multiplexor control signals, the logic block can implement several different boolean functions. Typically, such a logic block is not universal and may not be able to implement all the boolean functions of its inputs. The Actel FPGAs use multiplexor based logic blocks [1].

The classification of FPGAs based on the technology used to implement the programmable switches is as follows:

- **SRAM FPGAs**: In such FPGAs the switches in the interconnection resources are controlled by bits stored in a static random access memory (SRAM). Such a routing architecture coupled with a look-up table based CLB architecture is found in the Xilinx FPGAs. Since the look-up tables are also implemented using SRAM, all the configuration information in these FPGAs resides in SRAM making them easily reprogrammable. However, this reprogrammability comes at the cost of increased area and delay due to the SRAM cells and the relatively high capacitance routing switches (usually MOS pass transistors). In the Xilinx FPGAs there are horizontal and vertical routing channels between the rows and columns of CLBs. The logic blocks connect to the horizontal and vertical routing channels using a *connection block*. At the intersection of the horizontal and vertical channels there are *switch blocks* which allow signals to be switched between the tracks that meet at that switch block.

- **Antifuse FPGAs**: An antifuse is a two-terminal device that, when exposed to a high voltage, forms a permanent short circuit between the nodes on either side. Antifuse based FPGAs typically have rows of logic blocks with routing channels in between the rows. Further, each track in a routing channel has several distinct wire segments with antifuses in between. The inputs and outputs of the logic blocks connect to dedicated vertical segments. Vertical feedthroughs pass through the modules, serving as links between

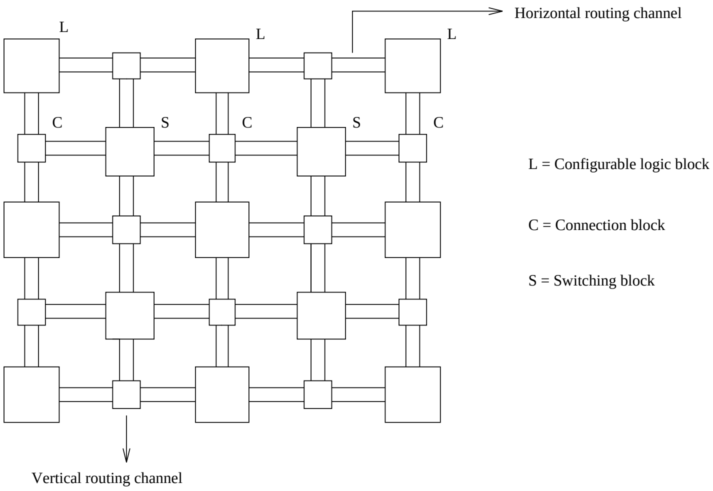


**Figure 1.3**: The Xilinx FPGA architecture.

the channels. There are antifuses at each intersection of a vertical and horizontal wire segment. The Actel FPGAs (see Fig. 1.4) has an antifuse based routing architecture and multiplexor based logic blocks. Since antifuses are small and have very low resistance and capacitance once programmed, antifuse based architectures have the advantage of having smaller die size and lower interconnection delays. However, they can be programmed only once.

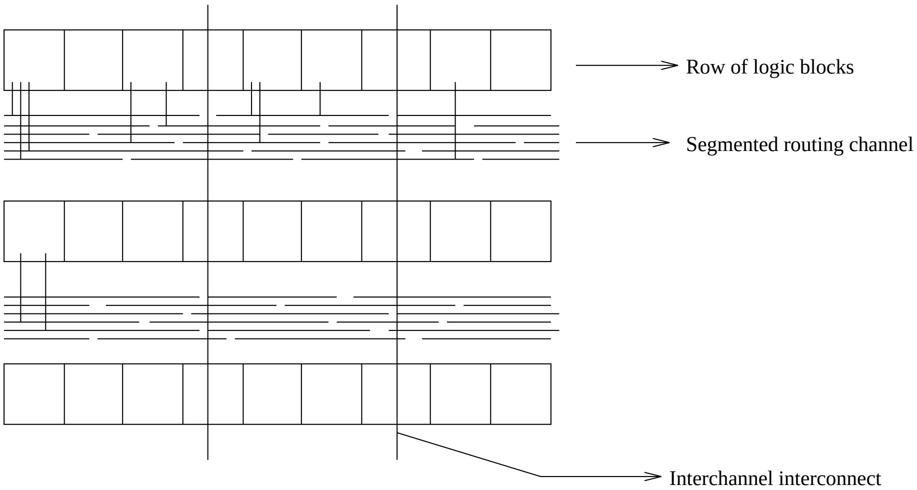


**Figure 1.4**: The Actel FPGA architecture.

In this thesis, we have implemented our FPGA related algorithms on the Xilinx 3000 series architecture.

### 1.1.3 Circuit Model

The circuit model provides an abstraction that captures the details of the circuit relevant to the problem being solved. For the problems addressed in this thesis a graph based modeling of circuits is adequate. Consider a circuit $C$, consisting of a set of modules $M = \{M_1, M_2, \ldots, M_n\}$, interconnected to implement the required functionality. Let

$$PI = \{x_1, x_2, \ldots, x_k\} \subset M$$

be the set of primary inputs to the circuit and

$$PO = \{f_1, f_2, \ldots, f_l\} \subset M$$

be the set of primary outputs. The topology of the circuit can is represented by a *circuit graph*, $G_C(V_C, E_C)$, where each module of $C$ is represented by a vertex in $V_C$, and an interconnection between two modules is represented by an edge in $E_C$ between the corresponding vertices (see Fig. 1.5). We will often use $M_i$ to denote the vertex in $V_C$ representing the $i$th module. Consider a connection between module $M_i$ and $M_j$. Typically either $M_i$ or $M_j$ will be the "source" that computes the value that is transmitted along the interconnection to the other module. Thus, the edge in the circuit graph representing the interconnection between $M_i$ and $M_j$ will be an edge directed from the source module to the sink module for the interconnection. A limitation of such a graph based model of the circuit is that it cannot capture certain aspects of multi-terminal nets: interconnections between modules having one source but more than one sinks. In the circuit graph based model, such a net is represented by separate edges from the source module to all the sinks. In particular, a multi-terminal net can be routed by constructing a Steiner tree on the terminals of the net. The sharing of wiring resources in such a Steiner tree cannot be modeled in the circuit graph. However, when we consider interconnection delays, the delay from the source of a multi-terminal net to its sinks can be different. Hence, for timing driven optimization problems the graph based model is adequate.

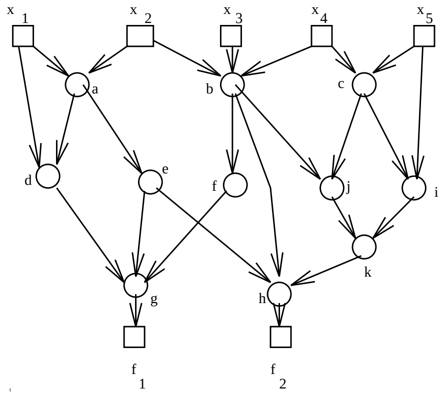


**Figure 1.5**: A circuit graph.

For a combinational circuit the graph $G_C$ is acyclic. Sequential circuits are usually decomposed into combinational parts by cutting them at the flip-flops and introducing *pseudo primary inputs* and *pseudo primary outputs*. Each combinational component is then handled separately by the timing driven optimization algorithms for the various problems of interest in this thesis.

Each vertex $v_i$ in the circuit graph is associated with a delay $d(v_i)$ equal to the delay of the module $M_i$ represented by $v_i$. Similarly, the propagation delay from a source module $M_i$ to a sink module $M_j$ is represented in the circuit graph as a delay $d(v_i, v_j)$ associated with the edge representing the connection between $M_i$ and $M_j$. Delay models will be discussed in more detail in the next subsection.

### 1.1.4 Delay Models

The model used to estimate interconnect delays plays an important role in the design of timing driven optimization algorithms. It dictates the assumptions which the algorithm uses to estimate delays. Several different delay models have been used in the literature.

- **Unit Delay Model**: This model assumes the delay between modules to be constant for all interconnections. This assumption is not realistic when communication delays form the dominant part of the total delay. The invalidity of this assumption for the case of FPGAs is further amplified by the scarcity of routing resources, which might force logically adjacent CLBs to be physically distant.

- **Nominal delay model**: This delay model allows different nets to have different delays, but requires that all the interconnections in the fanout of one node have the same delay. This model is more accurate than the unit delay model, but still does not present a totally correct picture, since it is possible for different interconnections in the fanout of a node to have vastly different delay values.

- **General delay model**: This model allows for different delays on interconnections, even if they fan out from the same node. This model captures the fact that different branches of the same net may be routed on paths with different delays. In the context of FPGAs, this is a more realistic delay model since different edges of the same net might be routed along paths having significantly different delays.

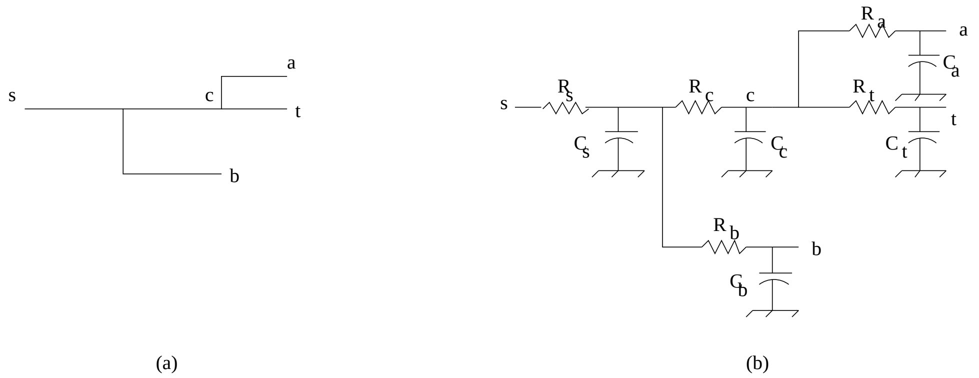


**Figure 1.6**: (a) A multi-terminal net. (b) The RC tree corresponding to the multi-terminal net.

Forcing all edges on a net to have the same delay (nominal delay model) or assuming that all edges in the network have the same delay (unit delay model) can result in erroneous identification of critical paths. We use the general delay model for delay estimation in all our algorithms. These different delay models require different methods of computing interconnection delays. For the unit delay model, any non-zero value of delay on all the edges suffices. Thus, the geometry of the interconnection can be completely ignored in the delay calculation. The interconnect delays in the nominal delay model can be computed by modeling a multi-terminal net as a lumped resistance and capacitance. On the other hand, the computation of delays in the general delay model require a more careful analysis of the distribution of resistance and capacitance in the net. If the exact topology of the net is known, it can be modeled as an RC tree, and the Penfield-Rubinstein formula [40] can be employed to compute the delays from the source to the different sinks. Thus, the net in Fig. 1.6 (a) can be modeled as the RC tree shown in Fig. 1.6 (b). The delay from the source to a sink, $t$, is given by the formula

$$T_t = \sum_i R_{it} C_i,$$

where

$R_{it}$ = resistance of the path common to the paths from the source to $i$ and $t$.

$C_i$ is the capacitance of node $i$ in the RC tree, and the sum ranges over all the node in the tree. The above formula gives an estimate of the Elmore delay [11] in the RC tree to the sink $t$.

We assume that the latest arriving signal at a module determines the output of the module. Thus, the longest path from a primary input to a primary output determines the delay of the circuit. Such a path is referred to as a *critical path*. A circuit may have several critical paths.

## 1.2 Organization of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 describes a new approach to technology mapping for FPGAs using the general delay model; Chapter 3 introduces the problem of timing driven relocation and introduces the concept of slack neighborhood graph to solve it; in Chapter 4 we show how timing driven relocation can be used as part of a new iterative algorithm for timing driven placement; Chapter 5 addresses the problem of timing driven reconfiguration for fault tolerance and yield enhancement in FPGAs; Chapter 6 discusses the timing driven placement re-engineering problem and our solution to it using the slack neighborhood graph approach; and finally in Chapter 7 we conclude with some directions for future research.

# Chapter 2

# Timing Driven Technology Mapping for FPGAs

In this chapter, we study the problem of technology mapping for look-up table based FPGAs, with delay minimization as the primary objective. We use the general delay model and allow different interconnections to have different delays even if they belong to the same net. An efficient algorithm for integrating technology mapping and layout, based on longest path computations and *biased* depth first search in directed acyclic graphs is proposed. An iterative approach is employed to use the delay information generated after placement and routing to direct technology mapping.

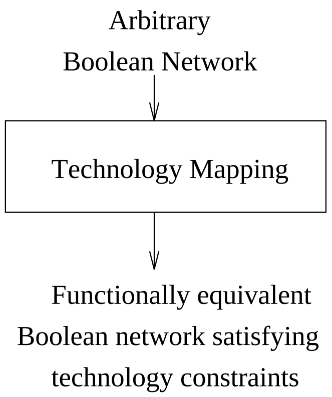


**Figure 2.1**: The technology mapping problem.

Traditionally, logic synthesis has been divided into two phases: a *technology independent phase* followed by *technology mapping*. The technology independent phase involves transformations on the boolean functions that are not dependent on the technology that is used to implement the functions. These transformations are aimed at reducing certain *technology independent* measures of the complexity of the boolean functions. One of the most commonly used technology independent metric for the area of the circuit required to implement a set of boolean functions is the total number of literals in the functions. Technology mapping takes the optimized boolean functions and maps them to the primitives available in the technology used to implement the boolean functions. The available primitives are typically defined by a library of logic cells that can be used to implement the functions. The technology mapping problem for look-up table based FPGAs differs from the traditional technology mapping problem because there is no explicit library of logic cells, instead there is an implicitly defined library due to constraints on the number of inputs and outputs of the CLBs. Thus, the technology mapping problem for look-up table based FPGAs involves mapping the given boolean functions to a functionally equivalent network consisting of CLBs. In traditional technology mapping, each cell in the library is associated with several attributes such as area, delay and power consumed. The technology mapping algorithm aims at finding a realization of the optimized boolean functions that minimizes a cost function such as the total area of the circuit, the critical path delay, the total power consumed, or some combination thereof. In technology mapping for look-up table based FPGAs all the "cells" are implemented using an SRAM based look-up table, hence the area, delay and power consumption of all the "cells" is identical. Thus, the traditional objectives in technology mapping are not relevant for FPGAs.

Much of the early work on design automation for FPGAs focussed on technology mapping for area minimization, [34] [12]. Recently more effort has been directed towards algorithms for FPGA technology mapping which minimize the delay in the critical path. The *Chortle-d* technology mapper [13] attempts to minimize the depth of the clustered circuit by using bin packing heuristics geared towards reduction of the number of levels. *DAG-map* [7] uses Lawler's labeling algorithm to produce a low depth implementation of the given boolean network. However, this algorithm is optimal only for DAGs which are monotone with respect to the number of distinct inputs entering a logic cone. The *Flow-Map* mapper [8] uses a modified version of Lawler's labeling algorithm, computing the labels by finding feasible cuts in a network flow formulation.

This algorithm produces a minimum depth implementation of the boolean network and is applicable to arbitrary networks. *TechMap* [43] uses clique partitioning in a compatibility graph to cluster the given boolean network. However, all these algorithms attempt to minimize the number of levels in the clustered circuit, without considering the actual delays which might arise when the clustered circuit is placed and routed. The underlying assumption for all these approaches is that the net delays between CLBs in the final design are relatively uniform. The scarcity of routing resources and the fact that there are substantial differences in delays among the available routing resources, make the above assumption unrealistic. In fact, our results indicate that often clusterings with larger number of levels, but a better "distribution" of the delays in the various paths result in lower critical path delay in the actual implementation.

Actual delays are taken into consideration in the technology mapping phase in the modified *mis-pga* mapper [35]. It uses a two phase approach: the first phase involves delay optimization during logic synthesis before placement, and the second uses logic resynthesis during a timing-driven placement phase. More recently, *Edge-Map* [49] has used an iterative approach similar to ours, along with an optimal clustering algorithm in the general delay model.

We study the problem of technology mapping for FPGAs, with delay minimization as the primary objective. The delay model used is different from the one used in all the previous approaches to solve this problem. We allow different edges to have different delays even if they belong to the same net. In the context of FPGAs, this is a more realistic delay model since different edges of the same net might be routed along paths having significantly different delays. Thus, forcing all edges on a net to have the same delay (nominal delay model) or assuming that all edges in the network have the same delay (unit delay model) can result in erroneous identification of critical paths. An efficient algorithm for integrating technology mapping and layout, based on longest path computations and *biased* depth first search in directed acyclic graphs is proposed. An iterative approach is employed to use the delay information generated after placement and routing to direct technology mapping.

## 2.1 Problem Formulation

The input to the technology mapping is a directed acyclic graph (DAG), $G(V, E)$, representing the boolean functions to be implemented using an FPGA. Since each vertex in the DAG rep-

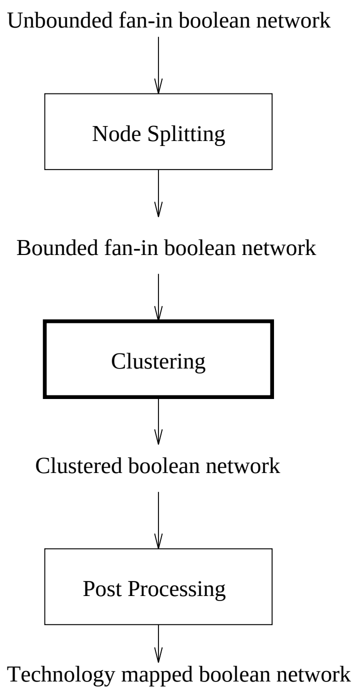


**Figure 2.2**: Typical steps in technology mapping for FPGAs.

resents a boolean function, we will often refer to it as a *boolean network*. We assume that the circuits being implemented are combinational. For a sequential circuit, each of the combinational components generated by cutting the circuit at the latches is mapped separately. Since we are interested in the timing driven technology mapping problem, we assume that each edge $(u, v)$ in the boolean network is associated with a delay $d(u, v)$ that is an estimate of the actual delay of the interconnection between the nodes represented by the vertices $u$ and $v$.

The technology mapping phase for implementing a given set of boolean functions involves finding a functionally equivalent boolean network that satisfies the constraints of FPGA technology. We consider lookup-table based FPGAs, which use static RAMs to implement the configurable logic blocks. The Xilinx FPGA series is an example of such FPGAs [50]. The technology constraints in such FPGAs arise mainly from the architecture of the logic blocks and are of the form:

1. The number of distinct inputs to a logic block is at most $k$ ($k$ is 5 for the Xilinx 3000 series architecture).

2. The number of distinct outputs of a logic block is at most $k'$.

3. If more than one boolean function is being implemented in a logic block then there are constraints on the maximum number of inputs that can be shared between the functions.

For most FPGAs currently being marketed, $k$ ranges between 4 and 9, and $k'$ is 1 or 2. Since we have implemented our algorithm for the Xilinx 3000 series architecture, we will assume that each CLB has 5 inputs. Although each CLB in the Xilinx 3000 series architecture can implement two boolean functions if they satisfy some input sharing constraints, for the simplicity of exposition we will assume that each CLB has only one output function. Thus, each node in the technology mapped boolean network must have at most 5 inputs and one output.

Technology mapping for look-up table based FPGAs typically involves the steps shown in Fig. 2.2. Since the initial boolean network being technology mapped may have nodes with more than 5 inputs, it is necessary to decompose such nodes into a boolean network of nodes with at most 5 inputs. This step is referred to as *node splitting*. The output of node splitting is a *feasible boolean network* in which each node has at most 5 inputs. The key step in technology mapping is *clustering*. It involves grouping of nodes in the boolean network into clusters each of which satisfies the FPGA technology constraints. Clustering is followed by a post-processing step that typically tries to exploit additional features provided by the FPGA architecture, such as the capability of implementing two boolean functions in one CLB provided they satisfy some input sharing constraints. We focus on the clustering step of the technology mapping process.

The clustering step can be guided by one or more of the following goals :

- Minimizing the number of CLBs or equivalently, minimizing the number of components in the clustered boolean network, resulting in lower area requirement.
- Minimizing the maximum delay along a path from an input to an output in the clustered boolean network.
- Producing an easily routable mapping.

### 2.1.1 Definitions

In order to clearly define the timing driven clustering problem for FPGA technology mapping, we need the following definitions:

- **Cone:** Given a node $v$, we define $cone(v)$ to be the subgraph of $G$ induced by the set of nodes consisting of $v$ and all its predecessors in $G$ [1]. We will use the terms cone and *logic cone* interchangeably.

- **Cluster:** A cluster is a set of nodes $C \subset V$, such that for any nodes $u, v \in C$ all nodes on the paths between $u$ and $v$ are also in $C$.

- **Feasible Cluster:** A cluster $C$ is said to be *feasible* if it can be mapped to a logic block without violating any of the technology constraints mentioned earlier. Thus, for the Xilinx 3000 series architecture a cluster is feasible if it has at most 5 incoming signals. Further, a feasible cluster must have only one node for which none of its successors is in the cluster. This node, $v$, is the *output node* of the cluster. The cluster $C$ is always contained in $cone(v)$ and the boolean function implemented by the CLB corresponding to the cluster is the function computed by the boolean network corresponding to the cluster at node $v$. Note that a feasible cluster may have other outgoing signals, but they are not computed by the CLB to which the cluster gets mapped.

- **Cut:** Given the cone of a node $v$, a cut is defined to be a partition $(X, \bar{X})$ of $cone(v)$ such that $v \in \bar{X}$ and $X, \bar{X}$ are both clusters.

- **$k$-feasible Cut:** A cut $(X, \bar{X})$ is said to be $k$-feasible if there are at most $k$ distinct signals crossing from $X$ to $\bar{X}$.

It should be noted that in a 5-feasible cut of the cone of a node, $v$, the cluster containing $v$ is a feasible cluster with respect to the Xilinx 3000 series architecture constraints. Some of the terms used are illustrated in Fig. 2.3.

A *valid clustering* of a DAG is a set of clusters, $C = \{C_1, C_2, \cdots, C_n\}$, such that

1. All the clusters are feasible.

2. Each node of the DAG is contained in at least one cluster.

3. For each input (that is not a primary input), $y$, to a cluster $C_i$, there must be a cluster $C_j \in C$ that has $y$ as an output.

[1] $u$ is a predecessor of $v$ if there is a directed path from $u$ to $v$.

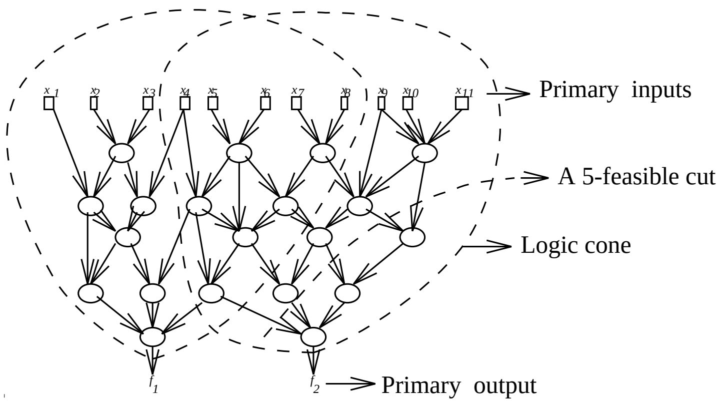


**Figure 2.3**: A boolean network illustrating some terms.

In a valid clustering of $cone(v)$, the cluster covering $v$ is referred to as the *apical cluster*. Notice that a 5-feasible cut of $cone(v)$ generates a feasible apical cluster covering $v$.

For a given clustering $C$, an edge $(u, v)$ is *visible* with respect to a path $P$ in the clustered network, if $u \in C_i$ and $v \in C_j$ and $C_i$ and $C_j$ are adjacent clusters on the path $P$. Otherwise, $(u, v)$ is *invisible* with respect to path $P$ (see Fig. 2.4).

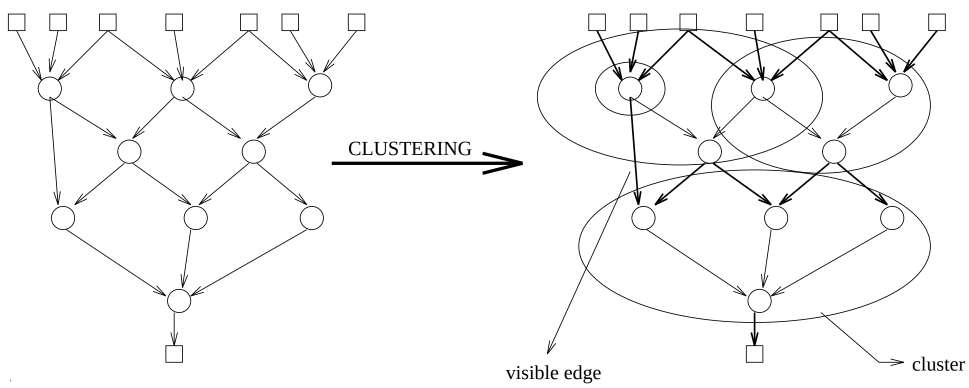


**Figure 2.4**: The DAG corresponding to a logic cone and a valid clustering.

It is important to note that we do not require that the clusters in a valid clustering be pairwise disjoint. Non-overlapped clustering can be arbitrarily bad with respect to both the number of clusters used and the delay in the critical path. Thus, unlike many traditional clustering algorithms, FPGA technology mapping algorithms produce overlapping clusters. The overlap between clusters entails duplication of logic, but this is not a severe penalty in FPGAs since the logic blocks are not constrained by the amount of logic (as measured by the number of nodes), but by the number of inputs.

The result of the technology mapping process is a valid clustering of the given boolean network. Subsequently, each cluster is mapped to a logic block and the logic blocks are interconnected according to the edges between the clusters.

For a path in the clustered network, we define the delay of the path to be the sum of the delays of the *visible* edges in the path. In the implementation of the clustered network, both the interconnections and the logic blocks contribute to the delay of any path. However, the delay due to the logic blocks can be added to the delays on the connections in the fanout of the logic block without altering the delay of any path passing through the logic block. This justifies the assumption of assigning delays only to edges in the boolean network. A *critical path* in the clustered network is a path with maximum delay. The *timing driven clustering* problem can be defined more precisely as follows:

*Given a feasible boolean network with estimates of the delays of the interconnections between the nodes, find a valid clustering that minimizes the critical path delay in the clustered network.*

Since placement and routing are done after the circuit to be implemented has been technology mapped, the delay estimates used in timing driven technology mapping are necessarily inaccurate. We attempt to overcome this fundamental limitation by integrating technology mapping and placement and routing in an iterative loop. The accurate delay information available after placement and routing is used to re-compute the edge delays after each iteration.

## 2.2 An Overview of Our Approach

The labeling based approaches for minimizing the number of levels [7], [8] make use of the fact that the optimal clustering of the cone of a node can be constructed using some of the optimal clusterings of its predecessors. However, in the general delay model, sub-optimal clusterings at some of the predecessors might be required to prevent edges with large delay from becoming visible. This causes the dynamic programming formulation for delay minimization in the unit delay model to break down for the general delay model. We now outline our heuristic for finding “good” clusterings in the general delay model.

### 2.2.1 The Iterative Improvement Algorithm

Since our technology mapping algorithm does not make any assumptions about the delay values assigned to the edges, it can be coupled with a placement and routing algorithm to form an iterative technology mapping algorithm. Getting good estimates of delays is the major bottleneck in using the general delay model in the technology mapping stage. One possible solution is to have an iterative algorithm which feeds back the delay information obtained after placement and routing to the technology mapping phase of the next iteration.

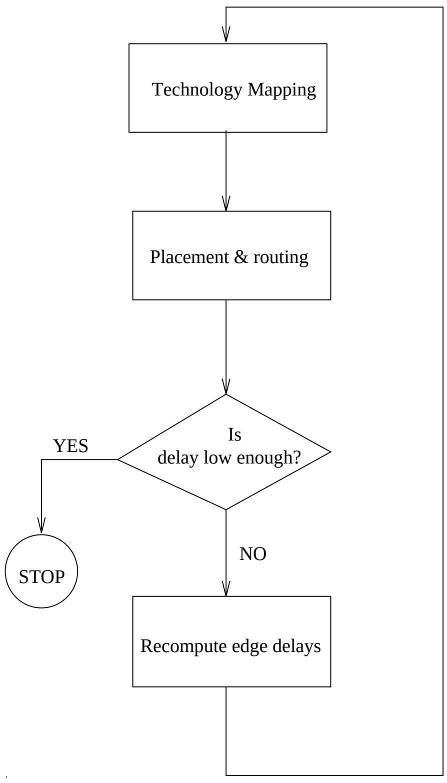


**Figure 2.5**: The flowchart for the iterative technology mapping algorithm.

The iterative algorithm starts with some delay estimates as the edge delays. In our experiments we found that the fanout of a node is a good estimate of the delay on the edges fanning out from that node. Our technology mapping algorithm is used to obtain a technology mapped circuit which is then placed and routed. From the placed and routed circuit we recompute the edge delays. A strategy for recomputing the edge delays is to assume the delay of an edge invisible in a mapping to be zero and the delay of the visible edges to be their actual propagation delays. In any iteration, the technology mapper then uses a (weighted) mean of the delay values assigned to an edge in all the previous iterations as the delay of the edge. A flowchart for such an algorithm is shown in Fig. 2.5. The input to our algorithm is a boolean network and the output is a completely placed and routed design. The experimental results we obtain indicate that such an approach is quite effective.

### 2.2.2 Biased Depth First Clustering

Given an edge weighted, 5-feasible boolean network $G(V, E)$, we want to generate a clustering which minimizes the maximum value of the sum of the delays of the visible edges on any path from a primary input to a primary output. Depth first search starting from the primary outputs seems to be a natural way to achieve a low delay clustering, since it tries to grow clusters *along* a path as much as possible. However, a naïve depth first search cannot be expected to perform well because it does not use the delay information in an intelligent manner.

Our algorithm employs the idea of a *biased depth first search* to produce a low delay clustering. We use the delay information to compute node weights which bias the depth first search to explore paths with greater delays first. The main motivation for this is that exploring high delay paths preferentially will cause "long" clusters to be generated along these paths, resulting in fewer visible edges on these paths. This should insure that the delay along long paths is not too large in the clustered network. Implicit in this approach is the following assumption about the relation between the critical paths in the original network and the clustered network:

- The critical paths in the original network and in any clustering of the network are the same.

Since the clusters are constrained by the number of inputs, the above assumption translates into the requirement that the number of distinct inputs to nodes on a path in the boolean network be proportional to the length of the path. This requirement is satisfied for most real circuits, although it is possible to contrive boolean networks having long paths with very few distinct inputs. Thus, biased depth first search should perform well on networks which satisfy this assumption.

The algorithm performs technology mapping for the cone of each primary output separately. For each logic cone, node weights are computed based on the edge delays; a biased depth first

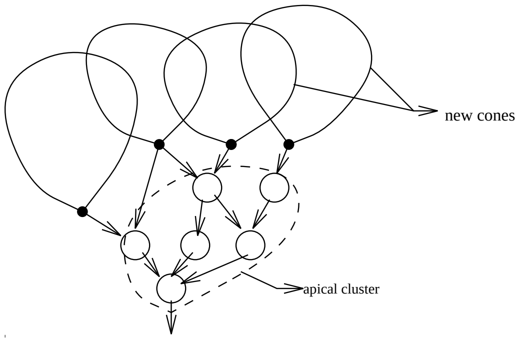


**Figure 2.6**: An apical cluster and the logic cones generated on deleting the apical cluster.

search is used to generate a topological ordering of the logic cone; densities of crossing edges are computed and a good feasible cut found to generate the apical cluster. Deleting the apical cluster from the logic cone results in a collection of (possibly overlapping) cones each of which is clustered recursively (see Fig. 2.6). However, the biasing weights and biased topological order do not need to be recomputed for the new cones.

Here is the pseudocode describing the clustering algorithm at the top level:

**for** each primary output in the boolean network {

Label the logic cone of the primary output;

Sort_array = *topological_sort (logic cone)*;

*Compute_biasing_weights(sort_array)*;

/* Now we do the biased DFS */

Biased_sort_array = *topological_sort (weighted logic cone)*;

/* Next we find the apical cluster */

*Find_apical_cluster (biased_sort_array)*;

Recursively cluster logic cones generated by deleting
the apical cluster;

}

In the next section we describe the details of the algorithm for finding a good apical cluster.

## 2.3 Finding an Apical Cluster

Our algorithm for finding a good apical cluster is guided by the edge delays in the boolean network. The main steps in our algorithm are shown in Fig. 2.7. The basic idea is to find an apical cluster that covers more nodes in the critical paths. The intuition behind this being that such an approach will result in a clustering that has fewer visible edges on the critical paths, resulting in lower critical path delay in the clustered boolean network. In order to find such an apical cluster without looking at an inordinately large number of cuts, we use a linear ordering of the nodes in the cone being clustered to find a good cut.

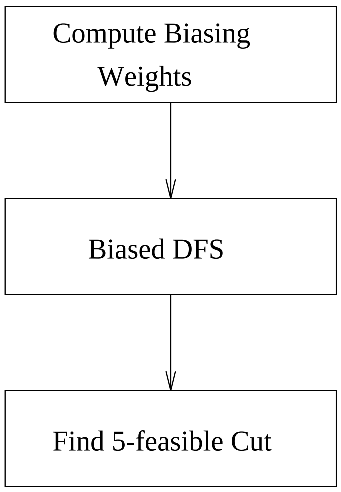


**Figure 2.7**: Steps in finding a good apical cluster.

### 2.3.1 Biasing Weights and DFS

The biasing weight of a node in the DAG is the length of the longest path passing through the node. These weights can be computed for all the nodes in the DAG in linear time using a dynamic programming based algorithm. An efficient procedure for computing the biasing weights is based on the fact that the longest path passing through a node in a DAG is obtained by concatenating the longest path *ending at* the node and the longest path *starting at* the node. Consequently, the biasing weights can be computed efficiently using dynamic programming. The steps involved in computing the weights are:

1. Compute the delay of the longest path *starting at* $v$, for each node $v$ in the cone.

2. Compute the delay of the longest path *ending at* $v$, for each node $v$ in the cone.

3. The sum of the two delays computed in the above steps gives $w(v)$.

Let $l(v)$ represent the length of the longest path starting at $v$. The computation in Step (1) (and by symmetry Step (2)) can be done efficiently using dynamic programming by using the following relation:

$$l(v) = \max_{u:(v,u)\in E} \{l(u) + d(v,u)\}.$$

Thus, by topologically sorting the nodes in the cone, the $l$ function can be computed in reverse topological order (starting from the sink and moving towards the sources) in linear time. Fig. 2.8 shows the logic cone for output function $f_1$ in the boolean network shown in Fig. 2.3 along with the biasing weights computed using the above algorithm.

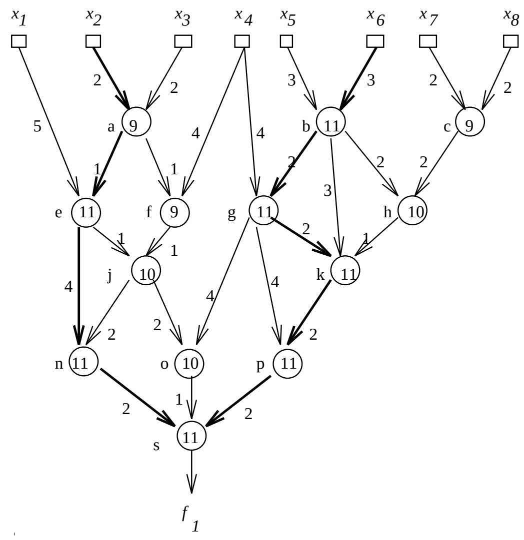


**Figure 2.8**: A logic cone with biasing weights. The longest paths through the nodes $a$ and $b$ are shown by the thick edges.

Once the biasing weights have been computed, we use a biased depth first search to topologically sort the nodes again. The biased depth first search uses the primary inputs as the starting nodes and the neighbors of a node are explored in order of decreasing weight. Once a node has been fully explored and the DFS is backtracking out of it, we place it at the beginning of the list of topologically sorted nodes. Since, the DFS explores the neighbors with higher weight first, the topologically ordered list has higher weight nodes towards the beginning. The linear ordering of the nodes obtained by arranging the nodes in the topologically sorted list from left to right has the following properties:

- All the edges are directed from right to left.
- Given any two nodes that are not connected by an edge, the node with higher weight is to the left of the node with lower weight.

Thus, the linear ordering of the nodes produced by biased DFS has the nodes that are contained in the more critical paths towards the left (see Fig. 2.9). This linear ordering is exploited to find a good cut defining the apical cluster efficiently.

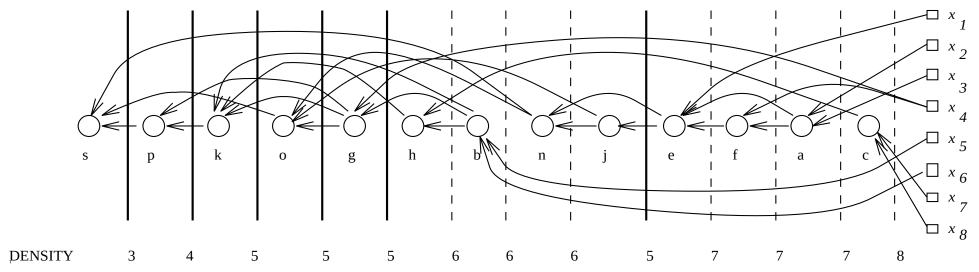


**Figure 2.9**: Topological order for the cone in Fig. 2.8 along with cut densities. The 5-feasible cuts are indicated by the dark lines and the infeasible ones by dotted lines.

## 2.4 Finding a Good Cut

Each position $i$ in the linear ordering produced by the biased DFS defines a cut of the cone being clustered. The first $i$ nodes in the linear order form the apical cluster corresponding to the cut in position $i$ (see Fig. 2.10). We restrict ourselves to the cuts induced by the linear ordering, and find a "good" apical cluster by finding a "good" cut that is 5-feasible.

The *density* at a position $i$ in the linear ordering is defined to be the number of distinct signals crossing position $i$. Thus, edges corresponding to different fan outs of the same signal are counted only once. Let $v_1, v_2, \ldots, v_n$ be the linear arrangement of the nodes from left to right. Then,

$$density(i+1) = density(i) + NumEnd(i) - 1,$$

where $NumEnd(i)$ represents the number of inputs (not necessarily primary inputs) which have $v_i$ as the leftmost node to which they fan out. $NumEnd(i)$ can be computed efficiently by looking at all the fanouts of each node in the network and incrementing the $NumEnd$ count for the node in the fanout which is leftmost in the topological order. The $density$ values can then be computed in a single pass over the topologically sorted nodes using the relation between $density(i)$ and $density(i+1)$ given above. This entire phase takes time linear in the size of the boolean network.

The cut corresponding to position $i$ in the linear ordering is feasible if and only if $density(i)$ is at most 5. Among all the feasible cuts in the linear ordering, we need to select one "good" cut to generate the apical cluster. We use two measures of goodness of a cut to make this choice:

- **Weight:** The weight of a cut is defined as the sum of the delays of all the edges that cross the cut. A cut with low weight is good since it makes low delay edges visible, thus reducing the possibility of large delays in the critical paths in the clustered network.

- **Volume:** The volume of a cut is equal to the number of nodes in the apical cluster corresponding to the cut. Thus, the volume of a cut in position $i$ is equal to $i$. A cut with high volume is good since it increases the likelihood of generating a clustering with fewer clusters.

We choose the cut that maximizes a weighted combination of these two metrics of the form

$$\alpha \cdot volume - \beta \cdot weight.$$

Fig. 2.10 shows the weights and volumes of the 5-feasible cuts in the linear ordering.

Since our algorithm examines the densities at all positions in the linear arrangement produced by the biased topological sort, it can "climb" over some "hills" in the density function. This "hill climbing" effect is visible in Fig. 2.9, where the increase in density after node $h$ to a value larger than 5 does not prevent the algorithm from finding the feasible cut after node $j$.

Once the apical cluster is found, we remove the nodes in the apical cluster and cluster the new cones produced recursively using the same algorithm.

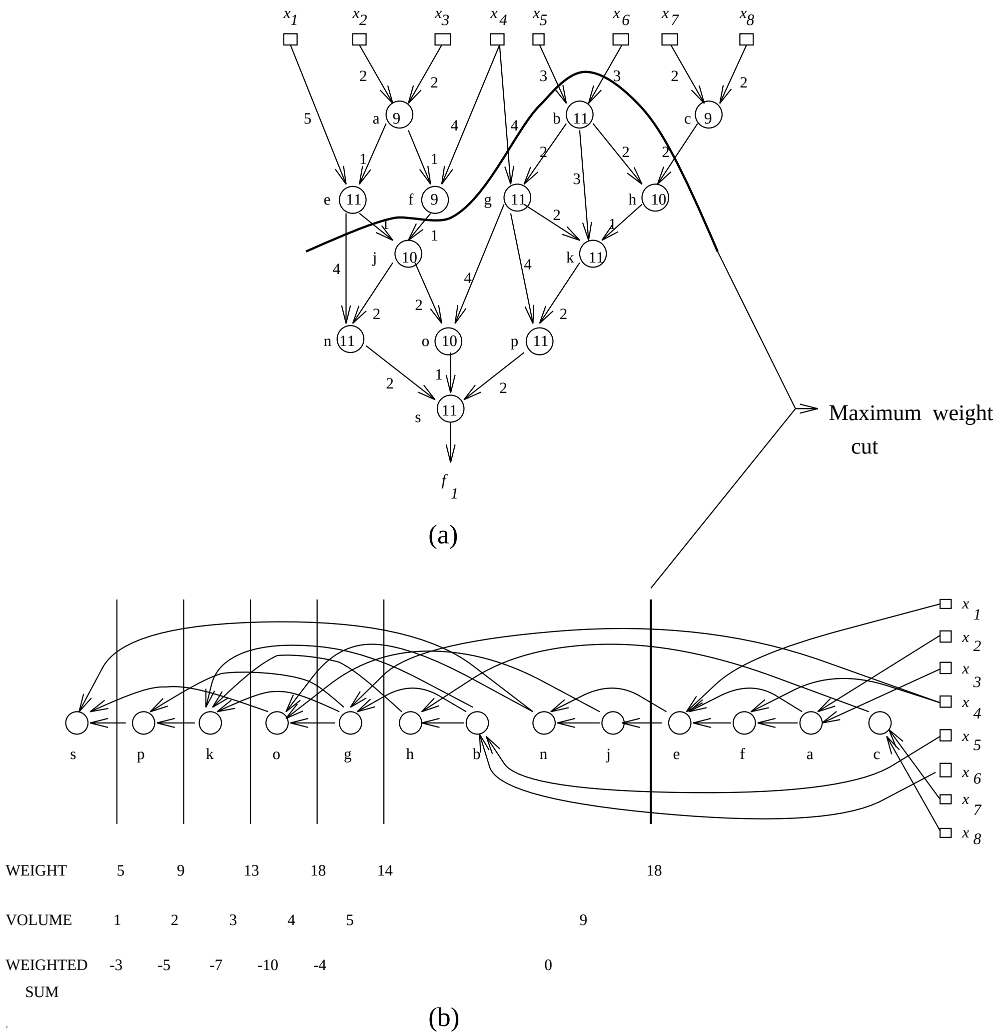


**Figure 2.10**: (a) A logic cone with biasing weight. (b) The topologically sorted linear ordering of the logic cone generated by biased DFS. The weights, volumes and weighted sums (assuming $\alpha = 2$, $\beta = 1$) for all the 5-feasible cuts are shown. The cut that maximizes the weighted sum is shown in (a).

## 2.5 Experimental Results

We have implemented the biased depth first search clustering algorithm and tested it on a set of MCNC benchmark examples. The implementation of our timing driven clustering algorithm was integrated as part of the *misII* program and can be invoked using the command *bias_clus* command. We used a version of *misII* that had the *flowmap*, *dagmap* and *flowpack* commands available.

The following sequence of steps was used in generating the experimental results:

1. Read in the benchmark boolean network using *read_blif* or *read_pla* after invoking *misII.*

2. Transform the boolean network into a simple and-or circuit using *tech_decomp -a 1000 -o 1000.*

3. Transform the and-or circuit into a 2-feasible boolean network using *tech_decomp -a 2 -o 2.*

4. Use *bias_clus* to cluster the 2-feasible network.

5. Use *write_xnf* command to output the clustered network in the Xilinx netlist format.

6. Transform the *.xnf* file produced to a *.lca* file using the sequence of Xilinx XACT commands *xnfmap* and *map2lca.*

7. Use the Xilinx automatic place and route tool *apr* to place and route the design obtained as a result of the previous step. We used the command
   `apr -i 0.0`
   to produce a path delay table in the report generated by *apr.* The path delay table contains the delays (in nanoseconds) for all the paths from primary inputs to primary outputs. This table was used to compute the worst case delay from primary input to primary output.

Table 2.1 shows a comparison between our results and the results in [14] where *flowmap* was used to perform the technology mapping and then *apr* was used to do the placement and routing. The column *CLBs* is the number of CLBs in the technology mapped circuit; the column *depth* is the maximum number of CLBs on a path in the clustered circuit, and the column *delay* is the

| Circuit | flowmap | | | bias_clus | | |
|---|---|---|---|---|---|---|
| | CLBs | depth | delay | CLBs | depth | delay |
| cordic | 17 | 4 | 58.3 | 21 | 6 | 38.3 |
| count | 34 | 5 | 87.0 | 30 | 5 | 61.0 |
| bw | 28 | 1 | 39.3 | 28 | 1 | 68.5 |
| f51m | 42 | 7 | 99.9 | 48 | 9 | 84.7 |
| frg1 | 60 | 6 | 87.1 | 52 | 6 | 47.7 |
| comp | 68 | 7 | 105.0 | 69 | 7 | 60.8 |
| myadder | 24 | 8 | 113.1 | 31 | 8 | 45.0 |
| 9sym | 96 | 5 | 98.3 | 75 | 5 | 52.0 |
| term1 | 117 | 5 | 96.1 | 79 | 5 | 86.5 |

**Table 2.1**: Comparison of *flowmap* and *bias_clus*. All the delays are in nanoseconds

delay in the critical path of the circuit. Although our algorithm produced circuits with larger depth than *flowmap* (*flowmap* produces minimum depth circuits), our results on delays in the critical path are consistently better (with the exception the circuit *bw*) to those for *flowmap*.

Thus, the critical path delay after placement and routing is about 30% better, on the average, than the results obtained when *flowmap* was used as the technology mapping algorithm. This supports our claim that minimizing the number of levels in the technology mapping process is not enough to guarantee a low delay in the critical path after placement and routing . Instead, it is often useful to use a more accurate estimate of the actual delays and perform timing driven technology mapping in the general delay model.

Table 2.2 shows the comparison of our results with those in [35] for the circuits which we were able to place and route using *apr*. The circuits that we were not able to place using *apr* had more CLBs than the limit imposed by the Xilinx 3000 series FPGAs. The results produced by our algorithm are better than those for *xln_p* on 6 of the benchmark circuits. However, the delay is much larger for the circuits *5xp1* and *misex2*. This is probably due to the fact that our algorithm used a significantly larger number of CLBs for these two circuits, resulting in a much harder placement and routing problem.

We tested our iterative algorithm on a small benchmark – *count* with 129 nodes (in the unclustered feasible circuit). In three iterations, the delays in the critical paths show significant decreases. Table 2.3 shows the delays of the five most critical paths after each of the three iterations. Tables 2.4 and 2.5 show how the delays in each critical path changes in successive

| Circuit | xln_p | bias_clus |
|---|---|---|
| z4ml | 31.0 | 25.2 |
| misex1 | 36.2 | 34.0 |
| vg2 | 76.3 | 65.3 |
| 5xp1 | 35.9 | 76.4 |
| misex2 | 53.7 | 82.0 |
| count | 79.02 | 49.3 |
| 9symml | 54.0 | 61.0 |
| 9sym | 53.7 | 52.0 |
| f51m | 76.6 | 74.7 |

**Table 2.2**: Comparison of *xln_p* and *bias_clus*. All delays are in nanoseconds.

| Iteration number | | |
|---|---|---|
| 1 | 2 | 3 |
| 61.0 | 56.3 | 54.0 |
| 58.3 | 54.0 | 53.5 |
| 55.8 | 54.0 | 53.2 |
| 55.2 | 53.2 | 52.1 |
| 54.0 | 52.0 | 50.0 |

**Table 2.3**: Delays (in ns) of the 5 most critical nets after each of the 3 iterations for circuit *count*.

iterations. The first column of Table 2.4 shows the delays of the five most critical paths after the first iteration. The second column shows the delays of these paths after the second iteration. Similarly, the first column of Table 2.5 shows the delays of the five most critical paths after the second iteration and the second column shows the delays of these paths after the third iteration. Note that the most critical paths after one iteration do not remain the most critical paths after the next iteration because our algorithm is biased to cluster along the most critical paths.

We ran *bias_clus* on a large number of other MCNC benchmarks, but the number of CLBs generated for these larger benchmarks could not be fitted onto one Xilinx 3000 series FPGA chip. So we could not analyze the performance of our performance driven clustering algorithm on these benchmarks.

| Iteration 1 | Iteration 2 |
|---|---|
| 61.0 | 50.3 |
| 58.3 | 55.2 |
| 55.8 | 56.3 |
| 55.2 | 56.2 |
| 54.0 | 55.0 |

**Table 2.4**: Change in the delays (in ns) of the 5 most critical paths between iterations 1 and 2.

| Iteration 2 | Iteration 3 |
|---|---|
| 56.3 | 49.0 |
| 54.0 | 53.2 |
| 54.0 | 51.0 |
| 53.2 | 54.0 |
| 52.0 | 50.0 |

**Table 2.5**: Change in the delays (in ns) of the 5 most critical paths between iterations 2 and 3.

## 2.6 Conclusions

In this chapter we have presented a timing driven technology mapping algorithm for table-look-up based FPGAs using the general delay model. Our biased depth first clustering automatically exploits reconvergent paths and does predecessor packing making it quite competitive with respect to the number of CLBs produced in the technology mapped circuit. As is evident from the results, reducing the depth of the technology mapped circuit alone is no guarantee for a low delay realization of the original circuit. It is crucial to take some estimate of the actual delay incurred when a particular edge is exposed into consideration in the technology mapping process.

# Chapter 3

# Timing Driven Relocation

In this chapter we define the problem of timing driven relocation and propose a solution to it using the concept of slack neighborhood graph. The solution to this problem is used in our algorithms for timing driven placement, reconfiguration and re-engineering. Timing driven relocation provides us with a powerful new tool to reconfigure a physically infeasible placement into a feasible one, without substantial increase in the critical path delay of the circuit. We use this as a unifying framework for developing algorithms for several placement-related problems in regular architectures.

## 3.1 Problem Formulation

The input to the timing driven relocation problem is a combinational circuit, its implementation on an FPGA specified by the placement of the modules and the routing of the interconnections, and a set of slots identified to be *infeasible* denoted $F$. Any circuit modules occupying infeasible slots need to be moved so that all the modules occupy only the feasible slots. Infeasible slots arise due to different reasons in different applications of the timing driven relocation problem. So, in the timing driven placement problem the infeasible slots are the ones where two or more modules overlap, while in the reconfiguration problem, the faulty slots constitute the infeasible set.

The circuit is specified by its interconnection topology, module delays and the signal arrival times and required times at the primary inputs and primary outputs respectively. The topology of the circuit can be abstracted as a *circuit graph*, $G_C(V_C, E_C)$, where each module of the circuit is represented by a vertex in $V_C$, and an interconnection between two modules is represented by an edge in $E_C$ between the corresponding vertices. In the subsequent discussion, we will often use $M_i$ to denote the *vertex* in $V_C$ representing the $i$th module. As part of the problem specification, the arrival times of the signals at the primary inputs, $A(x_i)$, the required arrival times, $R(f_i)$, at the primary outputs and the delays $d(M_i)$ associated with the modules, are also specified. In the case of lookup-table based FPGAs the module delays will all be identical.

The FPGA can be represented by the set of slots, $S$. The set of slots used in a particular implementation of the combinational circuit is the set $U \subseteq S$ and $\bar{U} = S - U$ is the set of *spare slots*. We assume that the IO blocks are restricted to the periphery of the grid of slots, while the other modules are placed in the *core* of the grid.

It is assumed that each module $M_i$ has a fixed delay $d(M_i)$ associated with it, independent of the actual placement of the modules. Further, propagation delay of modules is assumed to be "block-oriented", that is, it is assumed that the latest arriving signal determines the output. These assumptions are only for simplicity of delay computations and are not inherent in the proposed algorithms.

Timing driven relocation moves the circuit modules occupying infeasible slots to feasible slots through series of moves ending at spare slots. Further, the relocation is done such that the critical path delay of the circuit does not increase significantly. More formally, timing driven relocation generates a *relocation function* $\rho : U \rightarrow S - F$, such that

1. $\rho$ is a one-to-one mapping.

2. If $\rho(S_i) = S_j$ and $S_j \in U$, then $\rho(S_j) \neq S_j$.

Notice that this definition of the relocation function defines *relocation paths* from the infeasible slots to the spare slots. Further, none of the relocation paths intersect because $\rho$ is a one-to-one mapping. This means that the circuit modules occupying the slots on the relocation paths can be moved from an infeasible slot to a spare slot along the relocation paths without any conflicts. An obvious necessary condition for the existence of $\rho$ is that there should be at least as many spare slots as there are faulty ones. The main objectives of the relocation algorithm are:

- Minimize the increase in the critical path delay as a result of relocation of modules.

- Minimize the amount of reprogramming to be done to achieve the relocation.

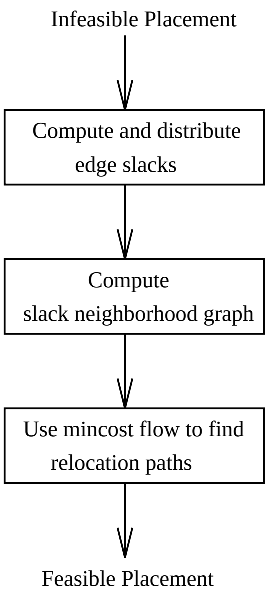


**Figure 3.1**: The main steps in our algorithm for timing driven relocation.

Figure 3.1 shows the main steps in our algorithm for timing driven relocation using slack neighborhood graphs. Fig. 3.2(a) shows a placed circuit with an infeasible set of slots. The vertex disjoint relocation paths shown in Fig. 3.2(c) result in the new placement shown in Fig. 3.2(d).

## 3.2 Slack Neighborhood Graph

In this section we introduce the concept of a slack neighborhood graph and give an efficient algorithm for its construction. We assume that we are given a parameter $\lambda$ which is a measure of the amount of timing degradation (increase in delay along the critical path) that is acceptable. $\lambda$ could be a part of the design specification of the system for timing driven reconfiguration and timing driven design re-engineering. In the case of timing driven placement, the relaxation parameter is decreased gradually as the iterations proceed [30].

Using the placed and routed design, we compute the slack of each edge in the circuit graph which is a measure of the amount by which the delay on the edge can be increased without violating any of the timing constraints. Since a delay increase can be translated into an increase

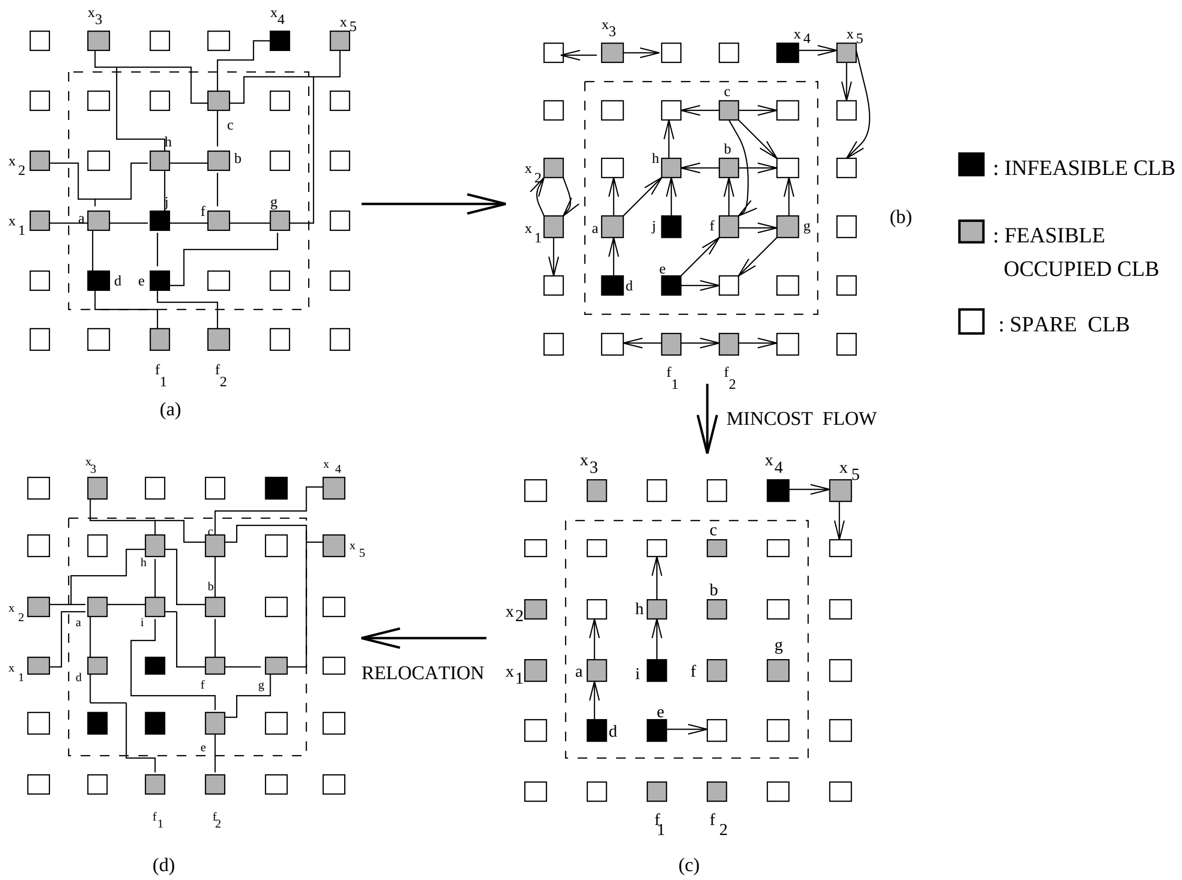


**Figure 3.2**: An example illustrating our timing driven relocation algorithm. (a) A placed and routed circuit and a set of infeasible slots. (b) The slack neighborhood graph (SNG) for the circuit in (a). (c) The vertex disjoint paths in the SNG computed using mincost flow. (d) The new placement after relocation of modules.

in the length of the interconnection corresponding to the edge, the slack can be interpreted as an upper bound on the amount by which the length of the interconnection can be increased, without violating the timing constraints.

Informally, the set of neighboring slots to which a module can be moved without violating any timing constraints is said to be its *slack neighborhood.* The graph in which the adjacency relation reflects these slack neighborhoods is referred to as the *slack neighborhood graph* (see Fig. 3.2 (b)). We now describe in detail the steps involved in the construction of the slack neighborhood graph.

### 3.2.1 Computing Slacks

Using the delay model described earlier, we associate a delay $d(e)$ with each edge $e$ in the circuit graph $G_C$. These edge delays can be computed readily since the circuit has been laid out. It is assumed that the signals arriving early wait for the later arriving signals. So, if the inputs to a module $M_i$ are denoted by $IN(M_i)$ then the arrival time at the output of $M_i$ is given by

$$A(M_i) = \max_{M_j \in IN(M_i)} \{A(M_j) + d(M_j, M_i)\} + d(M_i),$$

where $d(M_i, M_j)$ denotes the signal delay of edge $(M_i, M_j)$. Similarly, the required times at the outputs of the modules is given by

$$R(M_i) = \min_{M_i \in IN(M_j)} \{R(M_j) - d(M_j) - d(M_i, M_j)\}.$$

Edge delays are estimated using the semi-perimeter of the bounding box of the terminals connected by the edge. The arrival times and required times can be computed by the method of dynamic programming using a forward and backward topological sweep of the DAG $G_C$, in time linear in the size of the circuit graph.

The *slack*, $s(e)$, at an edge $e = (M_i, M_j)$ is defined as the difference between the required and arrival times at the *input* to $M_j$ on edge $(M_i, M_j)$. Thus,

$$s(e) = (R(M_j) - d(M_j)) - (A(M_i) + d(M_i, M_j)).$$

If modules are relocated so that the delay of an edge $e$ increases by at most $s(e)$, then no timing constraints will be violated, *provided the delays of none of the other edges increase.* In order to allow all the edges to incur increase in delay *simultaneously*, we need to distribute the edge slacks among all the edges of the circuit graph. This can be accomplished using a variant of the zero slack algorithm [36]. The distributed edge slack at edge $e$ is denoted by $s_d(e)$. The relaxation parameter, $\lambda$, is used to increase the slacks at all the edges by an amount equal to $\lambda$. Thus, the *relaxed slack* at edge $e$ is given by

$$s_r(e) = s_d(e) + \lambda.$$

### 3.2.2 Constructing the Slack Neighborhood Graph

Given the current placement, $\phi : V_C \rightarrow S$, and the relaxed slacks on all the edges of the circuit graph, we can define the *slack neighborhood* for each slot $S_i$, $S_i = \phi(M_i)$, as

$$\nu(S_i) = \{S_j \in S - S_i - F | d'(e) - d(e) \leq s_r(e) \text{ for all edges } e \text{ incident to module } M_i\},$$

where $d'(e)$ is the delay in edge $e$ when module $M_i$ is moved from $S_i$ to $S_j$, with all other modules remaining unchanged. So, the slack neighborhood of a slot contains all the slots to which $M_i$ can be moved without increasing the delay in any of the edges incident to $M_i$ by an amount more than that dictated by the relaxed slacks of the edges.

These slack neighborhoods are used to define a slack neighborhood graph $G_N(V_N, E_N)$, where there is a vertex in $V_N$ for each slot in $S$, and the there is an edge $(S_i, S_j)$ if and only if $S_j \in \nu(S_i)$ (Again, we are using $S_i$ to represent the vertex in $G_N$ corresponding to $S_i$). Fig. 3.2 (b) shows the slack neighborhood graph corresponding to the infeasible set and placement in Fig. 3.2 (a). The slack neighborhood of the module in position (2,4) consists of the slots (2,3), (2,5) and (4,4). It should be noted that the slack neighborhoods of IO modules are confined to the periphery of the slot array while those of the logic modules are confined to the core of the slot array.

### 3.2.3 The Relocation Cost Function

When a logic block in the combinational circuit is mapped from a slot $S_i$ to slot $S_j$, the following factors contribute to the reprogramming cost:

- $S_j$ needs to be reprogrammed to implement the functionality which was earlier implemented by $S_i$.
- All the nets having an endpoint in $S_i$ need to be rerouted to $S_j$. This entails reprogramming of the routing blocks through which the nets involving $S_i$ were being routed.

Thus, moving a module which has a lot of nets coming in and going out is much costlier than moving one which is not so "strongly anchored". We associate a *relocation cost*, $c(S_i, S_j)$, with each edge $(S_i, S_j)$ in the slack neighborhood graph. The cost is computed using congestion gradient [30], the number of nets connected to $S_i$ and the status of the slot $S_j$ (empty or occupied).

### 3.2.4 Relocation Using Mincost Flow

In this section we propose a solution to the relocation problem for FPGAs using *mincost flow* in a network derived from the slack neighborhood graph. The network flow approach has been used for reconfiguration of processor arrays by Codenotti and Tamassia [45]. However, the objective in their case was minimization of the total interconnection length. A significant feature of our algorithm is that it simultaneously achieves both low timing degradation and low reprogramming cost.

We transform the SNG into a flow network by adding a dummy source vertex, $s$, connected through zero cost, infinite capacity edges to all the vertices corresponding to the infeasible, occupied slots (represented by the set $V_s$). All the vertices corresponding to the spare slots (represented by the set $S_t$) are connected to a dummy sink, $t$, by similar edges. Note that $V_s = F$ and $V_t = S - F - U$.

Now that we have a weighted flow network, $G'_N$, we can use a mincost flow algorithm to find vertex disjoint paths in $G'_N$ from the sources to the sinks. We set the capacity of all the vertices in $G'_N$ other than $s$ and $t$ to 1. The following theorems establish the correctness of the flow based algorithm for computing the relocation paths that maps the infeasible placement to a new feasible placement. We omit the proofs due to lack of space.

**Theorem 3.1** (Integrality Theorem) *Given a flow network with integral capacities, there exists a mincost maximum flow in which the flow through all the edges is an integer.*

*Proof.* See [46]. □

If we restrict our attention to integral flows, the minimum flow through any path having non-zero flow is one unit. Further, all nodes have capacity 1, so no two paths with non-zero flow can share a node (except $s$ and $t$). These observations yield the following theorem:

**Theorem 3.2** (**Disjoint Paths Theorem**) *In any integral feasible flow $f$ of value $W$ in $G'_N$, the edges in $E_N$ with non-zero flow can be decomposed into $W$ vertex disjoint paths. Further, each path starts at a vertex $S_i \in V_s$ and terminates at a vertex $S_j \in V_t$.*

*Proof.* Since the flow is integral and the vertex capacities are all 1, an edge is either saturated (has flow of 1 unit) or has zero flow. Also, any vertex $x$ either has no incoming and no outgoing flow, or it has exactly one saturated edge coming in and exactly one saturated edge going out. So, the saturated edges in $f$ form simple, vertex disjoint paths in $G'_N$ starting at $s$ and ending at $t$. If we restrict these paths to $E'_N$ they start at a vertex in $V_s$ (since the only edges out of $s$ are to vertices in $V_s$) and end at a vertex in $V_t$ (since the only vertices having edges to $t$ are the ones in $V_t$). Since each path carries one unit of flow, the number of such paths in a flow of value $W$ is exactly $W$. □

Thus, the disjoint paths in the SNG computed by the mincost maximum flow start at an overcrowded slot and end at an empty slot with all intermediate slots being occupied by one module. Hence, we have the following theorem:

**Theorem 3.3** *Let $f$ be an integral maximum flow of value $|V_s|$ in $G'_N$, then the relocation function $\rho : V_N \rightarrow V_N$ defined as*

$$\rho(S_i) = \begin{cases} S_j & \textit{if } f(x,y) = 1 \\ S_i & \textit{otherwise} \end{cases}$$

*is a valid relocation function.*

*Proof.* First of all we note that since our flow network has integer capacities, there is an integral maximum flow in $G'_N$, if there is any feasible flow. By Theorem 3.2, we know that the saturated edges in $f$ form $|V_s|$ vertex disjoint paths from $V_s$ to $V_t$. So, there is a path from each source to a sink. Since these paths are vertex disjoint, no two vertices are mapped to the same vertex in $\rho$. So, $\rho$ is a valid reconfiguration function. □

Notice that a solution to the relocation problem exists only if $G'_N$ has a maximum flow of value $|F|$. If not, then the value of $\lambda$ needs to be increased to obtain a neighborhood graph with sufficiently high connectivity. An example of the vertex disjoint paths defining the reconfiguration mapping for the slack neighborhood graph in Fig. 3.2 (b) is shown in Fig. 3.2 (c). Notice that each path with non-zero flow starts at an infeasible occupied slot and ends at a spare slot. Moving the modules along these reconfiguration paths results in the configuration shown in Fig. 3.2 (d) in which all the modules are mapped to feasible slots. We used the CS2 package for mincost flow [17] in our implementation.

### 3.2.5 Analysis of Timing Degradation

In this section we present some analytic bounds on the increase in critical path delay during reconfiguration. The following theorem gives a naïve bound on the worst case increase in the delay of the critical path in the circuit.

**Theorem 3.4** *In the timing driven reconfiguration, the worst case increase in the delay along any path $\pi$ in the circuit is $2l(\pi)\lambda$, where $l(\pi)$ is the number of edges in the path and $\lambda$ is the relaxation parameter.*

*Proof.* The slack distribution phase insures that if the delays in the various edges are increased *independently* by amounts equal to the slacks allocated to them by the slack distribution, then there is no increase in the arrival times of the signals at the various outputs. Since we are increasing the slack allocated to all the edges by a constant, $\lambda$, in computing the slack neighborhoods, an isolated move of a node to a slot in its slack neighborhood can increase the delay along a path by at most $\lambda$. In the worst case, when two modules adjacent on a path are relocated simultaneously in a reconfiguration, the delay on the interconnection between them can increase by $2\lambda$ if they move in "opposite" directions. Further, in the worst case scenario these delay increases add up along all the edges in a path resulting in an increase of $2\lambda l_\pi$ in the delay along a path $\pi$. □

**Corollary 3.1** *No delay increase is incurred in the timing driven reconfiguration if $\lambda = 0$. Thus, a reconfiguration with no additional slack does not cause any increase in the critical path delays of the circuit.*

The bound on the increase in delay during the timing driven reconfiguration given by Theorem 3.4 is a worst case bound and the performance degradation is expected to be much lower on the average. We need some definitions before analyzing the performance of our timing driven reconfiguration algorithm in greater detail. Paths in the placed circuit graph are referred to as *circuit paths.* The *shadow of a circuit path* consists of the slots corresponding to the modules in the circuit path and the slots adjacent to the interconnections used by the path (see Fig. 3.4).

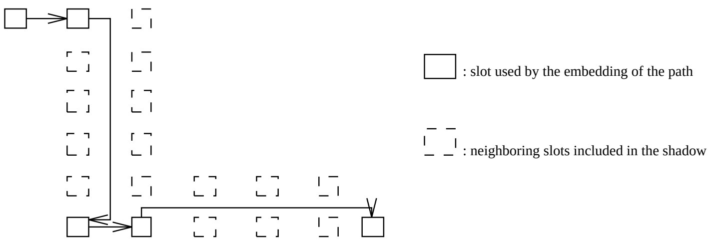


**Figure 3.3**: The shadow of a circuit path. Both the solid and dashed slots are included in the shadow.

Intuitively, the shadow of a circuit path identifies the set of slots in the vicinity of the embedding of the path. Thus, given a circuit path $\pi$, its shadow, $sh(\pi)$, is defined as

$$sh(\pi) = \phi(\pi) \cup \{S_i \in S | S_i \text{ is adjacent to } \phi(\pi)\},$$

where $\phi(\pi)$ refers to the slots occupied by modules in $\pi$ in the current placement (see Fig. 3.3). In the context of the definition of the shadow of a path, we define a slot to be adjacent to the embedding of a circuit path if it can directly access the channel used to route an interconnection in the path.

Paths in the SNG that carry non-zero flow in the min-cost flow computed by the reconfiguration algorithm, are termed *flow paths.* Flow paths are vertex disjoint and define the reconfiguration mapping. A flow path, $\pi_f = (S_1, S_2, \ldots, S_k)$, is said to *conform* to a circuit path, $\pi_C = (M_1, M_2, \ldots, M_l)$, if it satisfies the following conditions:

1. $\pi_f - \{S_k\} \subseteq sh(\pi_C)$,

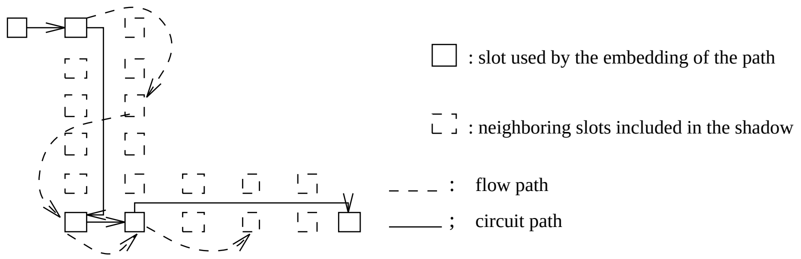


**Figure 3.4**: A flow path that conforms to the circuit path shown. All the slots shown in the figure form the shadow of the circuit path.

2. The vertices in $\pi_f$ occur in the same order as the order in which they are encountered as one moves along $\pi_C$ in the direction of signal flow.

3. If $S_i = \nu(M_j)$ then $S_{i+1}$ must be at or before the slot for $M_{j+1}$, in the ordering defined by the signal flow along the shadow of $\pi_C$.

Fig. 3.4 shows an example of a conforming path.

Thus, conforming paths correspond to a reconfiguration pattern in which the images of the slots under reconfiguration cascade along the shadow of a circuit path.

**Theorem 3.5** (**Conforming Path Theorem**) *In a reconfiguration along a flow path $\pi_f$, which conforms to circuit path $\pi_C$, the delay along $\pi_C$ is increased by at most $2\lambda$.*

*Proof.* Let the flow path be $\pi_f = (x_1, x_2, \ldots, x_k)$. By definition, if $\pi_f$ conforms to the circuit path $\pi_C = (y_1, y_2, \ldots, y_l)$, then $x_i \in sh(\pi_C), 1 \leq i < k$. We will consider the contribution of the relocation of each $y_i \in \pi_f \cap \nu(\pi_C)$ to the increase in the delay of the path $\pi_C$. The flow path $\pi_f$ defines the reconfiguration $x_1 \rightarrow x_2 \rightarrow \ldots x_{k-1} \rightarrow x_k$. Further, since $\pi_f$ conforms to $\pi_C$, the order of the $x_i$s is consistent with the direction of signal flow along the circuit path $\pi_C$ and after reconfiguration the slot occupied by any $y_i \in \pi_C \cap \pi_f$ is between the slots for $y_i$ and $y_{i+1}$ in the signal flow ordering along the shadow of $\pi_C$ before reconfiguration. Consider any module $y_i, 2 \leq i \leq k-1$, if $y_i \in \pi_f \cap \nu(\pi_C)$ then it is not moved by the reconfiguration defined by $\pi_f$ and does not contribute to any increase in the delay along $\pi_C$. Otherwise, the increase in $d(y_{i-1}, y_i)$ is equal to the decrease in $d(y_i, y_{i+1})$, resulting in no increase in delay due to the relocation of $y_i$. This is not true if $y_1$ or $y_k$ are relocated, but this can result in a delay increase of at most $2\lambda$ along the path. $\square$

The conforming path theorem illustrates that the naïve bound on the worst case increase in the delay along a path given by Theorem 3.4 is too pessimistic. In fact, the conforming path theorem suggests that costs should be assigned in such a way that the min-cost flow is biased towards using conforming paths. From the above discussion we conclude that the min-cost flow based reconfiguration can be accomplished in a manner that guarantees low performance degradation.

## 3.3 Applications of Timing Driven Relocation

Timing driven relocation is a powerful tool that can be used to develop algorithms for a lot of different problems. It allows our algorithms to move from an infeasible placement (the infeasibility arises due to different reasons in the different algorithms) to a feasible one with provably low timing degradation. Thus, it provides a powerful transformation on a placement in a regular architecture that proves useful in a variety of situations. In particular, it allows our algorithms to step through physically infeasible placements in their quest for a good placement. In the next three chapters we will study the problems of timing driven placement, reconfiguration and placement re-engineering. Timing driven relocation is one of the key steps in the algorithms we propose for all these problems.

# Chapter 4

# Timing Driven Placement

Placement is a fundamental problem in the design of VLSI circuits, and it involves finding "good" positions for the circuit modules on the silicon substrate. Since placement is followed by routing of the interconnections between the modules, traditional placement algorithms have used routability and total area as the main metrics for guiding placement. Indeed, placement has a big impact on the routability of interconnections and the final die size of the manufactured chip. In the early days of VLSI, the circuit modules had large delays, and consequently, the total delay of any circuit path was dominated by the delays of the modules. Since module delays do not depend on module placement, it was not necessary to take timing issues into consideration during placement. However, as feature sizes have shrunk, the module delays have decreased substantially, resulting in circuit delay being dominated by interconnect delays. Since the length and topology of the interconnects is determined by the placement of the modules, it has become vital to handle timing issues during placement.

The problem of timing driven placement has been studied extensively [4, 9, 15, 16, 20, 21, 28, 44, 47, 48]. The traditional approaches to this problem can be broadly classified into *path-based methods* [10, 21, 44], and *net-based methods* [9, 15, 16, 28]. Path-based methods consider timing requirements explicitly and try to satisfy timing requirements and physical requirements simultaneously during the placement phase. A major difficulty encountered in path-based methods is the enormous complexity of computation. In a net-based algorithm, timing constraints are first translated into physical constraints, such as upper and lower bounds on the lengths of nets. The placement algorithm then tries to generate a placement that satisfies these physical constraints on the nets, in the hope that this solution will also satisfy the original timing constraints. A drawback of the net-based methods is that they might over-constrain the placement process, since they choose a particular distribution of delays *before* starting the placement.

In this chapter, we present a new algorithm for timing driven placement that is applicable to architectures with a regular structure. Thus, the placement problems for Field Programmable Gate Arrays (FPGAs), restricted cell based designs and some of the Wafer Scale Integration (WSI) architectures fall within the scope of this work. Our algorithm uses an iterative approach. In each iteration there is a *compression phase* and a *relaxation phase.* The compression phase attempts to make the placement *delay feasible* by compressing the long paths that cause some of the primary output signals to arrive too late. However, the compression phase may produce an infeasible placement with some of the slots occupied by two (but no more than two) modules. This allows the compression phase more flexibility and often allows it to achieve the required decrease in delay. If an infeasible placement is produced in the compression phase, the relaxation phase, which carries out a *timing driven reconfiguration* of the infeasible placement to produce a feasible placement, will be executed. We use the concept of a *slack neighborhood graph* to achieve a provably good reconfiguration, in the sense that the delays in the critical paths are guaranteed not to increase beyond a certain bound. The neighborhood graph is computed using the slacks in the current placement. It captures the freedom of movement of the modules, without violating the timing constraints.

The good convergence behavior for our iterative algorithm indicated by our analytic results is validated by experimental results on benchmark circuits. Our approach can be viewed as a combination of both path-based and net-based approaches, since the compression phase is path-based while the relaxation phase is net-based. Also, our algorithm simultaneously places the input-output pins and the logic modules. Most iterative placement algorithms proposed in the literature [18] used very local transformations, such as pairwise swap, to move from one configuration to another. Our algorithm makes more global transformations, using flows in weighted slack neighborhood graphs to ensure that the new placement is good. Further, since the flow-based reconfiguration step of our algorithm transforms an infeasible placement (with overlapping modules) into a feasible one, this step can also be used as the back-end for some other performance-driven placement algorithms [44].

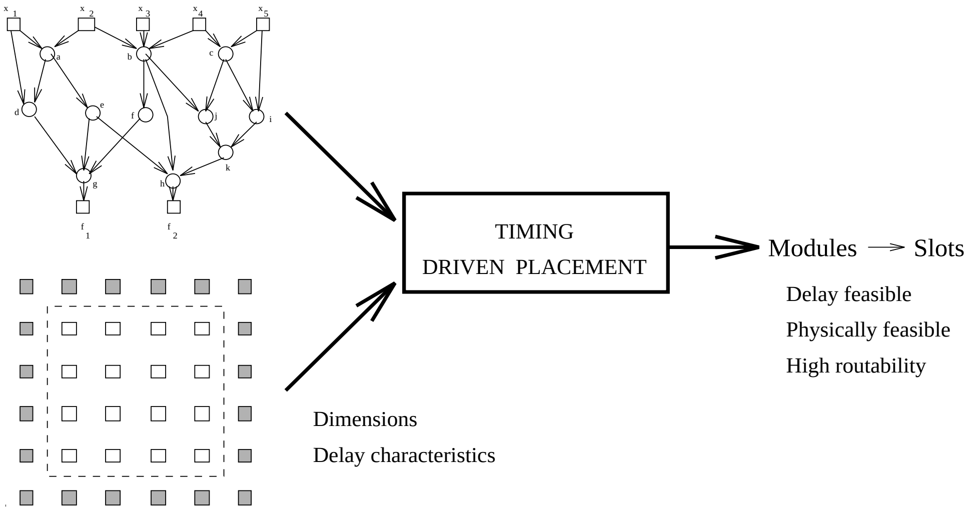


**Figure 4.1**: The timing driven placement problem for regular architectures.

## 4.1 Applicability and Assumptions

We assume that the underlying architecture on which the circuit modules have to be placed is regular. As discussed in Chapter 1, this implies that the architecture has the properties of discreteness and regularity. These properties of regular architectures allow us to use powerful graph theoretic techniques. FPGAs, identical standard cell based architectures and WSI arrays of blocks with a regular structure are examples of architectures that satisfy the above requirements. Under these restrictions, the underlying architecture can be viewed as a set of discrete, identical slots: $S = \{S_1, S_2, \ldots, S_m\}$, along with routing resources (routing channels, pre-placed interconnections, programmable switches, etc.). Because of the regularity in structure, it is easy to estimate interconnection delay without performing detailed routing. In the case of FPGAs, since all the routing resources are already present, the prediction of routing delay is made even easier. We assume that the input and output pins are restricted to the periphery of the grid of slots, while the other modules are placed in the *core* of the grid (see Fig. 1.2).

We assume that a fixed internal delay is associated with each module $M_i$ which is denoted $d(M_i)$. The difference in delays for the rising and falling transitions is ignored. We also make the assumption that all logic combinations can exist at the inputs of the modules. This introduces the possibility of tracing false paths. However, since we are dealing with fairly complex modules the assumption is a reasonable one. Further, propagation delay of modules is assumed to be "block-oriented", that is, it is assumed that the latest arriving signal determines the output. These assumptions are introduced only to simplify the delay computations and are not inherent in the proposed algorithms. The interconnection delay computations in our algorithm use the *general delay model* that allows different wire segments belonging to the same output net to have different delays. This is in contrast with the traditional net based delay models that associate a constant delay with a net, irrespective of its topology. Using the general delay model is specially important when modeling interconnection delays in architectures that have routing resources with widely differing conductivities and capacitances, or when there are programmable switches in the interconnecting paths (as in FPGAs).

## 4.2 Problem Formulation

The input to the timing driven placement problem is a circuit, $C$, consisting of a set of modules $\{M_1, M_2, \ldots, M_n\}$, interconnected to implement the required functionality. The topology of the circuit can be abstracted as a *circuit graph*, $G_C(V_C, E_C)$, where each module of $C$ is represented by a vertex in $V_C$, and an interconnection between two modules is represented by an edge in $E_C$ between the corresponding vertices. In the subsequent discussion, we will often use $M_i$ to denote the *vertex* in $V_C$ representing the $i$th module. Let

$$PI = \{x_1, x_2, \ldots, x_k\}$$

be the set of primary inputs to the circuit and

$$PO = \{f_1, f_2, \ldots, f_l\}$$

be the set of primary outputs. As part of the problem specification, the arrival times of the signals at the primary inputs, $A(x_i)$, the required arrival times, $R(f_i)$, at the primary outputs and the delays $d(M_i)$ associated with the modules, are also specified.

A placement is an injective mapping, $\phi$, from $V_C$ to the set of slots in the placement area. A placement is said to be *delay feasible* if it satisfies all the timing constraints. The aim of the timing driven placement problem is to find a delay feasible placement that is easily routable. We call a placement *(physically) feasible* if no slot is occupied by two or more modules, otherwise it is said to be *infeasible*.

## 4.3 Overview of the Algorithm

Our algorithm starts with an initial placement generated by placing the modules uniformly in the core of the grid, and the primary inputs and outputs on the periphery. A delay analysis is performed to identify the primary outputs where the required-time constraints are violated. Our algorithm selects, among these outputs, the one where the difference between the arrival time and required time is maximum and attempts to decrease the arrival time at this output in a two phase process consisting of a *compression phase* and a *relaxation phase* (see Fig. 4.2). In the compression phase, the *longest path tree* of the cone of this output is extracted, and paths in this tree are shortened by decreasing the interconnection delays on the edges in the tree. This is accomplished by moving modules and inputs towards the slot occupied by the primary output. This process is guided by the signal flow direction and is path-based. In the compression phase, up to two modules can occupy the same slot (making these slots "overcrowded"). This allows the compression phase to employ a more aggressive compression strategy that would not have been possible if the new placement was constrained to be a feasible one.

If the compression phase produces an infeasible placement, the original modules occupying the overcrowded slots need to be relocated. In the relaxation phase, relocation is carried out simultaneously for all the modules in such a way that the delays do not increase substantially. This is accomplished by using the algorithm for timing driven relocation described in Chapter 3. This requires the computation of the slacks for all the edges in the circuit graph. The slack of an edge measures the amount by which the delay of the edge can be increased without violating any timing constraints. The slacks of the edges incident to a module determine the neighborhood within which the module can be moved without violating the timing requirements. In any iterative algorithm for placement, it is essential that the mobility of the modules should be sufficiently high initially. This ensures that a bad initial placement does not cause the algorithm to get stuck in a high cost local minimum. In order to prevent the mobility from being completely governed by the slacks, we introduce a *relaxation parameter* ($\lambda$) that allows the algorithm to increase the values of edge slacks which will be referred to as *relaxed slacks*.

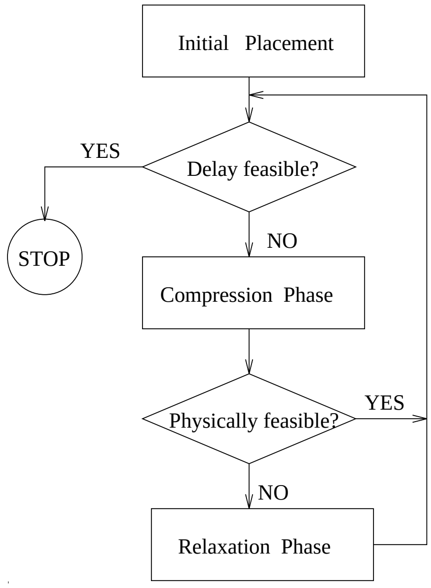


**Figure 4.2**: Overview of our timing driven placement algorithm.

The relaxed slacks are then used to compute a *slack neighborhood graph* (SNG) in which an edge from node $u$ to node $v$ indicates that the module corresponding to node $u$ can be moved to the slot currently occupied by the module corresponding to node $v$ *without violating any timing constraints*. In order to incorporate a routability measure into the placement process, each edge of the SNG is associated with a cost that measures the penalty, in terms of increase in congestion, that results from the move associated with that edge. A reasonable measure of this penalty is a *congestion gradient* that measures the difference in congestion in different areas in the current placement.

Once the weighted slack neighborhood graph has been computed, a min-cost flow formulation is used to find node disjoint paths in the slack neighborhood graph. These paths are then used to relocate the modules that were occupying the overcrowded slots in the infeasible placement. A min-cost flow ensures that congestion in the resulting placement will be low, leading to a more routable placement. Also, the fact that the reconfiguration is being carried out in the slack neighborhood graph guarantees that the paths in the circuit will not become too long. It might happen that the SNG is too sparse to allow enough node disjoint paths for the relaxation phase to reconfigure the infeasible placement to a feasible one. In such a scenario, the relaxation parameter is increased to allow more edges in the SNG, enabling successful reconfiguration.

The relaxation parameter is decreased according to a *shrinking schedule* in successive iterations of the algorithm to gradually "tighten" the relaxation phase. This is reminiscent of the cooling schedules used in simulated annealing [23]. It allows sufficient mobility of the modules during the early iterations and gradually restricts the mobility as the algorithm proceeds. The iterative process terminates when all the timing constraints are satisfied.

Fig. 4.3 shows the difference between the exploration of the solution space by our algorithm and traditional iterative placement algorithms. The solid path shows a possible path taken by a traditional iterative algorithm which moves from one feasible solution to another. The dashed path taken by our algorithm passes through some infeasible solutions and allows it to perform more aggressive compression.

We show one iteration of our algorithm in Fig. 4.5. Fig. 4.5(a) shows the placement at the beginning of the iteration, along with the longest path tree for primary output $f_1$. The compression of this longest path tree results in an infeasible placement shown in Fig. 4.5(b) (overcrowded slots are shown in black). This figure also shows the slack neighborhood graph. Mincost flow is used to find the node disjoint paths shown in Fig. 4.5(c) and these are used to reconfigure the placement to the feasible placement shown in Fig 4.5 (d). Some of the details in Fig. 4.5 will be explained in the subsequent sections.

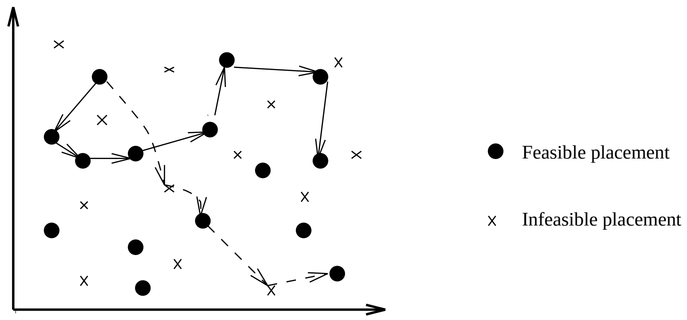


**Figure 4.3**: Exploration of the solution space by our algorithm (dashed path) versus that by traditional iterative algorithms (solid path).

## 4.4 The Compression Phase

The compression phase attempts to move the current placement closer to a delay feasible placement by reducing the arrival times at the outputs. A delay analysis is carried out on the

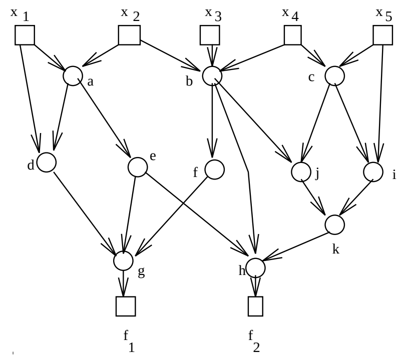


**Figure 4.4**: The boolean circuit used in the illustration of an iteration of our algorithm in Fig. 4.5

current placement to compute the actual arrival times of the signals at the various outputs and the outputs where the timing requirement is violated are identified. Associated with each such output is a set of long paths ending at the output, that are responsible for the violation of the timing requirement. We define the *cone* of a primary output $f_i$ to be the set consisting of $f_i$ and all its predecessors. Thus, all the nodes that can have an effect on the arrival time of the signal at $f_i$ are in $cone(f_i)$. The algorithm attempts to satisfy the timing requirement at $f_i$ by moving modules in the cone to new slots so as to decrease edge delays, a process referred to as *compressing the cone at* $f_i$. It should be noted that in the process of compressing $cone(f_i)$, up to two modules can be placed in the same slot. This ensures that compressing of the cone has enough flexibility to guarantee a substantial decrease in the critical path delay of the cone. If there are several outputs at which the arrival time exceeds the required time, the cone of the output with the maximum violation of the required time is chosen for compression.

The following steps are used to accomplish the compression of a cone, $cone(f_i)$, that violates the required-time constraint (see Fig. 4.6 :

1. **Computing the longest path tree(LPT)**: For each node $x$ in $cone(f_i)$, we find a longest path from $x$ to $f_i$. These paths form a tree which will be referred to as the *longest path tree* (see Fig. 4.7). This tree can be found in linear time using the following dynamic programming formulation: let $l(v)$ be the length of the longest path *starting* at vertex $v$, then

$$l(v) = \max_{(v,x)\in E} \{l(x) + d(v, x)\}.$$

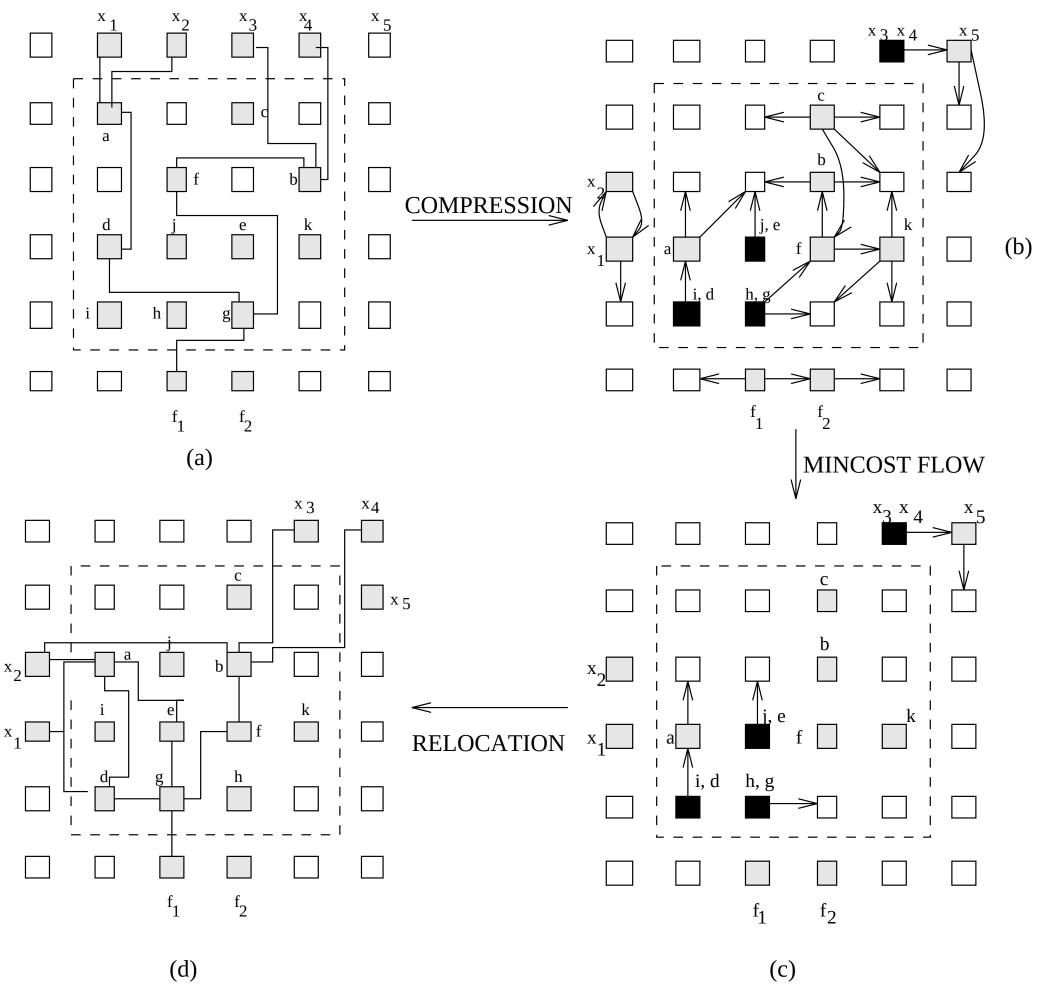


**Figure 4.5**: An iteration of our algorithm on the circuit in Fig. 4.4. (a) The initial placement and the longest path tree of $cone(f_1)$. (b) The infeasible placement obtained after compressing the longest path tree of $cone(f_1)$. The edges of the slack neighborhood graph are shown. (c) The edges with non-zero flow in the mincost flow computed on the transformed SNG. (d) The new placement obtained after relocating modules along the node disjoint paths in (c).

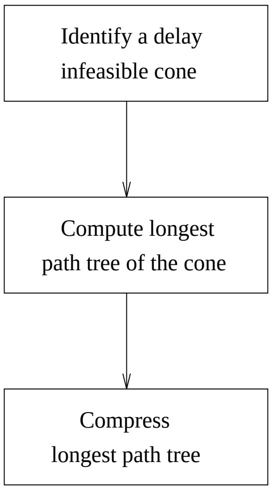


**Figure 4.6**: The steps in the compression phase.

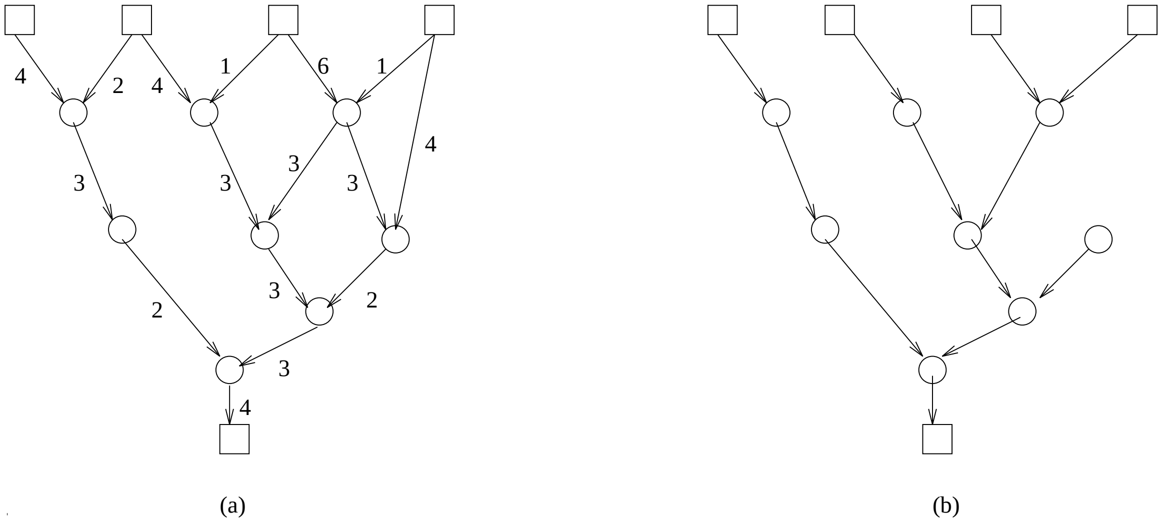


**Figure 4.7**: (a) An edge weighted logic cone and (b) the corresponding longest path tree.

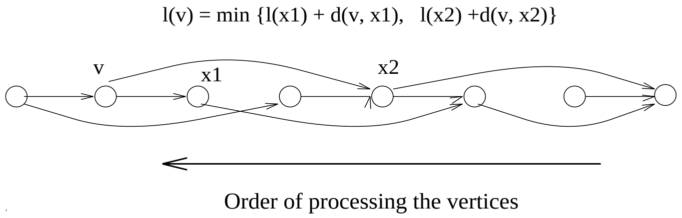


**Figure 4.8**: Illustration of the computation of the longest path tree.

Hence, by processing the vertices in reverse topological order (which is possible because $cone(f_i)$ is a DAG), $l(v)$ (and hence the edge $(v, x)$) can be computed in linear time for all the vertices in the cone (see Fig. 4.8). Note that there may be several different longest paths trees of a cone. Our algorithm chooses an arbitrary longest path tree of the cone being compressed. Intuitively, the longest path tree captures most of the critical paths in the cone, and is at the same time structurally simple. This makes the compression using LPTs very effective and relatively simple.

2. **Compressing the LPT**: The nodes in the LPT are processed in topological order starting from the tip of the cone, and an edge $(M_i, M_j)$ in the longest path tree is compressed by moving module $M_i$ closer to the slot occupied by module $M_j$. The amount an edge is compressed is proportional to the current delay of the edge (long edges have more potential for compression). When module $M_i$ is moved to shorten edge $(M_i, M_j)$, the entire subtree rooted at $M_i$ is displaced by the same amount to ensure that displacing $M_i$ does not increase the arrival time of the signal at $M_i$. This strategy of moving the entire subtree rooted at module $M_i$ is termed *conformal compression* (see Fig. 4.9). The compression algorithm also keeps track of the compression achieved so far on the unique path from $M_j$ to the root of the longest path tree, thus allowing it to adaptively decrease the amount of compression forced on edge $(M_i, M_j)$ if enough compression has already been achieved on the path from $M_j$ to the root of the longest path tree. Fig. 4.10 shows an example of edge compression. Note that the compression algorithm makes certain that an IO pin will not be relocated to a slot in the core of the grid, and a logic module will not be relocated to a slot in the periphery.

Consider the compression phase for the example shown in Fig. 4.5. Notice that the compression of the longest path tree shown in Fig. 4.5 (a) causes the inputs $x_1$ and $x_2$ to move much closer to the output $f_1$. Also, modules pairs ($i$, $d$), ($h$, $g$), ($j$, $e$) and ($x_3$, $x_4$) overlap in the resulting placement.

The compression algorithm attempts to make the task of the relaxation phase easier by relocating modules to empty slots, if they are available at the right distance to achieve the required compression. Further, an attempt is made to spread out the infeasible slots to make the relocation in the relaxation phase easier.

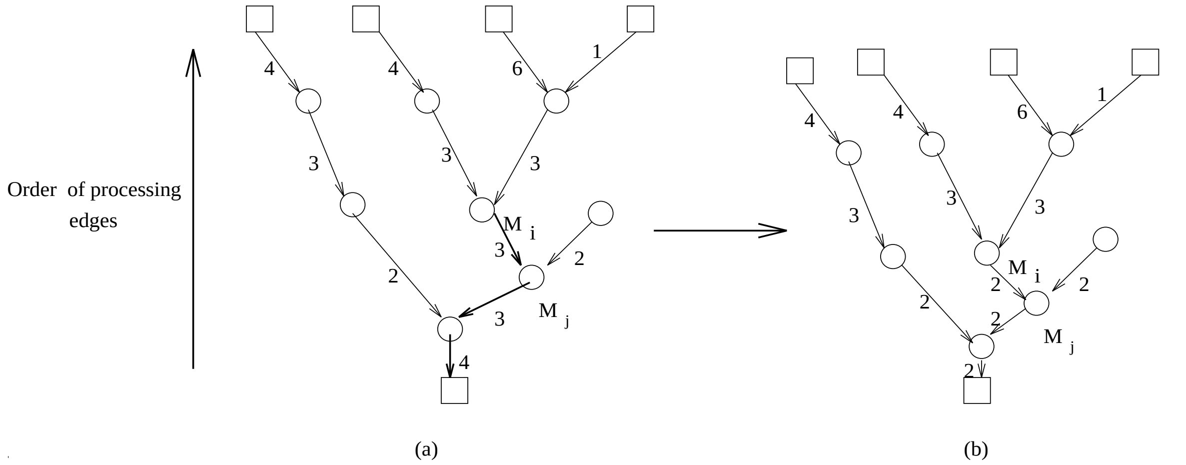


**Figure 4.9**: (a) An edge weighted longest path tree. (b) The longest path tree after compression of the three bold edges in (a).

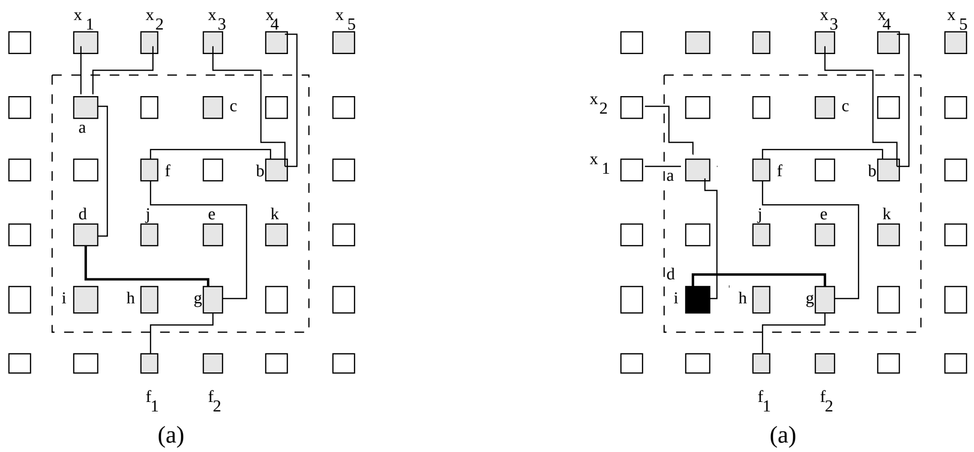


**Figure 4.10**: Conformal compression of the edge from module $d$ to module $g$. (a) Longest path tree and placement before compression. (b) Placement after compression of the edge. Notice that module $a$ also moves down.

## 4.5 The Relaxation Phase

If the compression phase generates a placement with overlapping modules, then the relaxation phase is used to get back to a feasible placement. The input to the relaxation phase is an infeasible placement with some slots containing two modules. During the relaxation phase the placement is reconfigured by moving modules from the overcrowded slots to the vacant slots. By defining the set of overcrowded CLBs as the infeasible set, we can use an algorithm for the timing driven relocation problem to transform the infeasible placement to a feasible one by moving one of the two modules overlapping on the infeasible CLBs to a spare CLB along a relocation path (see Chapter 3 for details of the SNG based timing driven relocation algorithm).

There are a few aspects of the timing driven relocation algorithm that need to be customized for the reconfiguration phase of our placement algorithm. In particular, the relaxation parameter needs to be controlled as the algorithm proceeds to regulate the mobility of the modules during the relaxation phase. Also, the cost associated with edges in the SNG need to be defined in a manner specific to the placement problem.

### 4.5.1 The Relaxation Parameter

As observed earlier, in the early iterations, we allow the modules higher mobility than that dictated by the slacks on their interconnections. This enables the algorithm to explore more freely placement configurations that would otherwise have been unreachable from the initial placement. This expansion in neighborhood is accomplished by using a *relaxation parameter* ($\lambda$) that is added to the slacks of all the interconnections. The edge slacks with the relaxation parameter added are referred to as the *relaxed slacks* ($rs(e)$). Thus,

$$rs(e) = ds(e) + \lambda.$$

The value of the relaxation parameter is decreased in successive iterations according to a *shrinking schedule*. However, if the relaxation phase in an iteration is not able to reconfigure an infeasible placement, the relaxation parameter is increased to expand the slack neighborhoods. In our experiments we halved the relaxation parameter from one iteration to the next.

### 4.5.2 Computing Edge Costs

One of the important objectives of the placement step is to ensure routability. We use a congestion metric as a guide to preferentially direct the reconfiguration process to move modules into areas with lower congestion. The motivation for this being that a placement with low *congestion gradient* (or a placement with more uniform congestion) is more likely to be routable.

The computation of congestion gradients requires the division of the given chip area into subareas or *precincts*. These precincts may be defined by dividing the chip area into rectangular subregions. The congestion in a precinct is the number of occupied slots in the precinct and the congestion gradient between precincts is the difference in their congestion. The congestion gradient measures the variation in congestion in various directions and is used to define a cost for each edge in the slack neighborhood graph. The cost of an edge $e = (S_i, S_j)$ in the SNG is given by

$$c(e) = cong\_grad(S_i, S_j) + is\_occ(S_j),$$

where $cong\_grad(S_i, S_j)$ is the difference in congestion between the precinct containing $S_j$ and the precinct containing $S_i$, and $is\_occ(S_j)$ is 1 if $S_j$ is occupied and 0 if it is empty. Thus, the cost of an edge is higher if it goes into a highly congested area or if it goes to an occupied slot.

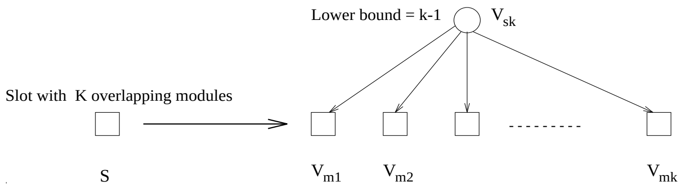


**Figure 4.11**: Handling $k$-way overlap.

### 4.5.3 Handling $k$-way Overlap

It is possible to extend the slack neighborhood graph based approach to timing driven reconfiguration to handle situations where $k$ modules overlap on one slot. For a slot $S$ that has $k$ modules: $M_1, M_2, \ldots, M_k$ assigned to it, we generate $k$ vertices $V_{M_1}, V_{M_2}, \ldots, V_{M_k}$ in the slack neighborhood graph. Vertex $V_{M_i}$ has a slack neighborhood determined using the slack associated with edges incident to module $M_i$. In order to ensure that $k-1$ of the modules get relocated in the relaxation phase, we add a new vertex $V_{sk}$ that has a *lower bound* of $k-1$ on the flow through it. $V_{sk}$ is connected to the vertices $V_{M_i}$ with zero cost edges (see Fig. 4.11). The compression phase needs to be modified to take advantage of $k$-way overlap. In our experiments, we found that $k$-way overlap (for $k = 3, 4, 5$) does not improve the performance of our placement algorithm.

## 4.6 Experimental Results

We executed our algorithm on a Sparc 10. We will refer to the package implementing our algorithm as *sysdias* (an acronym for *sys*tolic-*dias*tolic, reflecting the compression and relaxation phase in each iteration). *Sysdias* was tested on a set of MCNC benchmarks and compared to the *apr* (Automatic Place and Route) tool provided by Xilinx for their 3000 series FPGAs. The MCNC benchmarks were technology mapped and then placed and routed using *apr* to obtain the arrival times at the primary outputs. These arrival times were used as the required times at the outputs for our algorithm. Our placement algorithm was allowed to continue even after it generated a placement that satisfied the required times at all the outputs by decreasing the required times, until it was no longer able to achieve significant delay reduction in an iteration. The placement produced was then routed using *apr* to compute the actual critical path delay. The results of these experiments are shown in Table 4.1. The critical delay after routing for the placement generated by *sysdias* is less than that for the placement generated by *apr* by 13 percent, on the average. For all the benchmarks *sysdias* terminated in less than 50 iterations (that took a maximum of 6.5 minutes). This is significantly faster than the simulated annealing based *apr*. The total time taken over all the benchmarks by *sysdias* is less than the total time taken by *apr* by a factor of 5. Further, the most significant gains are on the large circuits : *C880* and *alu2*, thus demonstrating the ability of *sysdias* to handle large circuits efficiently.

To demonstrate the convergence rate of our algorithm, we ran *sysdias* on some larger, randomly generated circuits with up to 800 modules. Even for such large examples the convergence rate of our algorithm is very good. Typically, the number of iterations is around 100 and the running time is no more than 10-15 minutes. The results for some of these large circuits are shown in Table 4.2. Since these circuits are too big to be placed and routed on the Xilinx 3000 series architecture, we have no information on the actual critical path delays for these circuits.

| Circuit | # modules | *apr* Delay (ns) | *apr* Time (min) | *sysdias* Delay (ns) | *sysdias* # iterations | *sysdias* Time (min) |
|---|---|---|---|---|---|---|
| cordic | 48 | 56.5 | 1.3 | 45.0 | 25 | 0.8 |
| count | 85 | 71.5 | 4.7 | 60.2 | 32 | 1.5 |
| bw | 61 | 32.1 | 3.0 | 30.0 | 20 | 1.2 |
| f51m | 58 | 92.7 | 1.8 | 80.4 | 21 | 1.2 |
| frg1 | 91 | 79.7 | 3.4 | 70.0 | 32 | 2.3 |
| comp | 103 | 90.9 | 7.7 | 77.0 | 40 | 4.3 |
| term1 | 161 | 92.9 | 24.3 | 70.6 | 45 | 5.4 |
| C499 | 144 | 79.6 | 9.5 | 70.0 | 33 | 4.3 |
| C880 | 210 | 128.3 | 28.5 | 115.2 | 42 | 5.5 |
| alu2 | 232 | 192.3 | 91.1 | 177.0 | 48 | 6.5 |

**Table 4.1**: Comparison of critical delay after routing for *apr* and *sysdias*.

| Circuit | # modules | Grid Size | Initial Delay (ns) | Final Delay (ns) | # iter | T (min) |
|---|---|---|---|---|---|---|
| Random1 | 400 | 25 x 25 | 620 | 350 | 70 | 8.4 |
| Random2 | 550 | 25 x 25 | 710 | 500 | 50 | 6.2 |
| Random3 | 700 | 25 x 30 | 600 | 450 | 88 | 12.4 |
| Random4 | 800 | 30 x 30 | 1230 | 700 | 103 | 15.0 |
| Random4 | 800 | 40 x 40 | 1450 | 650 | 90 | 12.4 |

**Table 4.2**: Results for the placement of large randomly generated circuits. *# iter* refers to the number of compression-relaxation iterations, and $T$ refers to the running time of the algorithm.

The required times at the outputs were generated using the depth of the output as an estimate. The *Initial Delay* column gives the estimated delay of the critical path in the initial placement and the *Final Delay* column gives the (estimated) critical path delay in the final placement. Notice that when the circuit *Random4* is placed on a larger grid, the convergence is faster, possibly due to the fact that compression can be done more aggressively due to low congestion.

It is interesting to observe the manner in which the vector of arrival times at the primary outputs changes from iteration to iteration. Table 4.3 shows the arrival times at the outputs of a randomly generated example with 100 modules, 10 inputs and 3 outputs, being placed in a 12 by 12 array of slots. In most of the iterations there is a substantial decrease in one of the arrival times, corresponding to the cone that is compressed in that iteration. The arrival times at the other outputs might increase slightly due to the relaxation phase in an iteration. However, the use of slack neighborhoods for relaxation ensures that the delay increase is bounded. In fact,

| Iteration # | Arrival time (ns) | | |
|---|---|---|---|
| | Out1 | Out2 | Out3 |
| 0 | 120.5 | 220.0 | 130.6 |
| 1 | 130.2 | 150.0 | 125.0 |
| 2 | 90.0 | 130.0 | 130.7 |
| 3 | 95.2 | 135.6 | 90.3 |
| 4 | 90.2 | 120.0 | 90.3 |
| 5 | 100.3 | 115.0 | 100.4 |
| 6 | 90.0 | 110.0 | 85.0 |

**Table 4.3**: The arrival times at primary outputs in successive iterations.

the sum of the arrival times at the outputs decreases from one iteration to the next, most of the time (in Table 4.3 the exception is from iteration 4 to iteration 5). This provides strong evidence that our iterative compression-relaxation based algorithm converges well.

## 4.7 Conclusions

We present a new algorithm for performance driven placement when the underlying architecture is regular. Our iterative algorithm has two phases in each iteration: a compression phase and a relaxation phase. We develop a novel compression strategy based on the longest path tree of a cone. We introduce the concept of a slack neighborhood graph and use it to transform an infeasible placement produced in the compression phase to a feasible one using a mincost maxflow formulation. The slack neighborhood graph approach guarantees that relaxation does not undo the delay reduction achieved during compression. Our analytical results regarding the bounds on delay increase during relaxation are validated by the rapid convergence of *sysdias* on benchmark circuits.

One avenue of further research is to extend our approach to more general, non-regular architectures, such as macro-cell based designs, by using a multi-commodity mincost flow in the relaxation phase.

# Chapter 5

# Reconfiguration for Fault Tolerance and Yield Enhancement

In this chapter we examine the problem of reconfiguring the *placement* of a combinational circuit on an FPGA to tolerate a given fault pattern in the array of CLBs. The primary objective of the reconfiguration is to minimize the timing degradation, measured as the increase in the delay in the critical paths in the implementation.

Introduction of fault tolerance in VLSI devices has been advocated as a means of achieving both longer lifetime (through on line reconfiguration to avoid faults) and higher production yield (through production time reconfiguration of partially faulty devices). Reconfiguration problems for arrays of identical processing elements interconnected by a regular structure have been extensively studied [2], [25], [41], [42]. Memories have also been studied from the viewpoint of reconfiguration for yield enhancement [24]. However, none of these traditional reconfiguration approaches address the issue of timing driven reconfiguration. Also, in traditional reconfiguration of faulty processor arrays, the objective is to extract the largest possible sub-array consisting only of non-faulty processors. In our case the topology to be extracted from the faulty array of CLBs depends on the circuit for which the reconfiguration is being done.

FPGAs are ideally suited for fault tolerance through reconfiguration because of the ease of reprogramming inherent in their architecture. That a CLB is capable of implementing *any* boolean function with up to $K$ inputs gives the FPGA architecture a uniformity which plays a critical role in making efficient reconfiguration possible. Also, in most implementations of combinational circuits on FPGAs there are unused CLBs (typical utilization of CLBs is around 60-70%) which can be used as spares when some CLBs become faulty. Howard et al. [19] explored various fault-tolerant FPGA architectures. Narasimhan et al. [37] investigated algorithms for reconfiguring FPGAs using minimization of the total wire length of the reconfigured design as the primary objective. Since FPGAs are increasingly being used to implement high performance ASICs, it is important to do reconfiguration so as to minimize the timing degradation. The recovery time depends on the amount of reprogramming which needs to be done. So, it is important that the reconfiguration algorithm should make an effort to reduce the amount of reprogramming required for reconfiguration. Also, the reconfiguration should be significantly faster than doing placement and routing again to avoid the faulty CLBs.

## 5.1 Fault Tolerance in FPGAs

As FPGA manufacturers are making constant technological improvements to achieve higher levels of integration and higher speeds, FPGAs are used more and more often in the final marketed designs of digital systems. This emphasizes the need for on-line algorithms for reconfiguration in FPGAs to tolerate faults in the logic blocks, in order to increase the lifetime of the product. This is even more important in real-time and embedded applications of FPGAs.

Reconfiguration is also a useful technique for *yield enhancement* by tolerating faults in the FPGAs being used to implement a design. In this scenario, a design has been placed and routed to meet timing requirements, but some faulty CLBs are discovered prior to programming the FPGA. The placement and routing needs to be changed in such a way that the performance of the implementation does not degrade significantly.

The results of Howard et al. [19] also provide compelling evidence for the use of reconfiguration for yield enhancement in FPGAs. They argue that to overcome the relatively low logic density of FPGAs many large designs need to be split across multiple FPGAs resulting in large circuit delay. To overcome this problem, the level of integration of FPGAs needs to be increased and this leads to FPGAs with larger chip area, and consequently greater probability of faults. In order to get reasonable yield for such ultra large scale FPGAs it is imperative that some reconfiguration based fault tolerance scheme be employed.

## 5.2 The Fault Model

We consider only faults in CLBs. Faults in the routing blocks are not considered. This is realistic because in the current FPGA designs the CLBs are much more complex than the routing blocks. We use a block level fault model, which assumes that any fault in a CLB completely disables the CLB. This is a much stronger fault model than the "stuck at" fault model traditionally used for modeling faults in combinational logic. Thus, when a CLB becomes faulty, it will be discarded and an attempt will be made to map its functionality on a different CLB. A more refined fault model would take into account the fact that a faulty CLB may still be able to implement some (but not all) combinational functions.

We used two different models for the distribution of faults in our experiments:

- **Independent Fault Model:** This model assumes that the faults in CLBs are independent – that is the probability that a CLB is faulty is not dependent on the state (faulty/non-faulty) of the neighboring CLBs. The probability of a CLB being faulty is referred to as the *fault probability*.

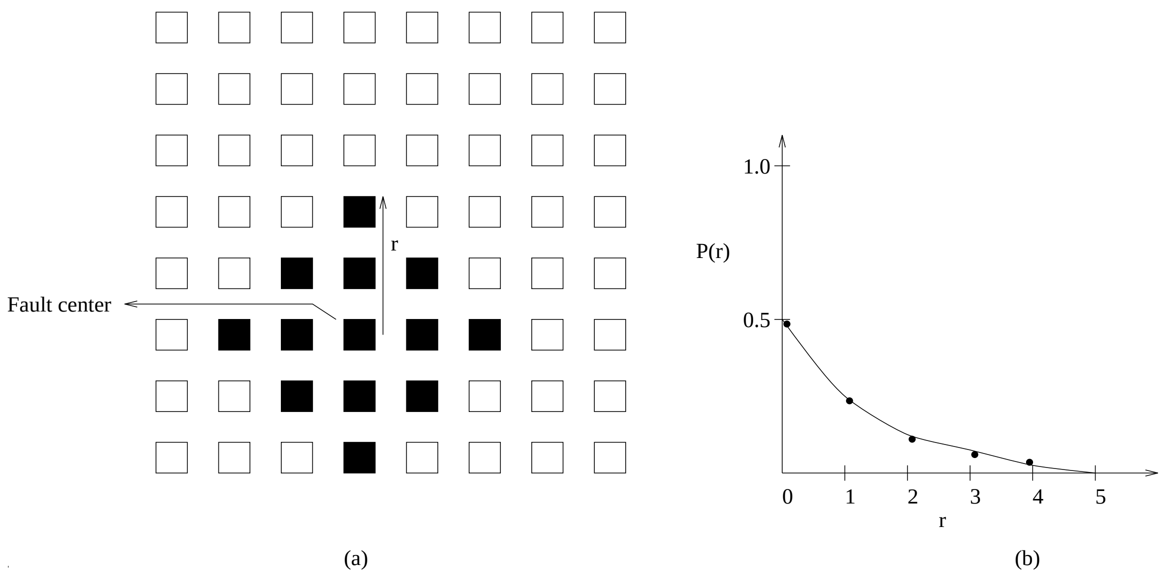


**Figure 5.1**: (a) A clustered fault of radius 2. (b) The exponential probability distribution for fault radius.

- **Clustered Fault Model:** This model assumes that faults occur in clusters. Thus, if a CLB is faulty then its neighbors have a higher probability of being faulty. We simulate this model by assuming that a CLB has a fixed probability (termed *cluster fault probability*) of being the *center* of a fault cluster of radius $r$. All the CLBs within distance $r$ from the center are assumed faulty. The radius $r$ is assumed to have an exponentially decreasing probability distribution (see Fig. 5.1).

## 5.3 Problem Formulation

The input to the timing driven reconfiguration problem is a combinational circuit, its implementation on an FPGA specified by the placement of the modules and the routing of the interconnections, and the set of CLBs identified to be faulty (the *fault pattern*). The circuit is specified by its interconnection topology, module delays and the signal arrival times and required times at the primary inputs and primary outputs respectively.

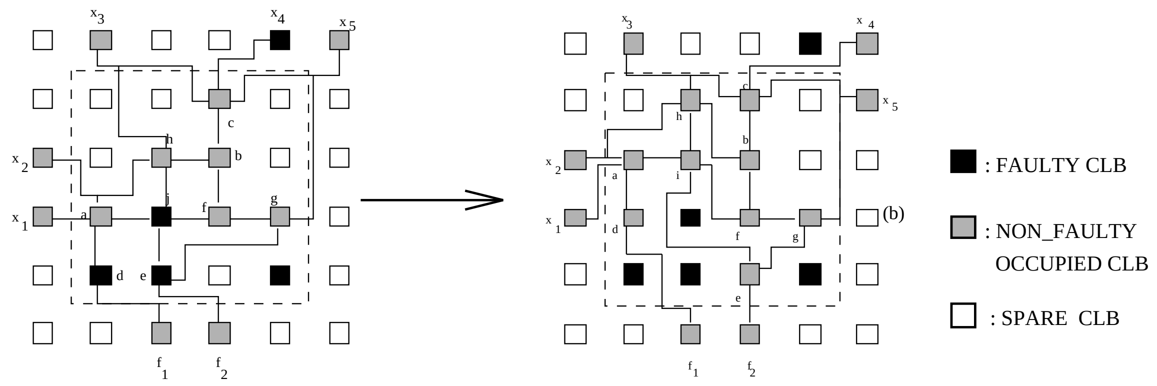


**Figure 5.2**: The reconfiguration problem.

The circuit and architecture specifications, and the delay model are the same as for the placement problem (see Chapter 4). Again we exploit the regularity of the underlying architecture in our algorithm for timing driven relocation.

We do not address the issue of testing an FPGA to detect the faulty CLBs. Since we are interested in FPGAs that are re-programmable, it is possible to test a CLB in several different configurations using techniques similar to those employed for the testing of combinational circuits. Thus, we assume that a testing program tests the FPGA and determines the set of faulty CLBs, $F \subset S$. Thus, the input to the reconfiguration algorithm is $F$ along with the current implementation of the combinational logic. The output of the reconfiguration process is a *reconfiguration function* $\rho : U \rightarrow S - F$, such that $\rho$ is a one-to-one mapping. An obvious necessary condition for the existence of $\rho$ is that there should be at least as many spare CLBs as there are faulty ones. The main objectives of the reconfiguration algorithm are:

- Minimize the increase in the critical path delay as a result of reconfiguration.

- Minimize the amount of reprogramming to be done to achieve the reconfiguration.

The first objective is a natural one for high performance ASICs designed using FPGAs. Low reprogramming overhead may be important when FPGAs are used in a system that requires high availability. However, this objective is relevant only when the FPGA architecture allows for selective reprogramming of CLBs and routing blocks. Some of the more recent FPGA architectures, such as the Concurrent Logic CLi6000 series and Algotronix FPGAs, have this feature.

## 5.4 Overview of Algorithm

The flowchart in Fig. 5.3 gives an overview of the main steps involved in our timing driven reconfiguration algorithm. Our reconfiguration algorithm attempts to relocate all the modules currently mapped to faulty CLBs in such a way that increases in the delays will not be excessive. To this end, we compute the slacks for the edges in the circuit graph. The slack of an edge measures the amount by which the delay of the edge can be increased without violating any timing constraints. The slacks of the edges incident to a module determine the neighborhood within which the module can be moved without violating the timing requirements. However, if a certain increase in critical path delay is acceptable, it is not necessary to have the mobility of the modules completely restricted by the slacks. We introduce a *relaxation parameter*($\lambda$) that allows the reconfiguration algorithm to increase the values of edge slacks. The slacks augmented by the relaxation parameter are referred to as *relaxed slacks*. The relaxed slacks are then used to compute a *slack neighborhood graph* (SNG) in which an edge from vertex $u$ to vertex $v$ indicates that the module corresponding to vertex $u$ can be moved to the CLB currently occupied by the module corresponding to vertex $v$ *without violating any timing constraints*. In order to incorporate a routability measure into the reconfiguration process, each edge of the SNG is associated with a cost that measures the penalty, in terms of increase in congestion,

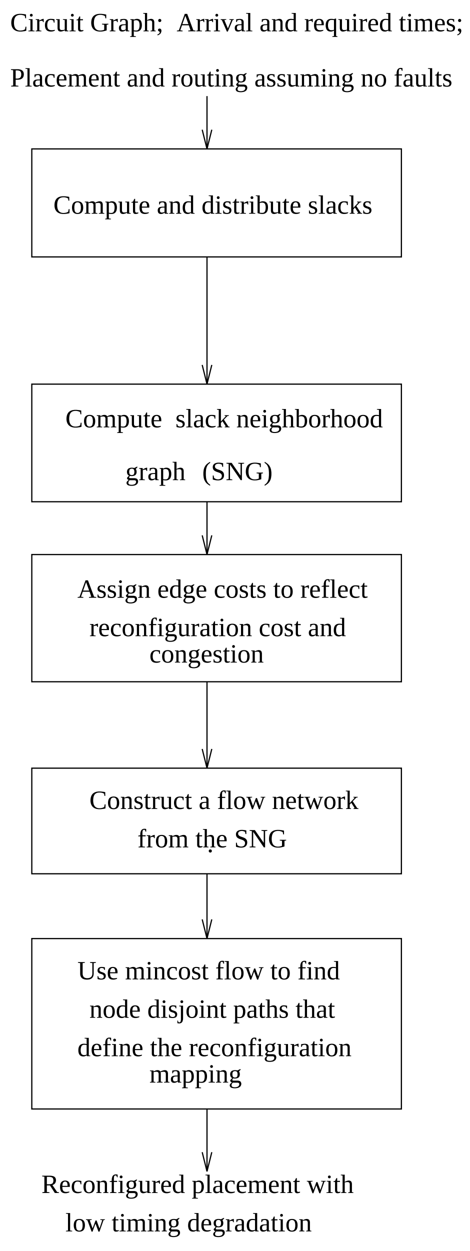


**Figure 5.3**: Flowchart for our timing driven reconfiguration algorithm.

that results from the move associated with that edge. Another factor that goes into the cost computation is the amount of reprogramming that is required due to the move associated with that edge.

Once the weighted slack neighborhood graph has been computed, a min-cost flow formulation is used to find vertex disjoint paths in the graph. These paths are then used to relocate the modules that were mapped to faulty CLBs. That reconfiguration is being carried out in the slack neighborhood graph guarantees that the paths in the circuit will not become excessively long. It might happen that the SNG is not dense enough to allow the relocation of all the modules mapped to faulty CLBs. In such a scenario, the relaxation parameter is increased to allow more edges in the SNG, leading to a successful reconfiguration. Unlike several other reconfiguration algorithms, our algorithm does not simply compute a matching from the faulty CLBs to the spare CLBs. A min-cost flow in SNG in general leads to a better solution. Fig. 5.4 illustrates the steps involved in our timing driven reconfiguration algorithm.

## 5.5 Experimental Results

We implemented the slack neighborhood graph based algorithm for timing driven reconfiguration in C on a sparc10. In this section we discuss the results obtained by running our algorithm on a number of benchmark circuits assuming random fault patterns with different fault probabilities.

We performed two sets of experiments. In the first set of experiments fault patterns were generated using the independent fault model, while in the second set of experiments fault patterns were generated using the clustered fault model described in Section 3. The input for each run of the algorithm consisted of an initial placement that satisfies all the timing constraints and a pattern of faulty CLBs. The initial placements were generated using the *sysdias* package [30]. Ten different fault patterns were generated for each value of fault probability ranging from 0.015 to 0.055 (in steps of 0.005). The size of the CLB array was 20 by 20 for all the experiments. For each circuit, reconfiguration was done using values of the relaxation parameter ranging from 0 to 30 (where a value of 10 corresponds to a distance of 1 in the CLB grid array) to study the relationship between timing degradation and the relaxation parameter. Timing degradation for a circuit as a result of reconfiguration was measured as the percentage increase in the average

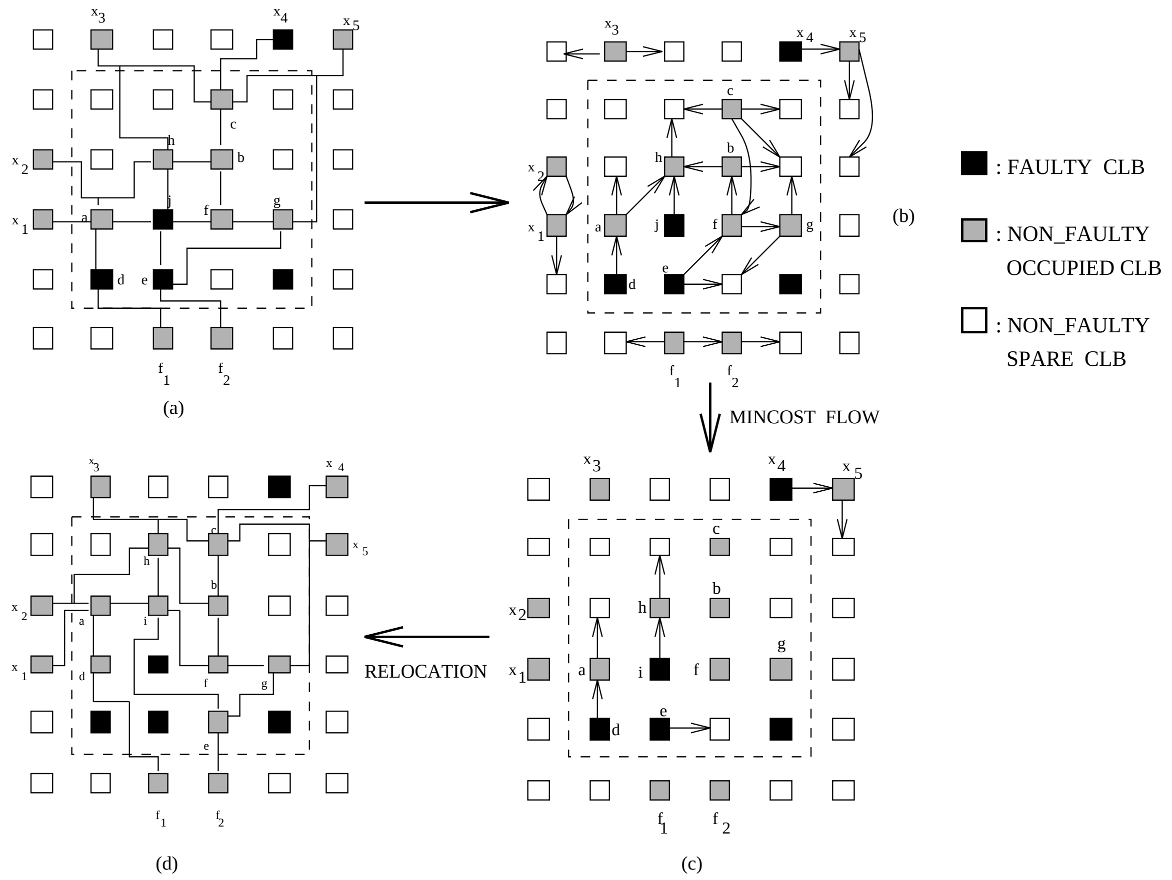


**Figure 5.4**: An example illustrating the timing driven reconfiguration algorithm. (a) A placed and routed circuit and a fault pattern. (b) The slack neighborhood graph (SNG) for the circuit in (a). (c) The vertex disjoint paths in the SNG computed using min-cost flow. (d) The reconfigured circuit.

of the arrival times at all the primary outputs. The variation in the *maximum* of the arrival times at the primary outputs was also observed to be similar to the average arrival time(in fact for all our experiments the degradation in the maximum arrival time was less than that in the average arrival time). Table 5.1 summarizes the average timing degradation for the 5 circuits used in our experiments for various values of the relaxation parameter. Negative values of the timing degradation indicate a decrease in the total arrival time after reconfiguration. Notice that for circuits C432 and C499 the timing degradation is consistently higher for the clustered fault model than for the independent fault model. However, for circuits c8, b9 and ttt2 the degradation is almost identical for the two models. This is probably because these circuits have fewer modules and hence the number of spares available is higher, making reconfiguration easier. The CPU time for one reconfiguration run was around 15-20 seconds.

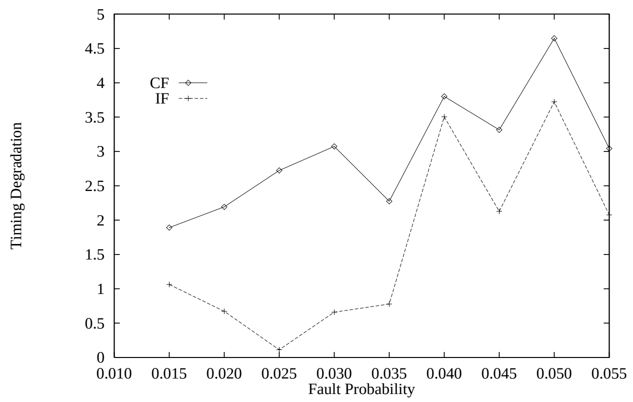


**Figure 5.5**: Results for circuit C499 with RP = 30. IF = Independent Fault Model; CF = Clustered Fault Model.

The graphs in Figures 5.5 – 5.6 show the timing degradation as a function of fault probability for some of the circuit/relaxation parameter pairs. The worst timing degradation was observed for C432, and it was around 6%. The average timing degradation for C432 was 0.7% for the independent fault model and 1.3% for the clustered fault model. For all the other circuits we tested, the average timing degradation was less than 1%. For many of the reconfigurations (specially with low fault probability) there was actually a decrease in the average arrival time. This demonstrates that our timing driven reconfiguration algorithm is able to successfully reconfigure faulty FPGAs with very low timing penalty. In general, with increase

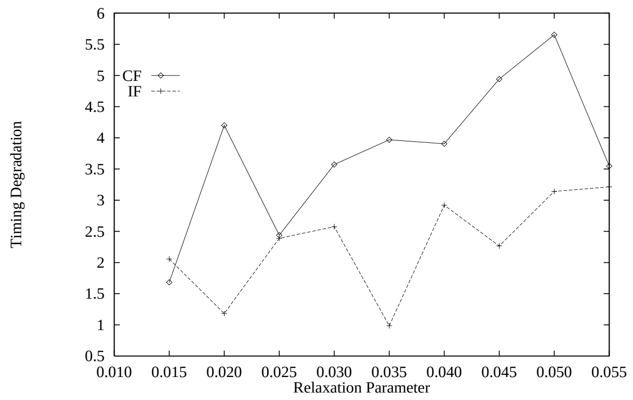


**Figure 5.6**: Results for circuit C432 with RP = 30.

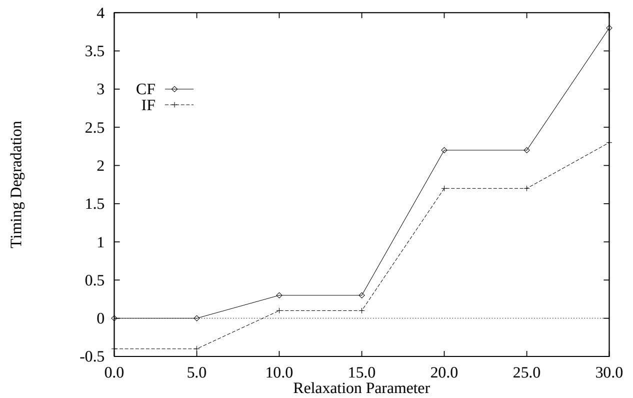


**Figure 5.7**: Variation of timing degradation with relaxation parameter for circuit C432.

| | RP = 0 | | RP = 10 | | RP = 15 | | RP = 20 | | RP = 25 | | RP = 30 | | Average | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Circuit | IF | CF | IF | CF | IF | CF | IF | CF | IF | CF | IF | CF | IF | CF |
| C499 | -0.1 | -1.5 | -0.4 | -0.8 | -0.4 | -0.8 | 0.1 | 0.7 | 0.1 | 0.7 | 1.1 | 1 .6 | 0.1 | -0.2 |
| C432 | -0.4 | 0.0 | 0.1 | 0.3 | 0.1 | 0.3 | 1.7 | 2.2 | 1.7 | 2.2 | 2.3 | 3.8 | 0. 7 | 1.3 |
| c8 | -0.1 | 0.0 | 0.1 | -0.1 | 0.1 | -0.1 | 0.2 | -0.1 | 0.3 | -0.1 | 0.2 | 0.1 | 0.1 | -0.0 |
| b9 | -0.1 | -0.8 | -0.5 | -1.0 | -0.7 | -1.0 | -0.2 | -0.3 | -0.2 | -0.3 | 0.3 | 0.4 | -0.2 | -0.6 |
| ttt2 | -0.0 | -0.1 | 0.1 | -0.0 | 0.1 | 0.0 | 0.2 | -0.0 | 0.2 | -0.1 | 0.2 | 0. 1 | 0.1 | -0.0 |

**Table 5.1**: Average percent timing degradation as a function of the relaxation parameter (RP). IF = Independent Fault Model; CF = Clustered Fault Model.

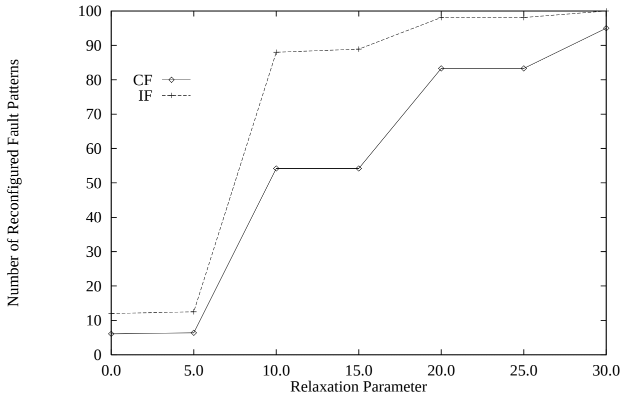


**Figure 5.8**: Number of successfully reconfigured fault patterns (measured as percent of the total number of fault patterns) vs relaxation parameter.

in fault probability there is an increase in the timing degradation since the presence of more faults restricts the slack neighborhoods and increases the value of flow required in the SNG for successful reconfiguration.

It is interesting to study the variation in timing degradation with changes in the relaxation parameter. As was predicted by Corollary 3.1, there is no timing degradation when the reconfiguration is done with relaxation parameter set to zero. However, several fault patterns cannot be reconfigured when the relaxation parameter is zero. Increasing the relaxation parameter enables us to reconfigure more fault patterns. However, the timing degradation increases. This behavior is observed in the Figures 5.7 and 5.8. Also, more faults can be reconfigured when the fault patterns are generated using the independent fault model than for the clustered fault model.

## 5.6 Conclusions

We presented a new framework for timing driven reconfiguration for fault tolerance and yield enhancement for FPGAs using slack neighborhood graphs. It is possible to apply this technique to other regular architectures such as wafer scale integrated arrays of processors. The use of min-cost flow in the slack neighborhood graph to find vertex disjoint paths for reconfiguration results in provably low timing degradation, as shown by our analytic and experimental results.

# Chapter 6

# Timing Driven Placement Re-engineering

In a typical design flow, the design may be altered slightly several times after the initial design cycle according to minor changes in the design specification either as a result of design debugging or as a result of changes in engineering requirements. Such changes in design specification are referred to as *engineering changes*. The process of generating a new design that incorporates the modifications to the circuit is referred to as *design re-engineering*. Specifically, we are interested in re-engineering a placement under timing constraints : a problem we call *timing driven placement re-engineering*. After the initial design cycle, a completely placed and routed design that satisfies all the timing constraints is available. The input to the problem of timing driven placement re-engineering is a placed and routed design and the engineering changes made to the circuit. The output is a new layout that incorporates the design changes and satisfies all the (possibly new) timing constraints (see Fig. 6.1). Since the engineering changes are typically local and relatively minor, it would be wasteful to re-do the entire layout of the modified circuit from scratch. Further, the localized nature of engineering changes means that the layout of the original circuit is a good starting point for finding a delay feasible layout of the modified circuit. An algorithm for minimizing the perturbation in the placement while handling engineering changes was presented in [6]. Techniques have also been proposed for exploiting the locality of engineering changes in circuit simulation [22]. More recently there has been an effort to integrate logic synthesis and layout through the use of local logic restructuring driven by routability considerations [5]. Such a technique for integration of logic synthesis and layout would benefit greatly from the use of an efficient algorithm for timing driven placement re-engineering.

We study the problem of timing driven placement re-engineering for *regular architectures*. The two architectures of interest are Field Programmable Gate Arrays (FPGAs) and gate arrays. The concept of a slack neighborhood graph is used to design an efficient algorithm for reconfiguring a physically infeasible placement (with some slots occupied by two modules) to a feasible one with provably low timing degradation. This serves as a crucial part of our algorithm for the timing driven placement re-engineering problem. Our algorithm exploits the regularity of the underlying architecture to ensure that the execution time is significantly less than that for re-doing the placement from scratch for the modified circuit. The regularity of these architectures allows us to use powerful graph theoretic techniques that would not be possible otherwise.

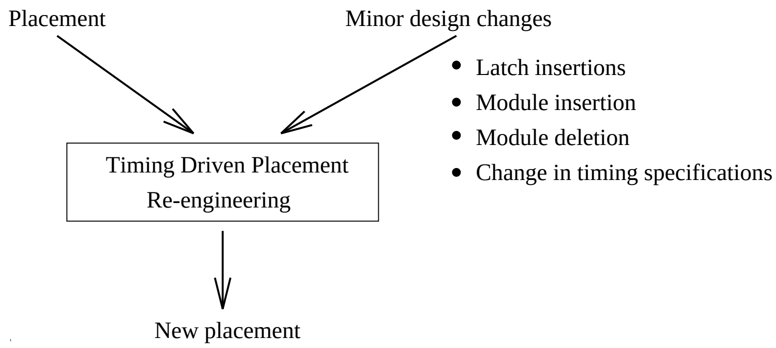


**Figure 6.1**: The timing driven placement re-engineering problem.

FPGAs are becoming increasingly important in the design of Application Specific Integrated Circuits (ASICs) since they provide both large scale integration and user-programmability. User-programmability results in shorter turn-around time and lower manufacturing costs. As constant technological improvements lead to higher levels of integration and higher speeds, FPGAs are becoming more prevalent in the final marketed designs of digital systems. This makes it essential to take timing into consideration at all stages in the implementation of a circuit on an FPGA.

We examine two kinds of engineering changes: (1) **Structural changes** and (2) **timing constraint changes**. A structural change modifies the topology of the circuit while a timing constraint change alters the timing constraints that must be satisfied by the implementation of the circuit. Since the deletion of modules and the associated interconnections requires only the removal of the modules and interconnections from the placement, it does not increase the arrival times of signals at the primary outputs. Hence, we will focus on the addition of new modules and interconnections to a placed and routed design. Notice that the addition of new modules and interconnections can cause violation of timing constraints, even if there is no change in the timing constraints, due to the creation of new paths in the circuit (see Fig. 6.3). Timing changes modify the module delays and the timing constraints (in the form of arrival times at the primary inputs and required times by which signals should arrive at the primary outputs) that must be satisfied by a delay feasible placement.

Figure 6.2 shows an overview of our framework for handling engineering changes. For structural changes, we have an initial placement phase where an affinity based maximum matching algorithm is used to decide the initial positions of the new modules. The initial placement algorithm is allowed to place the new modules on slots that are already occupied. Hence, a physically infeasible intermediate placement may result. This is transformed to a physically feasible placement using timing driven relocation based on the slack neighborhood graph approach. If the new placement violates some timing constraints, the compression-relaxation based scheme proposed in [30] is employed to generate a delay feasible placement. Since the changes to the circuit are minor and we start out with a delay feasible placement for the original circuit, the number of compression-relaxation iterations required is relatively small. For timing changes involving the tightening of some constraints, we use compression-relaxation iterations if some of the tighter constraints are violated by the current placement.

The organization of the remainder of the chapter is as follows: Section 2 is devoted to a more precise formulation of the placement re-engineering problem; Section 3 gives an overview of our approach to placement re-engineering for regular architectures; Section 4 discusses the problem of generating a good initial placement for the new modules added to the circuit; Section 5 describes our approach to timing driven relocation for mapping a physically infeasible placement (having overlaps between some modules) to a physically feasible one without significant degradation in circuit performance; Section 6 describes our approach to handling timing

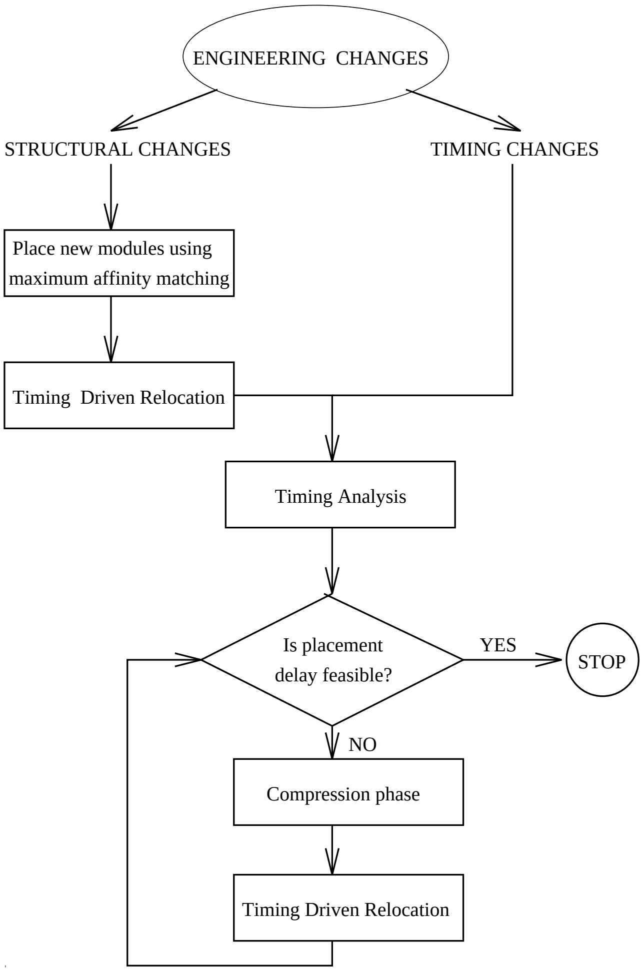


**Figure 6.2**: An overview of our approach to timing driven placement re-engineering.

violations either due to structural changes or due to tightening of timing constraints; finally Section 7 presents the results of our experiments with the Xilinx 3000 series FPGA architecture.

## 6.1 Placement Re-engineering

After an iteration of the design flow, we have a placed and routed design $I$ that will be referred to as the *current implementation.* The *current circuit specification* is $P = \langle C, T \rangle$, where $C$ represents the current circuit topology and $T$ is the set of *timing constraints* that must be satisfied by a *delay feasible* implementation.

- $C$ is represented by a directed graph $G_C(V_C, E_C)$ where each module of the circuit is represented by a vertex in $V_C$ and an interconnection between two modules is represented by an edge in $E_C$ between the corresponding vertices (see Fig. 1.2).
- The timing constraints consist of arrival times $A(x_i)$ at each primary input of the circuit, the module delay, $d(M_i)$, for each circuit module and the required time, $R(f_i)$, by which the signal should arrive at primary output $f_i$.

The current implementation, $I$, is an implementation of the current specification $P$. The circuit could be either combinational or sequential, however for the purpose of timing calculations we assume that the circuit has been cut at all the latches (thus creating pseudo-primary inputs and outputs). We assume that $T$ contains information about the arrival times and required times at these pseudo primary inputs and outputs. Thus, the timing calculation algorithms will assume that the underlying graph is a directed *acyclic* graph.

In *placement re-engineering* we are given the current circuit specification, $P$, the current implementation, $I$, and some engineering changes. The output of a placement re-engineering algorithm is a delay feasible placement for the new specification, $P'$. Further, it is desirable that the running time of the algorithm be proportional to the number of engineering changes.This makes it necessary to make use of $I$ in generating the new delay feasible placement for $P'$.

We focus on the re-engineering problem for regular architectures (see Chapter 1). Such an architecture can be viewed as a two dimensional array of identical slots: $S = \{S_1, S_2, \ldots, S_m\}$, along with routing resources (routing channels, pre-placed interconnections, programmable switches, etc.) A characteristic common to all these architectures is that it is easy to estimate interconnection delay without performing detailed routing because of their regularity. An important class of regular architectures is the Xilinx symmetric FPGA architectures. Traditional gate arrays and wafer-scale integrated arrays of processors are other examples of regular architectures. Further, there are several architectures that are *almost regular*. Standard cell architectures that have a few different types of cells fall into this category and it is possible to extend most of our algorithms to such architectures.

### 6.1.1 Engineering Changes

Typically, engineering changes have the following characteristics:

1. **Locality:** Engineering changes result from local restructuring of the circuit. So, they do not completely change the structure or function of the circuit. Changes that have global effect on the timing behavior or functionality of the circuit will not be classified as engineering changes.

2. **Small size:** The *zone of influence* of a change can be defined as the portions of the circuit that are affected by the change. Re-engineering changes should have a zone of influence that is a small fraction of the entire circuit. Since it is hard to estimate the zone of influence of a change, we will assume that the number of gates and nets that get changed by a set of engineering changes is a small fraction of the size of the circuit.

Engineering changes can be classified into two broad categories depending on which component of the specification tuple $P\langle C, T\rangle$ is changed.

- **Structural changes:** These changes alter the structure of the circuit and result in changes to the circuit topology. The structural changes that we consider are:

  1. *Module Addition:* Addition of new circuit modules along with the associated interconnections.

  2. *Module Deletion:* Deletion of some of the existing modules and their connections to other modules.

  3. *Interconnect Addition:* Addition of new interconnections among existing modules.

  4. *Interconnect Deletion:* Deletion of interconnections between circuit modules.

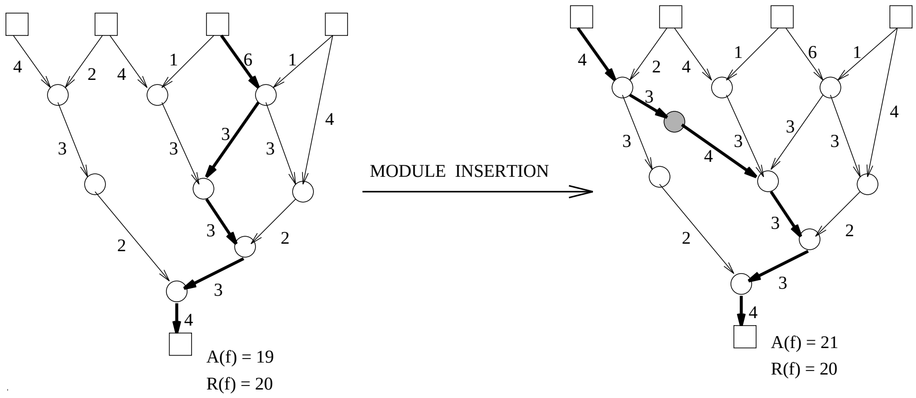


**Figure 6.3**: Structural changes can result in violation of timing constraints. Addition of the shaded module results in a new critical path (dark path) that increases the arrival time at the output to 21 causing violation of the required time constraint.

- **Timing changes:** These changes alter the timing constraints imposed on any delay feasible implementation of the circuit. We consider the following two kinds of timing changes:

  1. *Change in arrival time at a primary input or required time at a primary output.*
  2. *Change in delay of a circuit module.*

When a module or interconnection is deleted from the circuit, the implementation of the circuit can be easily updated to reflect the change by simply removing some geometries from the layout. Since this will not increase the arrival time of any signal, the resulting layout is delay feasible. So, from the point of view of timing driven placement re-engineering, deletions are trivial to handle. We will henceforth focus only on structural changes involving the addition of modules and/or interconnections and timing changes.

## 6.2 Overview of Our Approach

Figure 6.2 gives an overview of our approach to timing driven placement re-engineering. For structural changes involving addition of new modules along with the associated interconnections (we ignore module and interconnect deletions since they only involve the removal of some geometries from the layout), we have an initial placement phase followed by an iterative compression-relaxation phase if the initial placement is not delay feasible. For timing changes, we perform timing analysis with the modified constraints and if the current placement (which could be the placement generated by the initial placement phase for handling structural changes) is not delay feasible the iterative compression-relaxation scheme is used.

Our algorithm for generating the initial placement for new modules added as part of structural changes to the circuit is characterized by the use of a physically infeasible intermediate placement in the process of generating a new placement. We make use of an efficient algorithm for timing driven relocation in transforming a physically infeasible placement that has some slots occupied by two circuit modules to a physically feasible one with low timing degradation. The concept of a slack neighborhood graph is introduced and demonstrated to be useful in guaranteeing low timing degradation during the movement of modules in timing driven relocation. Timing driven relocation is also used in the relaxation phase of the compression-relaxation iterations.

In order to clarify the problem of timing driven placement re-engineering and motivate our approach, let us consider an example of an engineering change where a new module is being added to a circuit that has already been placed. Figure 6.4 shows an example of a new module being added to a circuit that is already placed and routed. Figures 6.4 (a) – (c) show the steps involved in our algorithm for handling module insertion. Figure 6.4 (d) shows the new placement produced by a naïve algorithm that simply searches for a vacant slot in the vicinity of the slots to which the new module has high connectivity. Notice that if a naïve algorithm is used for the placement of the new module, the new module is placed in a slot that is quite far from the modules to which it has high connectivity. This is because the slots in the vicinity of the new module's neighbors are occupied. On the other hand, our algorithm allows the new module to be placed on an occupied slot in the initial placement phase and then performs a timing driven relocation of the modules to remove overlaps.

Our algorithm for the initial placement of new modules consists of the following two steps:

1. **Affinity based initial placement of the new modules:** The placement of the new modules is determined on the basis of the "affinity" between new modules and the modules in the old circuit. One measure of affinity between a new module and an old one could be the number of interconnections between them. However, this measure of affinity is "wire length minimization" oriented. For timing driven placement one could define the affinity of a new module to a slot as the inverse of the timing degradation in the circuit

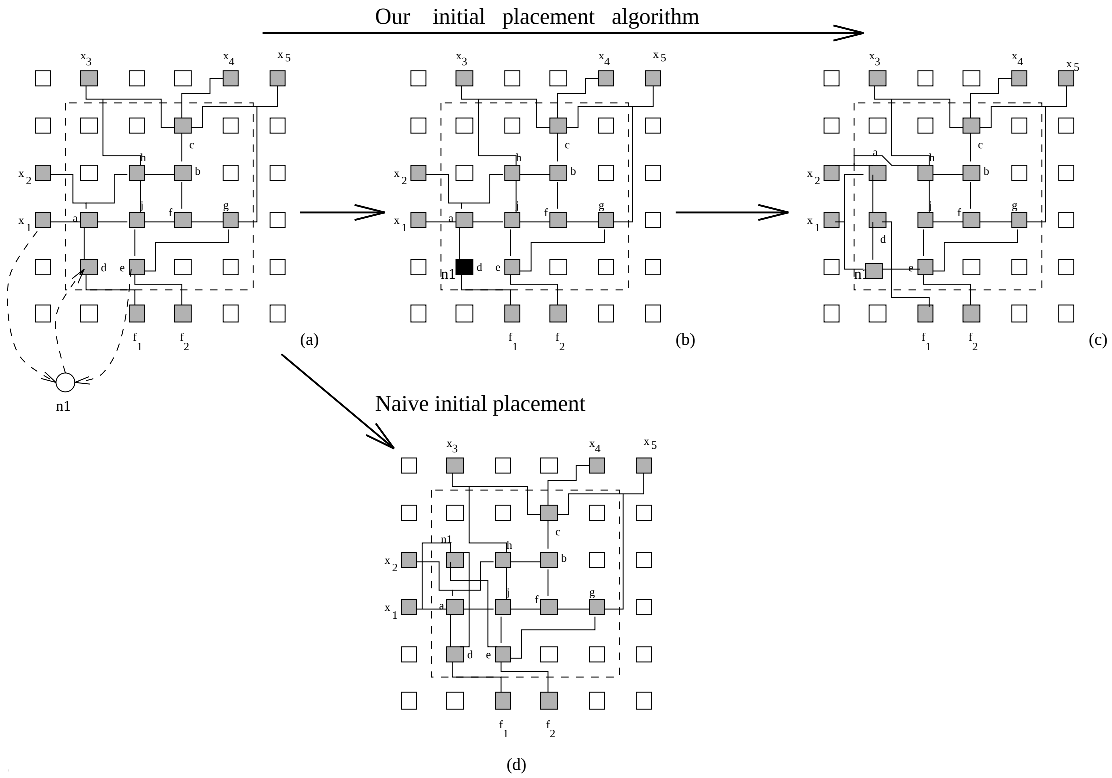


**Figure 6.4**: Comparison of our initial placement algorithm with a naïve one that places the new module in an empty slot close to the neighbors of the new module. (a) A placed and routed design to which a new module, $n1$, is being added. (b) A placement is determined for the new module using maximum affinity matching creating an overlap (shown as the dark slot). (c) New placement after timing driven relocation to remove the overlap. (d) The new placement generated by an algorithm that tries to find an empty slot near the modules to which the new module has high connectivity.

as a result of placing the new module at that slot. Notice that the affinity function for a new module is computed between it and all the slots, including those that are currently occupied. This is important since there may not be any empty slots for which the new module has high affinity. The affinity function can be used to find the "maximum affinity matching" between the new modules and the slots. This matching defines the placement of the new modules. However, the placement of the new modules may create overlaps with old modules resulting in an infeasible placement. This step is illustrated in Fig. 6.4 (a) and (b).

2. **Timing Driven Relocation**: If the placement becomes infeasible as a result of placing the new modules using maximum affinity matching, then we need the relocate the old modules occupying the infeasible slots in such a way that the timing degradation is minimized. This is precisely the timing driven relocation problem. This step is shown in Fig. 6.4 (b) and (c).

The above approach tries to minimize the timing degradation in the process of addition of new modules, but does not guarantee that the new placement will meet all the timing constraints. If the designer is willing to tolerate some timing degradation in the process of re-engineering then this approach is fine. However, if the new placement must be delay feasible then we need to use the compression-relaxation approach. If a timing constraint is tightened and the current implementation does not satisfy the tighter constraint, then we use the compression-relaxation approach to transform the current placement to one that is delay feasible. Since we start from a placement that was delay feasible for the original timing constraints, and the modification to the timing constraint is small, we expect to obtain a delay feasible placement after a few compression-relaxation iterations.

## 6.3 Affinity Based Placement

The first phase of our algorithm for handling the addition of new modules along with their interconnections involves selecting a good initial placement for the new modules. At this stage we allow the new modules to be placed in slots that are already occupied, thus resulting in a physically infeasible placement. Allowing a physically infeasible intermediate placement results in a significant increase in the quality of the final placement (see Fig. 6.4).

We will now present the details of the affinity based placement scheme. We denote the *affinity* of a module $M_i$ to slot $S_j$ by $A(M_i, S_j)$. Intuitively, the affinity of a new module to a slot is a measure of the quality of the new placement that results when the module is placed in that slot. Increase in the total wire length and the total increase in the arrival times of signals at the primary outputs are both taken into consideration in the computation of the affinities. If there is a large increase in the total wire length or the total of the arrival times of the signals at the primary outputs by placing the new module $M_i$ at slot $S_j$, then $A(M_i, S_j)$ is low. If a new module is placed in a slot that is not occupied, then it will not force any relocation of modules in the second phase of our algorithm. The affinity function takes this into consideration by having a term that depends on the status (vacant or occupied) of the slot. Thus, the affinity is defined by the following weighted sum

$$A(M_i, S_j) = \alpha \cdot (\Delta L(M_i, S_j))^{-1} + \beta \cdot (\Delta T(M_i, S_j))^{-1} + \gamma \cdot \mu(S_j),$$

where

$$\begin{aligned}
\Delta L(M_i, S_j) &= \text{total increase in the lengths of all the nets when module } M_i \text{ is placed on slot } S_j, \\
\Delta T(M_i, S_j) &= \text{total increase in the arrival times of signals at all the primary outputs of the} \\
&\quad\ \text{circuit when module } M_i \text{ is placed on slot } S_j, \\
\mu(S_j) &= \begin{cases} 1 & \text{if } S_j \text{ is vacant} \\ 0 & \text{if } S_j \text{ is occupied} \end{cases}
\end{aligned}$$

Notice that the definitions of $\Delta L(M_i, S_j)$ and $\Delta T(M_i, S_j)$ assume that $M_i$ is the only new module being added to the circuit and compute the effect of placing it on slot $S_j$ on the total wire length and total arrival time respectively. The effect of the other new modules being added to the circuit are ignored in the calculation of $A(M_i, S_j)$. Any attempt to simultaneously capture the effect of adding several modules to the circuit results in a drastic increase in the time required for computing the affinities. The computation of $\Delta L$ takes time proportional to the number of nets incident to the module for while the affinity is being computed. The computation of $\Delta T$ requires time proportional to the size of the fanout cone of the new module. Thus, the total time consumed in the computation of the affinities is $O(k \cdot (|E_C| + |V_C|) \cdot n)$

where $n$ is the number of slots in the architecture and $|E_C| + |V_C|$ is the size of the circuit graph.

We use the affinities computed above to place the new modules in slots in the underlying architecture. This is done by finding a *maximum matching* in a weighted bipartite graph, $G(V_M \cup V_S, E)$ where $V_M$ is the set of vertices corresponding to the new modules being added to the circuit, $V_S$ is the set of vertices corresponding to the slots in the architecture. The edges in $E$ are of the form $(u, v)$, $u \in V_M$, $v \in V_S$. Further, each edge $(u, v) \in E$ has a weight equal to the affinity of the module represented by vertex $u$ to the slot represented by vertex $v$. Since we allow for the possibility of a module being placed in any of the slots, the graph $G$ is a *complete* bipartite graph. So a maximum matching of size $|V_M|$ always exists (assuming $|V_M| < |V_S|$). Our algorithm for generating the initial placement of the new modules finds a maximum weight matching in $G$ using mincost flow in a modified graph having unit node capacities [17]. Such a matching yields a placement of the new modules, with the module corresponding to vertex $u \in V_M$ being placed on the slot corresponding to vertex $v \in V_S$ if edge $(u, v)$ is in the maximum weight matching. Since we have computed a maximum weight matching the resulting initial placement maximizes the sum of the affinities of the new modules to the slots on which they get placed. At the same time, no two new modules are placed on the same slot. This ensures that in the placement that results, a slot is occupied by at most two modules.

Thus after initial placement of the new modules we end up with a placement that may be physically infeasible. If the placement is physically infeasible we need to relocate the old modules occupying the "overcrowded" slots to vacant slots. Further, in the process of relocation it is essential to ensure that timing degradation is minimized. This is accomplished by using *timing driven relocation.* We use the algorithm described in Chapter 3 to perform timing driven relocation using slack neighborhood graphs. As in the relaxation phase of *sysdias* (see Chapter 4), the "overcrowded" slots constitute the infeasible set and the vacant slots are the spares to which modules need to be relocated. After timing driven relocation we obtain a physically feasible placement, but some timing constraints may still be violated by this placement. In the next section, we discuss our approach to resolving violation of timing constraints.

## 6.4 Handling Timing Violations

Once we have a physically feasible placement for the modified circuit we perform timing analysis using the new timing specification. If the current placement violates some timing constraints, we use an iterative algorithm based on the compression-relaxation approach of [30] to produce a delay feasible placement. This section provides a brief overview of the compression-relaxation approach. Details can be found in Chapter 4.

The compression phase attempts to make the placement delay feasible by compressing the long paths that cause some of the primary output signals to arrive too late. However, the compression phase may produce an infeasible placement with some of the slots occupied by two (but no more than two) modules. This allows the compression phase more flexibility and often allows it to achieve the required decrease in delay. If an infeasible placement is produced in the compression phase, the relaxation phase, which carries out a *timing driven relocation* to produce a physically feasible placement. The slack neighborhood graph based algorithm for timing driven relocation described in Section 5 is used in the relaxation phase.

A delay analysis is carried out on the current placement to compute the actual arrival times of the signals at the various outputs and the outputs where the timing requirement is violated are identified. Associated with each such output is a set of long paths ending at the output, that are responsible for the violation of the timing requirement. We define the *longest path tree* (LPT) of a primary output, $f_i$, to be the subgraph of the cone of $f_i$ consisting of the edges in the longest path from each vertex in the cone of $f_i$ to the primary output $f_i$. The LPT captures several longest paths in the cone of $f_i$ simultaneously. The compression algorithm attempts to satisfy the timing requirement at $f_i$ by moving modules in the cone to new slots so as to decrease edge delays, a process referred to as *compressing the cone at* $f_i$. It should be noted that in the process of compressing $cone(f_i)$, up to two modules can be placed in the same slot. This ensures that compressing of the cone has enough flexibility to guarantee a substantial decrease in the critical path delay of the cone. If there are several outputs at which the arrival time exceeds the required time, the cone of the output with the maximum violation of the required time is chosen for compression. Details of the construction of the LPT and the compression algorithm can be found in Chapter 4.

The compression algorithm attempts to make the task of the relaxation phase easier by relocating modules to empty slots, if they are available at the right distance to achieve the required compression. Further, an attempt is made to spread out the infeasible slots to make the relocation in the relaxation phase easier.

## 6.5 Experimental Results

We have implemented our algorithms as a package called *reeng*. For our experiments we used the Xilinx 3000 series FPGA architecture as the underlying architecture on which circuits are placed. Experiments were conducted using technology mapped MCNC combinational benchmark circuits.

The initial delay feasible placement for the circuits was generated using the *sysdias* package [30]. Structural and timing changes were generated randomly. Both the number of structural and timing changes and the circuit modules affected by the changes were controlled probabilistically. Roughly speaking, the expected number of engineering changes of a particular type introduced by our algorithm for generating random engineering changes is equal to the product of the probability of that change and the number of nodes in the circuit. We studied the behavior of our algorithm with the number of changes made to the circuit specification.

One of the crucial requirements for any placement re-engineering algorithm is that it should be significantly faster than re-doing the placement for the modified circuit from scratch. In order to study this aspect of our algorithm, we compare the number of compression-relaxation iterations required to generate a delay feasible placement and the run time for *reeng* with those required by *sysdias* for placing the modified circuit from scratch. Since *sysdias* starts from a randomly generated initial placement and then performs compression-relaxation iterations until a delay feasible placement is obtained, the number of compression-relaxation iterations used by *reeng* and *sysdias* is a good measure of the complexity of placement re-engineering using these two methods. These results are shown in Table 6.1. The probabilities for structural changes and timing changes were both set to 0.1 for the generation of engineering changes. Thus, roughly 10% of the nodes and interconnections were affected by the structural changes and around 10% of the timing constraints were altered. The amount by which a timing constraint was tightened (that is the reduction in required time, $R(x)$) in a timing change was a uniformly distributed

| Circuit | # mod | EC | | *sysdias* | | | *reeng* | | |
|---|---|---|---|---|---|---|---|---|---|
| | | SC | TC | # iter | Time | Delay | #iter | Time | Delay |
| cordic | 48 | 6 | 4 | 25 | 48 | 50.5 | 4 | 3 | 51.0 |
| count | 85 | 7 | 10 | 32 | 85 | 60.0 | 3 | 3 | 58.5 |
| bw | 61 | 9 | 5 | 20 | 80 | 32.0 | 4 | 4 | 30.0 |
| f51m | 58 | 7 | 5 | 21 | 93 | 83.4 | 3 | 3 | 85.0 |
| frg1 | 91 | 12 | 8 | 32 | 150 | 75.5 | 5 | 6 | 75.0 |
| comp | 103 | 10 | 12 | 40 | 260 | 80.4 | 4 | 4 | 82.0 |
| term1 | 161 | 18 | 16 | 45 | 303 | 70.6 | 5 | 6 | 75.0 |
| C499 | 144 | 12 | 10 | 33 | 260 | 73.0 | 4 | 4 | 72.6 |
| C880 | 210 | 18 | 22 | 42 | 335 | 120.5 | 5 | 6 | 125.0 |
| alu2 | 232 | 21 | 22 | 48 | 410 | 183.5 | 7 | 7 | 185.4 |

**Table 6.1**: Comparison of running time (in seconds), delay (after routing) of the critical path (in nanoseconds) and number of compression-relaxation iterations for *reeng* and *sysdias*. In *sysdias* the entire placement is re-done from scratch.

random variable in the range $[0, 0.1R(x)]$. The columns SC and TC refer to the number of structural and timing changes respectively. In SC only module additions/deletions are counted. Table 6.1 shows that the run time for *reeng* is an order of magnitude smaller than the run time for *sysdias* and the number of compression-relaxation iterations is also proportionally smaller. This confirms our hypothesis that the delay feasible placement for the modified circuit is "close" (in the configuration space of all placements) to the delay feasible placement for the original circuit and hence can be found more efficiently by using the delay feasible placement for the original circuit as a starting point. Table 6.1 also shows the actual delay in the critical path for the placed circuits after routing using the Xilinx *apr* tool. The critical path delays using *reeng* and using *sysdias* are approximately the same. This is to be expected since both the packages terminate once all the timing constraints are satisfied.

We also studied the variation in run time of *reeng* with respect to the probability of structural and timing changes. The graph in Figure 6.5 shows the variation in the run time for *reeng* on circuit *alu2*. It is interesting to observe that the increase in running time with probability of structural change is almost monotonic. The non-monotonic sections of the graph may be due to the fact that it is possible that a smaller number of changes may sometimes have a larger affect on the critical paths in the circuit. Also a higher probability of change does not necessarily imply that the actual number of changes introduced by our algorithm for generating random engineering changes is larger. The increase in run time is slightly faster when the probability of timing change is increased. This could be due to the fact that timing changes tend to have a more global effect on the placement since they might force the relocation of several modules. Structural changes on the other hand are more local, and only if there is a violation of some timing constraint due to structural changes, their effects are propagated to other parts of the circuit.

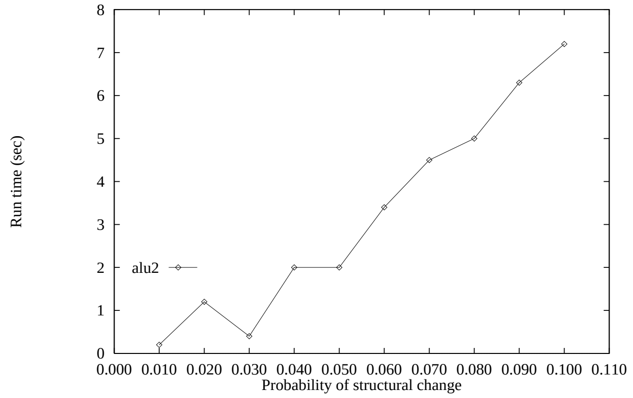


**Figure 6.5**: Variation in run time of *reeng* with increase in structural changes.

Another parameter of interest in evaluating a placement re-engineering algorithm is the difference between the initial placement and the new placement generated after placement re-engineering. We measured the variation in the number of module movements during re-engineering with the probability of change. We observed that there is a strong correlation between the running time of *reeng* and the number of module movements. Figure 6.6 shows the variation in number of module movements with structural change probability for *alu2*.

## 6.6 Conclusions

We have presented a new approach to the timing driven placement re-engineering problem for regular architectures. Our algorithm handles both structural changes in the circuit topology as well as changes in the timing constraints imposed on the placement. A novel algorithm for generating the initial placement for new modules using maximum affinity matching is proposed. Timing driven relocation using the concept of a slack neighborhood graph is used in

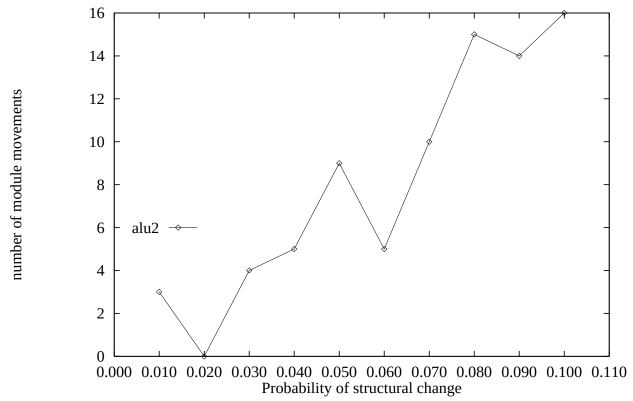


**Figure 6.6**: Variation in number of module movements with structural change probability.

transforming physically infeasible intermediate placements to physically feasible ones with low timing degradation. Our experimental results on the Xilinx 3000 series FPGA architecture demonstrate the viability of our approach.

# Chapter 7

# Conclusions

We have addressed several optimization problems that arise in the design of VLSI circuits, with the satisfaction of timing constraints as the primary objective. Although we focussed on algorithms for regular architectures, many of our algorithms can be extended to architectures that have some irregularities, albeit at the cost of increase in computation time. Field Programmable Gate Arrays (FPGAs), identical standard cell based architectures and WSI arrays of blocks having a regular structure are the major architectures that are regular. The regularity of these architectures allows us to use powerful graph theoretic techniques that would not be possible otherwise. In this chapter we discuss the main contributions of this thesis and mention some areas for future research.

## 7.1 Contributions

Here are some of the major contributions of this thesis:

- **Use of general delay model in FPGA technology mapping:** All the previous work in technology mapping for look-up table based FPGAs had used either the unit delay model or the nominal delay model to estimate interconnect delays. We explored the use of the general delay model in our technology mapping algorithm and demonstrated that it results in better mapping solutions. In particular, we showed that mapping solutions with a larger number of levels may have lower critical path delay after placement and routing, than the solution with the minimum number of levels.

- **Integrating technology mapping and placement:** A "chicken-and-egg" situation arises in the use of the general delay model for technology mapping: the exact interconnect delays are not available until after technology mapping, placement and routing. In order to overcome this problem, we proposed the integration of technology mapping and placement in an iterative loop by feeding back information about exact interconnection delays from one iteration to the subsequent technology mapping phase. This allows our algorithm to incorporate exact delay information during technology mapping. This approach has subsequently been adopted in [49].

- **Slack neighborhood graph:** We introduced the concept of a slack neighborhood graph to capture neighborhoods within which circuit modules can move without increasing the critical path delay of the circuit. The regularity of the underlying architecture allows us to compute the slack neighborhood graph efficiently. We introduce the problem of timing driven relocation to reconfigure a placement to avoid infeasible slots, without significant timing degradation. Minimum cost network flow in a flow network derived from slack neighborhood graphs is used to find relocation paths. Relocation of circuit modules along these paths is shown to result in reconfiguration with provably low timing degradation. Timing driven relocation provides a powerful timing preserving transformation on a placement in a regular architecture and we use it to provide a unified framework for solving the problems of timing driven placement, reconfiguration for fault tolerance, and design re-engineering.

- **Compression-relaxation based placement:** We presented a new iterative algorithm for timing driven placement for FPGAs. This algorithm uses physically infeasible placements as "stepping stones" to reach a delay feasible placement. The compression phase uses the longest path tree of a delay infeasible cone to compress several paths simultaneously, and in the process may cause modules to overlap. The relaxation phase uses our algorithm for timing driven relocation to obtain a physically feasible placement without undoing the reduction in critical path delay achieved during the compression phase.

- **Timing driven reconfiguration:** Traditionally, reconfiguration for fault tolerance and yield enhancement has been studied in the context of processor arrays. In such a scenario, the objective is to extract the largest fault free subset of processors with a certain topology (such as a mesh). To the best of our knowledge, our work is the first to address the issue of reconfiguring circuit placement with minimization of timing degradation and reprogramming cost as the main objective.

- **Design re-engineering:** We have studied the problem of handling small structural and timing changes in a circuit without having to re-do the placement from scratch. A novel algorithm for generating the initial placement for new modules using maximum affinity matching is proposed. Timing driven relocation using the concept of a slack neighborhood graph is used in transforming physically infeasible intermediate placements to physically feasible ones with low timing degradation.

## 7.2 Directions for Future Research

One important problem that needs to be addressed for developing better algorithms for timing driven optimization problems is that of *delay estimation* in the initial phases of design flow. This requires the development of accurate models that capture not only the delay characteristics of the architecture, but also the effects of the placement and routing algorithms. Of course, any such model will be specific to a particular architecture and a class of layout algorithms. However, it would be nice if a general methodology can be developed for generating such models.

The problem of technology mapping for sequential circuits was addressed in our research by simply technology mapping the combinational circuits and "padding in" the flip-flops into the CLBs. Recent work [38] has shown that in the unit delay model, cycle time can be reduced by combining retiming and technology mapping. Extensions of these results to more general delay models would be interesting and useful. Another interesting problem is to explore other techniques for integrating technology mapping and layout.

It would be interesting to explore the extension of the concept of slack neighborhood graph to architectures with less symmetry. It is clear that the time required for the generation of the slack neighborhood graph will be more for such architectures, but it might yield substantial improvements in the algorithmic techniques for solving placement-related problems (such as timing driven placement, reconfiguration and design re-engineering). The extensions to standard cell and gate array architectures are worth exploring because these architectures are "almost regular".

# Bibliography


[1] *Actel FPGA Data Book and Design Guide,* Actel, 1994.

[2] K. P. Belkhale, P. Banerjee, Reconfiguration Strategies in VLSI Processor Arrays, *Proc. ICCD, Oct. 1988, Rye Brook, NY, pp. 418–421.*

[3] M. Breuer, Min-cut Placement, *J. Design Automation and Fault Tolerance, Vol. 1, No. 4, Oct. 1977, pp. 343–362.*

[4] M. Burnstein, M. N. Youseef, Timing-Influenced Layout Design, *Proc. 22nd DAC, 1985, pp. 124–130.*

[5] S. C. Chang, K. T. Cheng, N. S. Woo, M. Marek-Sadowska, Layout Driven Logic Synthesis, *Proc. DAC, 1994, pp. 308–313.*

[6] C. Choy, T. Cheung, An Algorithm to Deal with Incremental Layout Alteration, *Proc. 34th Midwest Symposium on Circuits and Systems, Vol. 2, 1991, pp. 850–853.*

[7] J. Cong, A. Kahng, P. Trajmar, K. C. Chen, Graph Based FPGA Technology Mapping for Delay Optimization, *ACM International Workshop on Field Programmable Gate Arrays, 1992, pp. 77–82.*

[8] J. Cong, Y. Ding, An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs, *Proc. ICCAD, (1992), pp. 48–53.*

[9] A. E. Dunlop, V. D. Agrawal, D. N. Deutsch, M. F. Jukl, P. Kozak, M. Weisel, Chip Layout Optimization Using Critical Path Weighting, *Proc. 21st DAC, 1984, pp. 133–136.*

[10] W. E. Donath, R. J. Norman, B. K. Agrawal, S. E. Bello, S. Y. Han, J. M. Kertzberg, P. Lowry, R. I. McMilan, Timing Driven Placement Using Complete Path Delays, *Proc. 27th DAC, 1990, pp. 84–89.*

[11] W. C. Elmore, The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers, *J. Appl. Phy., Vol. 19, January 1948, pp. 55–63.*

[12] R. Francis, J. Rose, Z. Vranesic, Chortle-crf : Fast Technology Mapping for Lookup Table-Based FPGAs, *Proc. 28th DAC, 1991, pp. 227–233.*

[13] R. J. Francis, J. Rose, Z. Vranesic, Technology Mapping for Delay Optimization of Lookup Table-Based FPGAs, *Proc. ICCAD, 1991, pp. 568–571.*

[14] T. Gao, K.-C. Chen, J. Cong, Placement and Placement Driven Technology Mapping Algorithms for Xilinx FPGAs, *IEEE ASIC Conference, 1993.*

[15] T. Gao, P. M. Vaidya, C. L. Liu, A New Performance Driven Placement Algorithm, *Proc. ICCAD, 1991, pp. 44–47.*

[16] T. Gao, P. M. Vaidya, C. L. Liu, A Performance Driven Macro-Cell Placement Algorithm, *Proc. 29th DAC, 1992, pp. 147–152.*

[17] A. V. Goldberg, An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm, *Technical Report STAN-CS-92-1439, Computer Science Department, Stanford University, Stanford, CA, 1992.*

[18] S. Goto, An Efficient Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Layout, *IEEE Trans. Circuits Syst., Vol. CAS-28, Jan. 1981, pp. 12–18.*

[19] N. J. Howard, A. M. Tyrrell, N. M. Allinson, The Yield Enhancement of Field-Programmable Gate Arrays, *IEEE Trans. on VLSI Systems, Vol. 2, 04 1994, pp. 115–123.*

[20] M. A. B. Jackson, E. S. Kuh, M. Marek-Sadowska, Timing-Driven Routing for Building Block Layout, *Proc. ISCAS, 1987, pp. 518–519.*

[21] M. A. B. Jackson, E. S. Kuh, Performance-Driven Placement of Cell Based ICs, *Proc. 26th DAC, 1989, pp. 370–375.*

[22] Y. JU, Incremental Circuit Simulation and Timing Analysis Techniques, *PhD. Thesis, Univ. of Illinois at Urbana-Champaign, 1993.*

[23] S. KIRKPATRICK, C. D. GELATT, JR., M. P. VECCHI, Optimization by Simulated Annealing, *Science, 13 May 1983, Vol. 220, No. 4598.*

[24] S. Y. KUO, W. K. FUCHS, Efficient Spare Allocation for Reconfigurable Arrays, *IEEE Design and Test, Feb. 1987, pp. 24–31.*

[25] H. T. KUNG, M. S. LAM, Fault-tolerance and Two Level Pipelining in VLSI Systolic Arrays, *Proc. of M.I.T. Conf. on Advanced Research in VLSI, 1984, pp. 74–83.*

[26] U. LAUTHER, A Min-cut Placement Algorithm for General Cell Assemblies Based on Graph Representation, *Proc. 16th DAC, 1979, pp. 1–10.*

[27] E. L. LAWLER, K. N. LEVITT, J. TURNER, Module Clustering to Minimize Delay in Digital Networks, *Proc. IEEE Trans. on Computers, C-18(1) (1969), pp. 47–57.*

[28] M. MAREK-SADOWSKA, S. P. LIN, Timing-Driven Placement, *Proc. ICCAD, 1989, pp. 94–97.*

[29] A. MATHUR, C. L. LIU, Performance Driven Technology Mapping for Lookup-Table Based FPGAs using the General Delay Model, *2nd Intl. ACM/SIGDA Workshop on FPGAs, Berkeley, 1994.*

[30] A. MATHUR, C. L. LIU, Compression-Relaxation: A New Approach to Performance Driven Placement for Regular Architectures, *Proc. ICCAD, 1994, pp. 130–136.*

[31] A. MATHUR, K. C. CHEN, C. L. LIU, Applications of Slack Neighborhood Graphs to Timing Driven Optimization Problems in FPGAs, *Proc. 3rd ACM/SIGDA Symposium on FPGAs, 1995, pp. 118–124.*

[32] A. MATHUR, C. L. LIU, Timing Driven Reconfiguration for Fault Tolerance and Yield Enhancement in FPGAs, *submitted for publication, 1994.*

[33] A. MATHUR, K. C. CHEN, C. L. LIU, Re-engineering of Timing Constrained Placements for Regular Architectures, *to appear in Proc. ICCAD, 1995.*

[34] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, A. Sangiovanni-Vincentelli, Logic Synthesis for Programmable Gate Arrays, *Proc. DAC, 1990, pp. 620–625.*

[35] R. Murgai, N. Shenoy, R. K. Brayton, A. Sangiovanni-Vincentelli, Performance Directed Synthesis for Table Look Up Programmable Gate Arrays, *Proc. ICCAD, (1991), pp. 572–575.*

[36] R. Nair, C. L. Berman, P. S. Hauge, E. J. Yoffa, Generation of Performance Constraints for Layout, *IEEE Trans. on Computer-Aided Design, Vol. 8, Aug. 1989, pp. 860–874.*

[37] J. Narasimhan, K. Nakajima, C. S. Rim, A. T. Dahbura, Yield Enhancement of Programmable ASIC Arrays by Reconfiguration of Circuit Placements, *IEEE Trans. on Computer-Aided Design, Vol. 13, Aug. 1994, pp. 976–986.*

[38] P. Pan, C. L. Liu, Optimal Technology Mapping of Loopless Sequential Circuits for LUT-based FPGAs, *manuscript, 1995.*

[39] G. Persky, PRO - An Automatic String Placement Program for Polycell Layout, *Proc. 13th DAC, 1976, pp. 417–424.*

[40] J. Rubinstein, P. Penfield, M. A. Horowitz, Signal delay in RC Tree Networks, *IEEE Trans. on Computer-Aided Design, July 1983, pp. 119–127.*

[41] A. L. Rosenberg, The Diogenes Approach to Testable Fault-tolerant Arrays of Processors, *IEEE Transactions on Computers, 32 (1983), pp. 902–910.*

[42] M. Sami, R. Stefanelli, Reconfigurable Architectures for VLSI Processing Arrays, *Proceedings of the IEEE, 74 (1986), pp. 712–722.*

[43] P. Sawkar, D. Thomas, Area and Delay Mapping for Table-Look-Up Based Field Programmable Gate Arrays, *Proc. 29th DAC, (1992), pp. 368–373.*

[44] A. Srinivasan, K. Chaudhary, E. S. Kuh, RITUAL : A Performance Driven Placement Algorithm for Small Cell ICs, *Proc. ICCAD, 1991, pp. 48–51.*

[45] B. Codenotti, R. Tamassia, A Network Flow Approach to the Reconfiguration of VLSI Arrays, *IEEE Transactions on Computers, 40 (1991), pp. 118–121.*

[46] R. E. Tarjan, Data Structures and Network Algorithms, Chap. 8, *SIAM, 1983.*

[47] S. Teig, R. L. Smith, J. Seaton, Timing-Driven Layout of Cell-Based ICs, *VLSI Systems Design, May 1986, pp. 63–73.*

[48] M. Terai, K. Takahashi, K. Sato, A New Min Cut Placement Algorithm for Timing Assurance Layout Design Meeting Net Length Constraints, *Proc. 27th DAC, 1990, pp. 96–102.*

[49] H. Yang, D. F. Wong, *Edge-Map: Optimal Performance Driven Technology Mapping for Iterative LUT Based FPGA Designs, Proc. ICCAD, 1994, pp. 150–155.*

[50] Xilinx, *The XC4000 data book*, Xilinx Inc., 1991.

# Vita

Anmol Mathur was born on 21st July, 1968 in Meerut, India. He received his schooling in St. Xavier School, Jaipur. He entered the Indian Institute of Technology, Delhi in 1986, having secured the 2nd rank in the Joint Entrance Examination for the IITs. He received the B. Tech. in Computer Science and Engineering from IIT, Delhi in 1990 and was awarded a gold medal by the institute for securing the highest GPA amongst all the graduating students.

Anmol joined the Computer Science department at the University of Illinois at Urbana-Champaign in Fall 1990 on a University of Illinois Fellowship and received his M.S. in 1993 and Ph.D. in 1995. His research interests include algorithms for VLSI CAD, computational geometry, combinatorics and parallel computer architectures. He joined the VLSI CAD group at Silicon Graphics Inc., Mountain View, CA as a member of technical staff after completing his doctorate.