LOGIC AND LAYOUT OPTIMIZATION FOR SEQUENTIAL CIRCUITS

BY

PEICHEN PAN

B.Engr., Xidian University, 1984
M.S., Peking University, 1987

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

LOGIC AND LAYOUT OPTIMIZATION FOR SEQUENTIAL CIRCUITS

Peichen Pan, Ph.D
Department of Computer Science
University of Illinois at Urbana-Champaign, 1995
Professor C. L. Liu, Advisor

Computer-aided design of VLSI circuits is usually carried out in three synthesis steps: high-level synthesis, logic synthesis and layout synthesis. Each synthesis step is further broken into a few optimization problems. In this thesis we study several such problems in logic and layout synthesis.

We study how the technique of retiming can be introduced to enhance the solution of three problems in logic synthesis. Specifically, we consider a partial scan approach to the problem of design-for-testability in which a set of scan signals (instead of scan flip-flops) is pre-selected. We propose an algorithm that uses retiming to position flip-flops on the pre-selected scan signals so that these signals can be scanned. Next, we combine resynthesis with retiming to reduce the cycle time of a sequential circuit. We also integrate retiming into the technology mapping step for look-up table based field programmable gate arrays (FPGAs). We present an optimal cycle time technology mapping algorithm for sequential circuits.

In layout synthesis, we study the area minimization problem in floorplanning (also known as the floorplan sizing problem). We propose two area minimization algorithms for general floorplans. Both algorithms can be viewed as generalizations of the classical area minimization algorithm by Otten and Stockmeyer. A fast pseudo-polynomial area minimization algorithm for an important class of hierarchical floorplans is also proposed. We settle an open problem on the complexity of the area minimization problem for hierarchical floorplans by showing it to be NP-complete. Finally, we study a graph constraint reduction problem in symbolic layout compaction. We present a polynomial algorithm that achieves optimal reduction.

To
my grandparents Aichun & Kaixiang,
my parents Jifang & Jiazhen,
and
my wife Ping

Acknowledgments

First and foremost, I would like to thank my advisor, Professor C.L. Liu for his guidance, support, and patience. He was always there to listen to my problems, whether they were research-related or personal, and to provide valuable suggestions, comments, and ideas. Without the help of his insights, suggestions, and critiques this work would have never matured into a thesis. During the course of my Ph.D. studies, I developed a deep respect and admiration for him as a researcher as well as a person. I consider myself privileged to have worked under such a great advisor. For all these and much much more he has my utmost gratitude.

I am grateful to the other members of my thesis committee, Professor Sung-Mo Kang and Professor Rajesh Gupta, for taking time out of their busy schedules to serve on my thesis committee, and for their encouragement and interest. Their suggestions on an early draft of the thesis greatly improved the presentation.

I would like to thank Professors Prithviraj Banerjee, Herbert Edelsbrunner, Janak Patel, and Douglas West for the challenging and interesting courses I took from them.

I would also like to thank Joyce Woodworth for her superb administrative work.

Some of the results reported here are based on papers co-authored with Sai-keung Dong, C.L. Liu, and Weiping Shi. I would like to thank them for their enormous contributions.

Our research group has been a great source of wisdom and knowledge for me. It has been a tremendous learning experience to be in this group. I would like to thank my present and former colleagues: Cheng Chang, Xiangfeng Chen, Ki-seok Chung, Sai-keung Dong, Tong Gao, Ran Libeskind-Hadas, Taewhan Kim, Anmol Mathur, Unni Narayanan, Chaeryung Park, Prashant Saxena, Yachyang Sun, and Too-Seng Tia, for all the technical and non-technical discussions which greatly helped my studies and life here at Illinois.

The love, support and encouragement of my family and my wife's family made my pursuit of high academic endeavor possible. I feel very fortunate and grateful to them.

My special thanks go to my wife, Ping Jiang. Her love, encouragement, and faith in my abilities kept me striving to achieve more than what I thought I could. Her unconditional support and endless sacrifices made this thesis possible. Every bit of this thesis has her contributions. Finally, I thank my son, Derek and daughter, Karen for the happiness and joy they bring into my life.

Table of Contents

Chapter

List of Tables

List of Figures

Chapter 1

Introduction

As the complexity of VLSI circuits increases, automation of the design process becomes increasingly important. The advantages of computer-aided design include

- reducing the design time,
- reducing the design cost,
- improving the design quality.

Since the design process for VLSI circuits is very complex, it is often broken into several subtasks which are usually performed in a sequential manner. The design process can be viewed as a sequence of transformations, starting from a behavioral description of a design, to a register-transfer description, to a logic description (circuit), and finally to a layout. A layout contains the detailed geometrical information for the fabrication of a chip. Figure 1.1 shows a generic flow of this process. The main subtasks are high-level synthesis, logic synthesis and layout synthesis. It should also be noted that most current CAD systems cannot complete a design in one pass as shown in Figure 1.1. A number of iterations may be necessary in order to produce a high quality final design.

In fact, each synthesis step in Figure 1.1 is still too complex to be solved as a single problem, and is again broken into several optimization problems. The optimization problems are then solved separately to obtain an overall solution. For example, typical problems in layout synthesis are floorplanning/placement, global and detailed routing, and compaction.

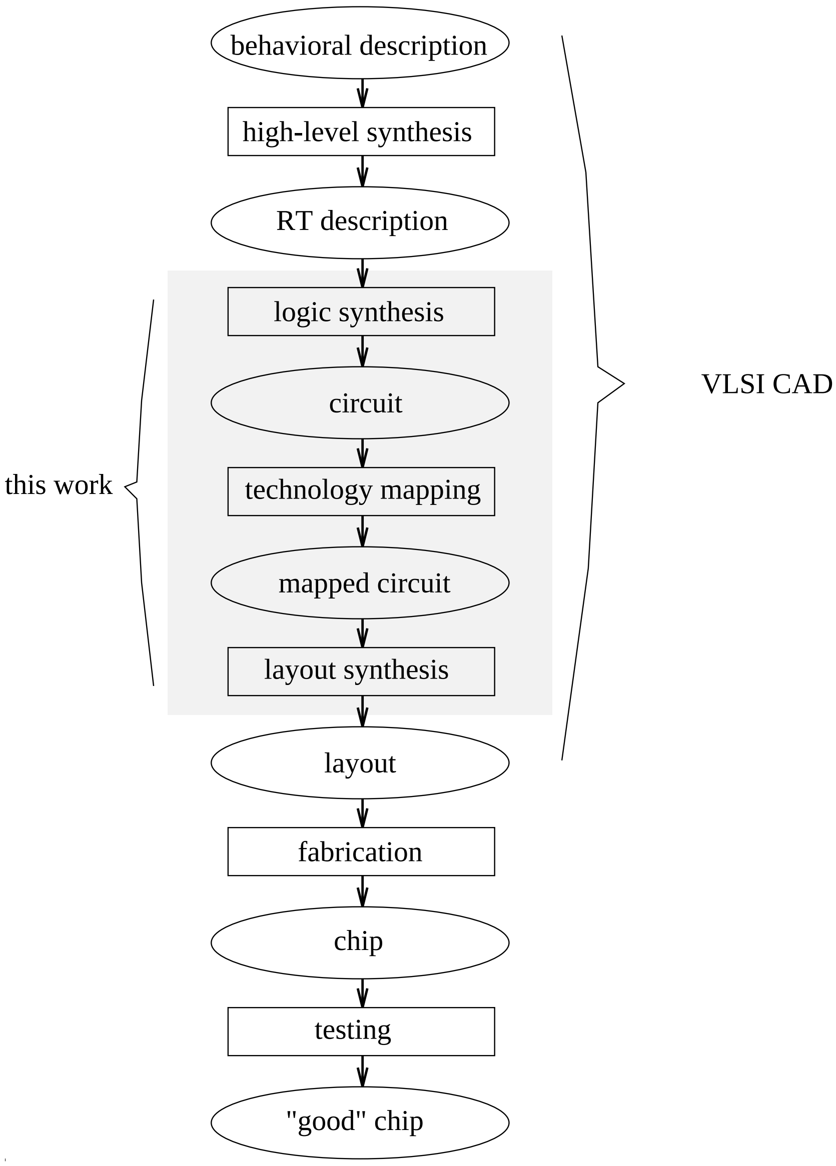


Figure 1.1: The generic flow of VLSI design.

Testability is another important issue in the design of VLSI circuits. A chip fabricated in the fabrication line might deviate from the desired functionality due to design and fabrication errors, or fabrication defects. The goal of testing is to try to detect errors and defects by applying a sequence of test patterns to the chip and observing its responses. Though testing itself is the last step in the VLSI design process, testing considerations are usually included in all synthesis steps in order to enhance the testability of the chip.

1.1 Logic synthesis

Logic synthesis is the process of transforming a register-transfer description of a design to a logic circuit. Logic synthesis is typically carried out by a sequence of optimizations.

Most circuits we encounter in practice are sequential ones, namely, there are flip-flops (FFs) in the circuits. For sequential circuits, there is a functionality-preserving transformation known as retiming [45]. Retiming is a process of repositioning the FFs without changing the logic or the functionality of the circuit. The original purpose of introducing the retiming technique is to minimize the cycle time, the number of FFs, or the combination of both. Efficient algorithms for these purposes have been proposed in the literature.

Because of its inability of modifying the logic of a circuit, retiming is rarely integrated into the solution of other optimization problems. It is traditionally used only as the pre- and/or post-processing step of other optimization steps. Many optimization problems for sequential circuits are still posed under the assumption that the FF positions are fixed and in many cases the optimization is carried out only on the combinational logic between the FFs. Obviously, the assumption that the FFs have fixed positions greatly restricts the solution space that can be explored for an optimization problem on sequential circuits. Therefore, to find the best solution to an optimization problem, retiming need to be integrated into the solution process of the problem.

In this thesis, We first study how the technique of retiming can be introduced to enhance the solution of three problems in logic synthesis: partial scan for testability enhancement, cycle time reduction, and technology mapping for look-up table based FPGAs.

Sequential circuits are difficult to test mainly because the state signals in the feedback loops are difficult to control and observe. Partial scan has been proposed as a design-for-testability technique which attempts to make some of the signals controllable and observable. This is achieved by scanning the FFs driven by the signals. The cost of partial scan is directly related to the number of scan signals, which is equal to the number of scan FFs. Traditional partial scan approaches assume that the positions of the FFs in a circuit are fixed. Therefore, scan signals cannot be selected arbitrarily — they must be signals that drive FFs. We study an approach to partial scan in which scan signals are selected without considering whether they drive FFs. This allows maximum freedom for the selection of scan signals. Of course, to actually scan these pre-selected signals, eventually, each of them must be made to drive a FF. We propose in this thesis an algorithm that uses retiming to position FFs on a set of pre-selected signals so that these signals can be scanned. Our experimental results show significant reduction in the number of scan FFs with very little overhead.

Timing optimization is an important issue in logic synthesis. The speed of a sequential circuit is determined by its cycle time which is the maximum delay on the combinational paths in the circuit. Though retiming can be used to reduce cycle time by repositioning the FFs, its effectiveness is very limited because retiming does not modify the logic of a circuit. In general, resynthesizing the logic will produce much better results. In the conventional approach, combinational logic between the FFs are resynthesized so that combinational path delays between FFs are all upper-bounded by the target cycle time. This approach lacks a global view in two ways: First, signal dependencies across FF boundaries are not exploited. Second, the requirement that all combinational path delays be upper-bounded by the cycle time is unnecessarily stringent since a circuit can be retimed to achieve a given cycle time even though some of its combinational path delays exceed it. We present a method which combines resynthesis with retiming to reduce the cycle time of a sequential circuit. Experimental results show that the cycle time produced by our algorithm is better than that by algorithms which carry the retiming and resynthesis steps separately.

Field programmable gate arrays (FPGAs) has evolved rapidly as an important ASIC technology. A look-up table (LUT) based FPGA consists of an array of programmable logic blocks (PLBs) together with programmable interconnections. The core of a PLB is a k-input LUT (k-LUT) which can implement any combinational logic with up to k inputs and a single output, where k is a positive integer ranging usually from 3 to 9. The technology mapping problem for LUT-based FPGAs is to produce, for a given circuit, an equivalent circuit comprising of

k-LUTs. The conventional approach for solving this problem is again based on the assumption that the positions of the FFs in a circuit are fixed. Specifically, the FFs in a circuit are removed to obtain a combinational network. Then the combinational network is mapped using a mapping algorithm for combinational circuits. Finally, the FFs are connected back (to the mapping solution of the combinational network) to generate a mapping solution for the sequential circuit. We propose in this paper a new approach to the technology mapping problem for sequential circuits. In our approach, retiming is fully integrated into the technology mapping process. We present an efficient technology mapping algorithm that can produce a mapping solution with minimum cycle time for circuits without loops under the unit delay model. The algorithm is also extended to the general delay model, in which a mapping solution with provably good cycle time will be produced. The algorithm can also be used for circuits with loops by properly removing some of the FFs to break the loops first. Experimental results show that our algorithm consistently produces better mapping solutions than the ones based on the conventional approach.

1.2 Layout synthesis

After a circuit has been optimized and mapped, layout synthesis follows. As was shown in Figure 1.1, the objective of layout synthesis is to produce a layout from a mapped circuit. Compared to other synthesis steps, time spent for layout synthesis still dominates. As a result, there is a constant need to design faster algorithms that can produce better results for various optimization problems in layout synthesis. In this thesis, we study two layout problems: area minimization in floorplanning and constraint reduction in compaction.

Floorplanning is an important stage in layout synthesis. Because of its complexity, floorplanning is usually carried out in two separate steps. The first step is to determine the floorplan that specifies the relative positions of the subcircuits on a chip. Since each subcircuit may have several possible layout alternatives with different dimensions, the second step is to select a layout alternative for each subcircuit so that the total chip area is minimized while the floorplan determined in the first step is held fixed. The second step is referred to as the area minimization problem.

Previous area minimization algorithms for general floorplans are based on either mathematical programming or the branch-and-bound technique. We propose two new algorithms. Our algorithms can be viewed as generalizations of the classical algorithm for slicing floorplans due to Otten and Stockmeyer in the sense that they reduce naturally to their algorithm for slicing floorplans. Our algorithms are provably better than an exhaustive method for all the examples we examined. The complexity of the area minimization problem for hierarchical floorplans was an open problem. We settle this open problem by showing it to be **NP**-complete. We also present a fast pseudo-polynomial area minimization algorithm for an important class of hierarchical floorplans.

Compaction is the process of producing an area-efficient physical layout from a symbolic layout while enforcing the separation (design rules), electrical connectivity, and user-specified requirements. Both physical separation and electrical connectivity requirements which constitute most of the requirements can be expressed as graph constraints, linear inequalities of the form $x_i - x_j \geq c$, where x_i and x_j are variables and c is a constant.

Modern designs are usually represented hierarchically due to their complexity. Compaction of a hierarchical design is usually carries out in two phases. In the first phase, in a bottom-up pass through a design's hierarchy, a port abstraction is generated for each subcell. A port abstraction is a simplified definition of a subcell; it is simply a system of graph constraints among the x- or y-coordinates of some of the layout components in the subcell. A port abstraction will be passed to the next level of the hierarchy for the generation of the port abstraction for its parent cell. After the root cell is reached, the constraint system formed by the coordinates of all the layout components in each subcell is solved in a top-down fashion. To reduce the execution time and memory usage of the compaction process, it is desirable to reduce port abstractions first, then pass them to the next level. Therefore, we study the graph constraint reduction problem, i.e., the problem of producing, for a given system of graph constraints, an equivalent system with the fewest constraints. Previously, the problem was formulated as that of removing a maximum number of "redundant" constraints. We found this formulation not only render the problem to be NP-hard, but also can not guarantee that the solutions are optimal. We then present a new formulation of the problem. In our formulation, constraints can be removed as well as added. Our formulation always achieves an optimal reduction. We further present a polynomial algorithm for the new formulation.

1.3 Organization of thesis

The remainder of the thesis is organized as follows: In the first part, we study three problems in logic synthesis. In Chapter 2, we study partial scan with retiming. In Chapter 3, we study retiming and resynthesis for cycle-time reduction. We present a procedure for generating resynthesis timing constraints for the combinational logic of a sequential circuit. In chapter 4, we study the technology mapping problem for sequential circuits for look-up table based FPGAs. We present our new approach to the problem. In our approach, retiming is fully integrated into the technology mapping process.

In the second part, we study two layout problems. In Chapters 5 and 6, we study the area minimization problem in floorplanning (also known as the floorplan sizing problem). In Chapter 7, we study the problem of graph constraint reduction in compaction. We present our new formulation of the problem which always achieves the optimal reduction.

Finally, Chapter 8 concludes this thesis with some proposed future work.

Chapter 2

Partial Scan with Pre-selected Scan Signals

Sequential circuits are difficult to test mainly because state signals in the feedback loops are difficult to control and observe. Scan design has been proposed as a design-for-testability technique for sequential circuits [1]. Due to the high area and performance penalty associated with full scan, partial scan is often used in practice.

The main problem in partial scan design is to select a set of scan signals. Several methods have been proposed for this purpose [16, 17, 40, 69]. Among them, cycle-breaking has been demonstrated to be a very effective one [16, 17]. The goal of cycle-breaking is to select a small set of signals such that their removal will simplify the cycle structure of a circuit, or will break the cycles that cause many of the hard-to-detect faults. In conventional approaches, the positions of the FFs in a circuit are assumed to be fixed. Therefore, scan signals cannot be selected arbitrarily — they must be signals that drive FFs. In this case, the cycle structure of a circuit can be captured by a graph on the FFs called *S-graph*. Breaking up the cycles in the circuit amounts to breaking up the cycles in the corresponding S-graph. Effective heuristics as well as exact algorithms have been proposed for selecting a minimum set of FFs to break up the cycles in an S-graph [3, 11, 43].

Retiming is a process of repositioning the FFs without changing the functionality or the structure of a circuit [45]. Retiming has been applied to partial scan. Kagaris and Tragoudas [38] proposed a method which uses a legal retiming to reposition the FFs. Then, the usual

S-graph approach is adopted to select scan signals in the retimed circuit. It was demonstrated that the number of scan FFs can be reduced this way. However, the legal retiming requirement still restricts the selection of scan signals. Moreover, it is not clear what FF configuration can lead to the best reduction in scan FFs. A configuration with the minimum number of FFs is suggested in [38], the reduction in the number of scan FFs is, however, moderate.

A recent approach to partial scan selects a best possible set of cycle breaking signals (under certain criteria) without considering the availability of the FFs. This allows maximum freedom for the selection of scan signals, after all the cycle structure of a circuit is independent of the positions of the FFs in the circuit. Of course, to actually scan these pre-selected signals, eventually, each of them must be made to drive a FF. Legal retiming can be used for this purpose. In general, however, legal retiming alone is not enough. Recently, Chakradhar and Dey [12] showed that through a combination of retiming and logic replication, desired FF configuration in which each pre-selected signal drives a FF can be obtained. They further proposed an iterative algorithm for doing so. Their algorithm, however, inherently has no control over the area overhead due to logic replication. Actually, it might introduce unnecessary replication. For example, because of the iterative nature of the algorithm some logic might be replicated several times.

We will present a new approach to the problem of determining a FF configuration in which each pre-selected signals drive a FF. In our approach, the problem is transformed to that of determining a set of FF movements in the circuit, which, in turn, is reduced to a problem of breaking the cycles in a special graph introduced here. The FF movements can be carried out with possible logic replication to generate an equivalent circuit. It is guaranteed that the modified circuit has the *same* cycle structure as the original circuit and the pre-selected signals break the *same* set of cycles in both the original and the modified circuits. Since signals are made to drive FFs simultaneously in our approach, it is guaranteed that no logic is replicated more than once. An algorithm based on the new approach is proposed. The algorithm explicitly minimizes the combinational (logic gates) and sequential (FFs) area overheads incurred due to logic replication.

2.1 Preliminaries and problem definition

A (synchronous) sequential circuit consists of combinational logic separated by FFs. FFs are assumed to be edge-triggered D-type flip-flops and are synchronized by a global clock. For our purpose, a sequential circuit is modeled as an edge weighted directed (multi-)graph. The vertices are the primary inputs (PIs), the primary outputs (POs), and the combinational logic elements. The edges are the interconnections. There is an edge from u (*head*) to v (*tail*) with weight t if the output of u, after passing through t FFs, is an input to v. We will use $w(e)$ to denote the weight of edge e. An edge corresponds to a signal — the signal passing through the corresponding interconnections in the circuit.

Retiming is a process of repositioning the FFs without changing the functionality or the structure of the circuit, first studied by Leiserson *et al.* [45]. A retiming r can be represented by a mapping from the nodes to the integers such that $r(v) = 0$ whenever v is a PI or PO. $r(v) = i$ means that i FFs are removed from each outgoing edge of v and i FFs are added to each incoming edge of v. The physical meaning of $r(v) = i$ is that the computation at v is delayed by i clock cycles, and in the mean time any usage of the output of v is advanced by i clock cycles. Since $r(v) = 0$ if v is a PI or PO, the I/O characteristics of the circuit is preserved after retiming.

In the circuit resulted from applying retiming r, the weight of an edge $e = (u, v)$, denoted $w_r(e)$, is then $w(e) + r(v) - r(u)$. In general, it is possible that $w_r(e) < 0$. In that case, we can imagine that there are $w_r(e)$ *negative* FFs on e. A FF delays a signal by one clock cycle. Analogously, a negative FF advances a signal by one clock cycle. Though there is no physical realization of a negative FF, introduction of such a concept enables us to explore more retiming possibilities. (Note that the negative FFs created by retiming can always be cancelled out by positive FFs eventually.)

In practice, the FFs on the outgoing edges of a vertex can be shared. As a result, the number of FFs on the outgoing edges of a vertex is the maximum of the numbers of FFs on the edges instead of their sum. Similarly, if we want to scan several outgoing edges of a vertex, only one scan FF is needed. In our graph model of a circuit, we do not explicitly model FF sharing because retiming can change the FF count on an edge. However, in counting the FFs as well as the scan FFs, we assume that sharing is employed implicitly.

Unlike retiming which does not modify the structure of a circuit, *replication*, on the other hand, is a process that structurally modifies a circuit while preserving its functionality. Informally, replicating a node means to create several copies of the node and distribute the fanouts of the node among the copies. Here we give the formal definition for duplicating a node. Suppose $Eout(v)$ is partitioned into two groups, E_1 and E_2. We create two copies of v, v_1 and v_2. Each incoming edge of v is duplicated as an incoming edge of v_1 and an incoming edge of v_2. An edge in E_1 will become an outgoing edge of v_1 and an edge in E_2 will become an outgoing edge of E_2. Figure 2.1 illustrates the process.

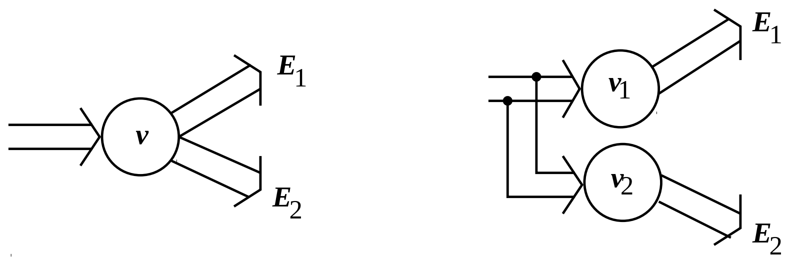


Figure 2.1: Duplicating v.

To *satisfy* an edge, we mean to position a FF on the edge. The problem addressed in this chapter can be formally defined as follows: Given a circuit and a set of edges, $\mathcal{S}$, in the circuit, structurally modify the circuit using replicating and retiming such that the edges in $\mathcal{S}$ are satisfied while the cycles broken by the edges in $\mathcal{S}$ are preserved even after retiming and replication. We call such a modified circuit an *$\mathcal{S}$-configuration* of the original circuit. The edges in $\mathcal{S}$ will be referred to as the *$\mathcal{S}$-edges*. Our primary objective is to find an $\mathcal{S}$-configuration with a small number of replicated vertices. A secondary objective is to reduce the increase in the number of FFs in the $\mathcal{S}$-configuration.

We first consider using retiming alone to satisfy all the $\mathcal{S}$-edges. This is not always possible in general. A necessary and sufficient condition for retiming to achieve this was presented in [12]. However, the first part of the condition was incorrect. The following is the correct one.

Theorem 2.1 *There is a retiming to satisfy all the $\mathcal{S}$-edges iff (1) For each path from a PI to a PO, the number of $\mathcal{S}$-edges does not exceed the number of FFs, and (2) For each cycle, the number of $\mathcal{S}$-edges does not exceed the number of FFs.*

Proof: ($\Longrightarrow$) Since retiming changes neither the number of FFs on a path from a PI to a PO nor the number of FFs on a cycle, obviously both (1) and (2) must be true.

($\Longleftarrow$) We determine a specific retiming to obtain an $\mathcal{S}$-configuration. Let G be a copy of the graph of the circuit (not including the edge weights). We assign weights to edges in G as follows: If e is an $\mathcal{S}$-edge, its weight is $w(e) - 1$; otherwise, its weight is $w(e)$. (Note that $w(e)$ denotes the weight of edge e in the circuit.) We add an extra vertex v_0 to G and connect v_0 to and from all the POs and PIs. All the added edges have weight zero. Because of (1) and (2) it can be easily verified that G contains no negative cycle. Let $r(v)$ be the weight of the shortest paths from v to vertex v_0. We now show that r is a legal retiming such that the retimed circuit is an $\mathcal{S}$-configuration. If v is PI or PO, obviously, $r(v) = 0$ since both (v_0, v) and (v, v_0) have weight zero. Let $e = (u, v)$ be an $\mathcal{S}$-edge. Then, by the triangle inequality $r(u) \le r(v) + w(e) - 1$. That is, $w_r(e) = w(e) + r(v) - r(u) \ge 1$. Namely, there is at least one FF on e in the retimed circuit. If $e = (u, v)$ is not an $\mathcal{S}$-edge, we then have $r(u) \le r(v) + w(e)$, i.e., $w_r(e) = w(e) + r(v) - r(u) \ge 0$. Therefore, r is a legal retiming of the circuit such that the retimed circuit is an $\mathcal{S}$-configuration. □

From Theorem 2.1 it is obvious that an $\mathcal{S}$-configuration cannot be obtained through retiming alone when some cycles and/or paths (from PIs to POs) contains too many $\mathcal{S}$-edges. However, as far as cycle-breaking is concerned, each cycle needs to contain only one satisfied $\mathcal{S}$-edge and a path from a PI to a PO does not need to contain any one at all.

We now use an example to illustrate how replication can be used to obtain an $\mathcal{S}$-configuration for a circuit violating the conditions in Theorem 2.1. Consider the circuit in Figure 2.2(a) with s being the only $\mathcal{S}$-edge. Here, s breaks two cycles: one containing f_1 and the other containing f_2. Retiming cannot position a FF on s. The reason is that s is also on paths $g_1g_2g_3$ and $i_2g_2g_3$, and neither of them contains a FF. Noticing that these two paths are not in any cycle, there is no need to put a FF on s as far as these two paths are concerned. Therefore, we replicate g_2 as shown in Figure 2.2(b). We then retime g_2 by 1 in the replicated circuit as shown in Figure 2.2(c). Now there is a FF on s. Note that s cuts the same set of cycles in both circuits in Figure 2.2(a) and (c). Therefore, the circuit in Figure 2.2(c) is an $\mathcal{S}$-configuration of the circuit in Figure 2.2(a). As is evident from this example, replicating can facilitate retiming because the copies of a vertex can be retimed differently in the replicated circuit as opposed to

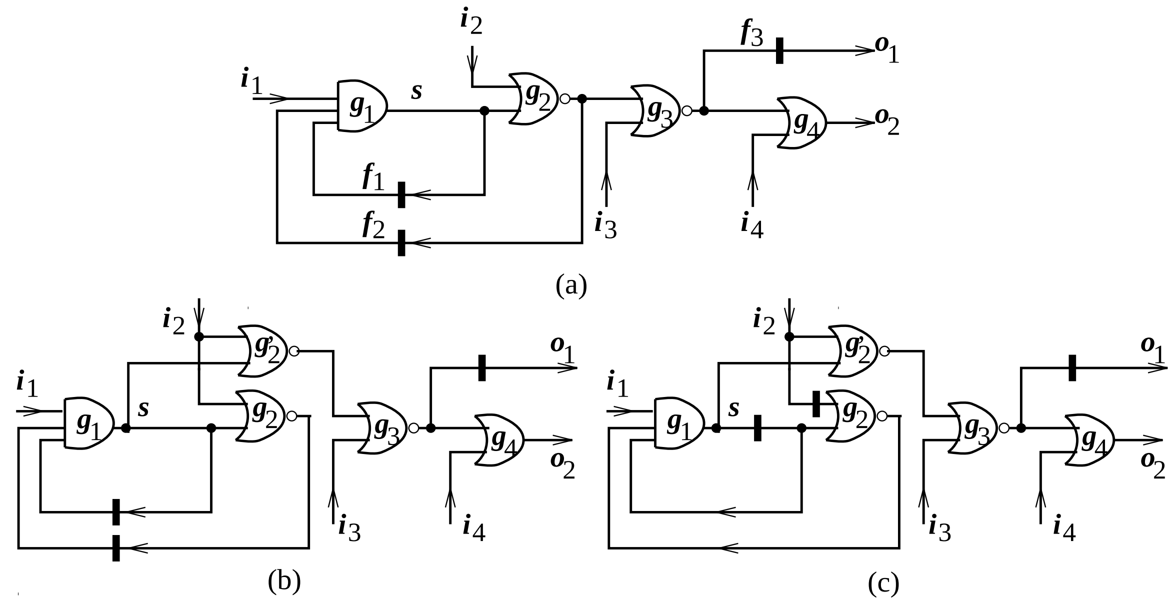


Figure 2.2: Obtaining an $\mathcal{S}$-configuration using replicating and retiming.

that in the original circuit where the copies have to be retimed in the same way since they are actually one vertex.

In the next section, we will present a systematic way to identify a sub-circuit for replication in order to obtain an $\mathcal{S}$-configuration of a circuit.

2.2 A new approach

In this section, we propose a new approach to the problem of determining an $\mathcal{S}$-configuration of a circuit. Our algorithm presented in the next section is based on a refinement of the ideas presented here.

We view the $\mathcal{S}$-edges as the demands and the FFs as the resources. The ultimate question is how to utilize the resources to meet the demands. In our approach, we directly determine the FF movements in order to satisfy the $\mathcal{S}$-edges. First, we examine how to carry out a single FF movement. Suppose s is an $\mathcal{S}$-edge and f is a FF. We want to move f to s. Let A_s and A_f denote the sets of cycles passing s and f, respectively. Let $C_{s,f}$ denote the set of vertices on the paths from the head of s to the tail of the edge containing f. (See Figure 2.3(a).) To move f to s, $C_{s,f}$ is replicated as $C_{s,f}$ and $C'_{s,f}$, and then the vertices in $C_{s,f}$ are retimed by

1 as shown in Figure 2.3(b). Obviously, the set of cycles cut by the FF on s in Figure 2.3(b) is $A_{s,f} = A_s \cap A_f$. Therefore, the problem of determining an $\mathcal{S}$-configuration becomes that of selecting a set of FF movements such that $\cup A_{s,f}$ (over all selected FF movements) contains all the cycles cut by the edges in $\mathcal{S}$. The combinational area overhead is determined by $\cup C_{s,f}$. From Figure 2.3(b) it is also clear that the sequential area overhead (FFs) is determined by the vertices outside $\cup C_{s,f}$ which connect to one or more vertices inside $\cup C_{s,f}$.

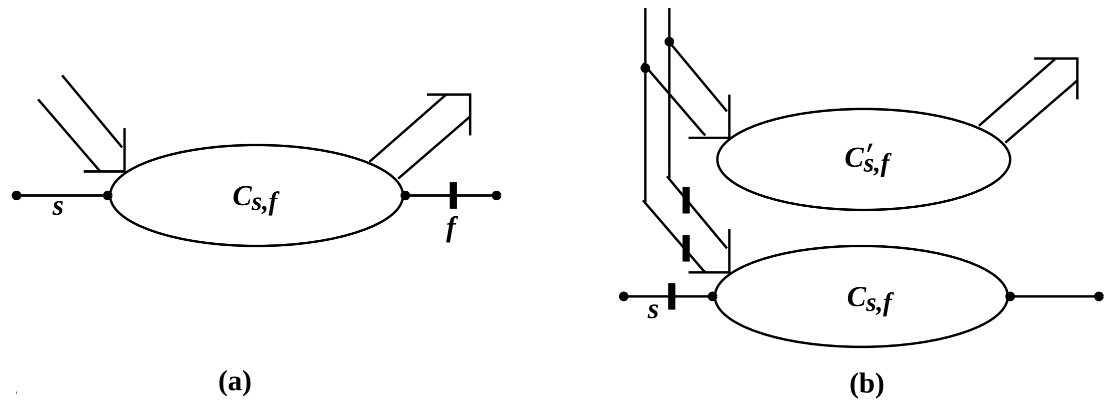


Figure 2.3: Carrying out a single FF movement.

We now determine what kind of FF movements are good for reducing area overheads. Suppose there is a cycle that contains both s and f. Let p be the path from s to f in the cycle. If there is another $\mathcal{S}$-edge s_1 in p, obviously, $C_{s_1,f} \subseteq C_{s,f}$. To reduce combinational area overhead, it is desirable to move f to s_1 instead of s. For a similar reason, if there is a FF f_1 on p, it is desirable to move f_1 instead of f to s. In summary, f is moved to s to cut a cycle only if there is no other $\mathcal{S}$-edges or FFs on the path from s to f in the cycle.

We define a directed graph called the *M-graph* which is introduced to identify (i) all possible FF movements and (ii) all the cycles in $\mathcal{C}$ that are cut by the $\mathcal{S}$-edges. Let $\mathcal{F}$ denote the set of FFs in the circuit. The vertex set of the M-graph is $\mathcal{S} \cup \mathcal{F}$. There are three types of edges in the M-graph: from $\mathcal{S}$-edges to FFs, from $\mathcal{S}$-edges to $\mathcal{S}$-edges, and from FFs to $\mathcal{S}$-edges. The edges from $\mathcal{S}$-edges to FFs are referred to as *M-edges* which are introduced to denote the possible FF movements. There is an M-edge from $s \in \mathcal{S}$ to $f \in \mathcal{F}$ if there is a path in $\mathcal{C}$ from s to f containing no other element in $\mathcal{S} \cup \mathcal{F}$. All other edges are introduced to identify the cycles cut by the $\mathcal{S}$-edges in the circuit. There is an edge from $s_1 \in \mathcal{S}$ to $s_2 \in \mathcal{S}$ if there is a path in $\mathcal{C}$ from s_1 to s_2 containing no other element in $\mathcal{S} \cup \mathcal{F}$, and there is an edge from $f \in \mathcal{F}$ to $s \in \mathcal{S}$ if there is path in $\mathcal{C}$ from f to s containing no other element in $\mathcal{S}$. Note that there is no edge among the vertices in $\mathcal{F}$ in the M-graph. As an example, for the circuit in Figure 2.4(a), Figure 2.4(b) shows its M-graph.

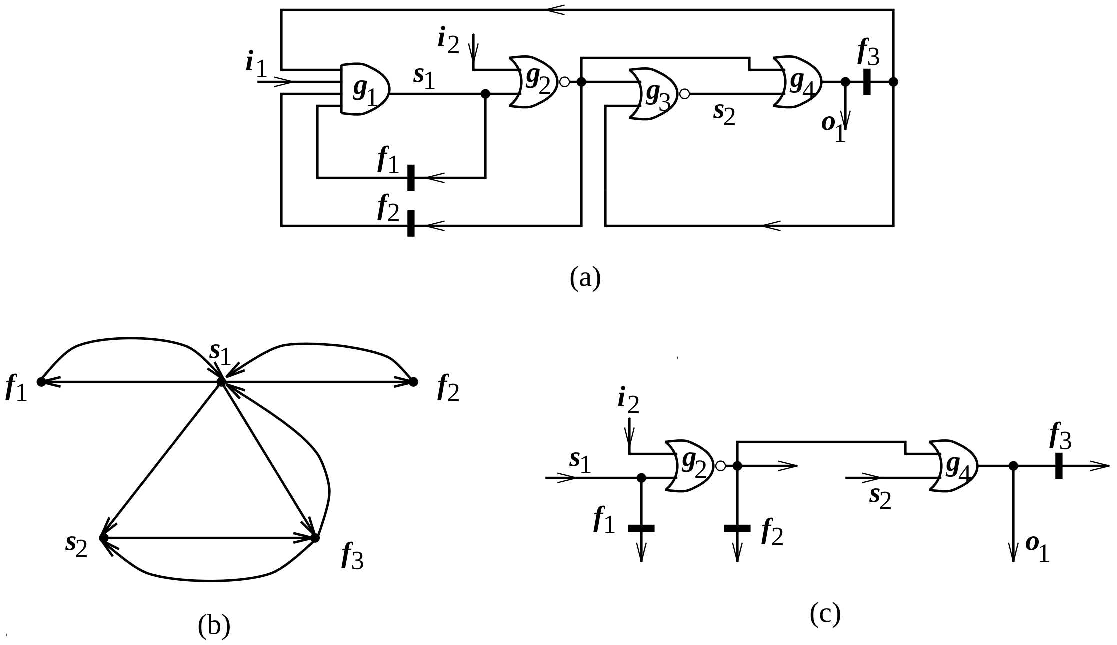


Figure 2.4: (a) A sequential circuit, (b) the corresponding M-graph, (c) the sub-circuit of a cover.

Lemma 2.1 *For each cycle containing one or more $\mathcal{S}$-edges in $\mathcal{C}$, there is a unique corresponding cycle containing at least one M-edge.*

As an example, for the cycle in $\mathcal{C}$ shown in Figure 2.5(a), there is a corresponding cycle in the M-graph as shown in Figure 2.5(b). On the other hand, a cycle in $\mathcal{C}$ containing no $\mathcal{S}$-edge has no counterpart in the M-graph.

If FF f is to be moved to $\mathcal{S}$-edge s, the logic between s and f may be replicated so that the cycles and paths that contain s but not f can "bypass" all the FFs introduced by moving f to s. (See Figure 2.3.) The *associated sub-circuit* of an M-edge (s, f) consists of all gates (together with their interconnections) on the paths from s to f which contains no other element in $\mathcal{S} \cup \mathcal{F}$ in the circuit. The inputs of the sub-circuit are the edges that come from a gate (or PI) not in the sub-circuit and the outputs are the edges that go to a gate (or PO) not in the sub-circuit.

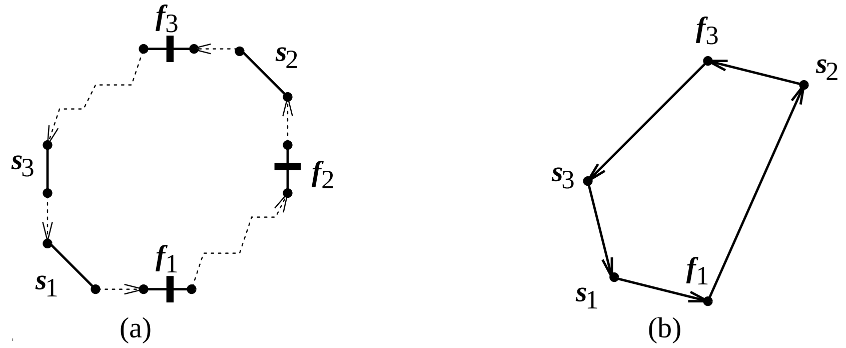


Figure 2.5: (a) Sketch of a cycle (dotted lines are paths containing no FF or $\mathcal{S}$-edge), (b) the corresponding cycle in the M-graph.

As a result, s is one of the inputs of the sub-circuit. As a convention, f is also included in the sub-circuit. The edge that f is on is, of course, an output of the sub-circuit. Similarly, we define the *associated sub-circuit* of a set of M-edges to be the union of the associated sub-circuits of all the M-edges in the set. Again, The inputs are the edges that come from a gate (or PI) not in the sub-circuit and the outputs are the edges that go to a gate (or PO) not in the sub-circuit.

A set of M-edges is called a *cover* if the removal of the edges in the set breaks all the cycles in the M-graph. In other words, a cover is a set of FF movements which, if carried out, guarantee that all the cycles passing through one or more $\mathcal{S}$-edges contain at least one satisfied $\mathcal{S}$-edge. For example, for the cycle in Figure 2.5(a), we can either move f_1 to s_1 or move f_3 to s_2. In terms of covers, either (s_1, f_1) or (s_2, f_3) must be included in a cover in order to break the corresponding cycle in the M-graph in Figure 2.5(b).

Let $\mathcal{M}$ be a cover and $\mathcal{C}_1$ be its associated sub-circuit. See Figure 2.6(a). We introduce a copy of $\mathcal{C}_1$, $\mathcal{C}_1'$, and move the edges without FFs to $\mathcal{C}_1'$ as shown in Figure 2.6(b). This circuit is, then, retimed by moving the FFs to the input edges of $\mathcal{C}_1$ as shown in Figure 2.6(c). Let $\mathcal{C}_\mathcal{M}$ denote the final circuit in Figure 2.6(c). The $\mathcal{S}$-edges that are inputs of $\mathcal{C}_1$ drive FFs in $\mathcal{C}_\mathcal{M}$.

Theorem 2.2 *$\mathcal{C}_\mathcal{M}$ is an $\mathcal{S}$-configuration of $\mathcal{C}$.*

Proof: Obviously, $\mathcal{C}_\mathcal{M}$ and $\mathcal{C}$ have the same functionality. We only need to show that for every cycle containing at least one $\mathcal{S}$-edge in $\mathcal{C}$, it contains at least one satisfied $\mathcal{S}$-edge. Let p be a cycle in $\mathcal{C}$ that contains at least one $\mathcal{S}$-edge. From Lemma 2.1, p has a corresponding cycle in

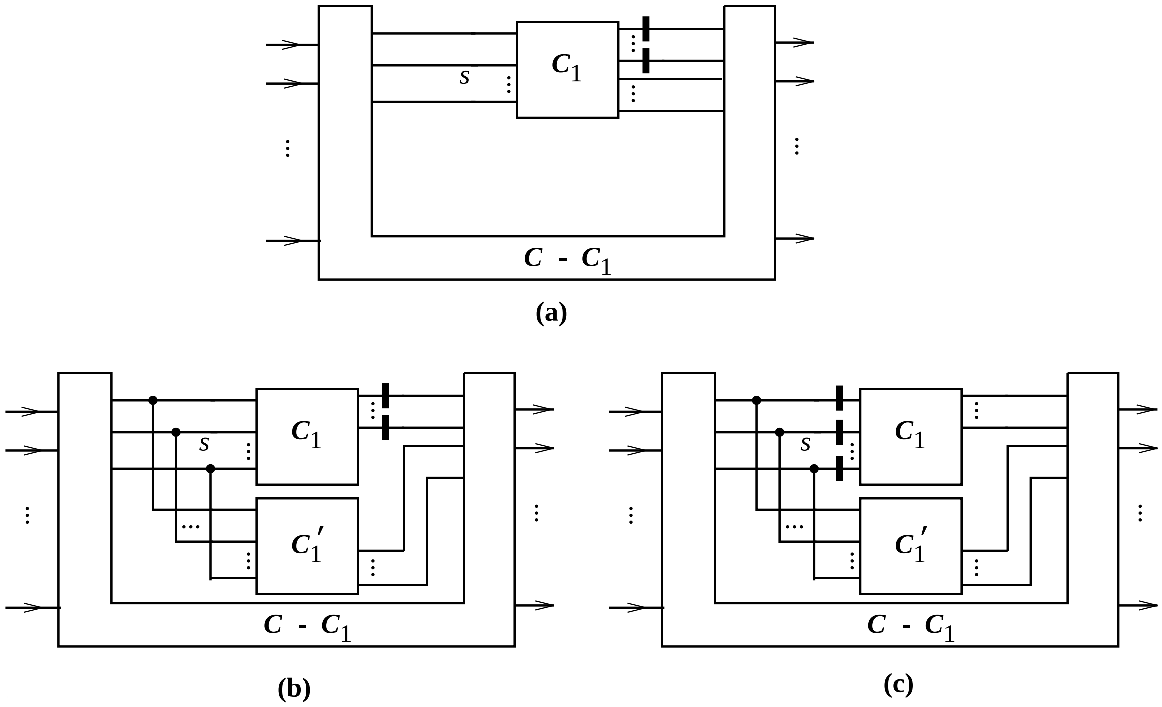


Figure 2.6: (a) Circuit $\mathcal{C}$, (b) $\mathcal{C}$ after replication, (c) $\mathcal{C}_{\mathcal{M}}$.

the M-graph. Let (s, f) be an $\mathcal{S}$-edge in cover $\mathcal{M}$ that breaks this cycle in the M-graph. By construction, p contains the satisfied $\mathcal{S}$-edge s. $\square$

As an example, for the circuit in Figure 2.2(a), its M-graph is shown in Figure 2.7(a) and edges (s, f_1) and (s, f_2) form a cover. Figure 2.7(b) shows the sub-circuit associated with the cover. The $\mathcal{S}$-configuration produced by our approach is exactly the one in Figure 2.2(c). Notice that our approach automatically detects that it is unnecessary to move f_3 to s. As another example, for the circuit in Figure 2.4(a), the only cover in the M-graph in Figure 2.4(b) is the set consisting of all M-edges. Figure 2.4(c) shows the sub-circuit associated with this cover and Figure 2.8 shows the corresponding $\mathcal{S}$-configuration.

2.3 An algorithm for determining an $\mathcal{S}$-configuration

In this section, we present an algorithm for determining an $\mathcal{S}$-configuration of a circuit based on finding a cover in the M-graph of a circuit. A high-level description of the algorithm is presented in Figure 2.9. We now describe each step of the algorithm in details.

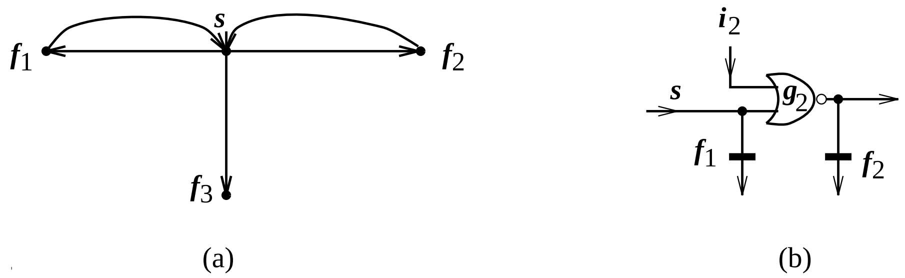


Figure 2.7: (a) M-graph, (b) sub-circuit associated with a cover.

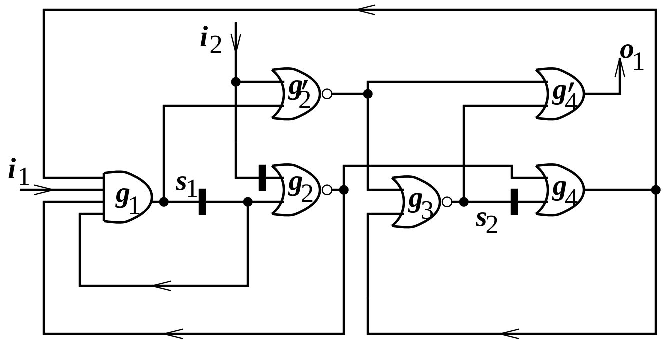


Figure 2.8: $\mathcal{S}$-configuration of the circuit in Figure 2.4.

	Input: A circuit $\mathcal{C}$ and a set of edges $\mathcal{S}$.
	Output: An $\mathcal{S}$-configuration of $\mathcal{C}$.
1.	Determine an initial FF configuration by retiming the circuit;
2.	Compute a cover with small associated sub-circuit in the M-graph of the retimed circuit;
3.	Simultaneously identify the logic to be replicated and minimize the number of FFs in the resulting $\mathcal{S}$-configuration by another retiming;
4.	Generate the $\mathcal{S}$-configuration.

Figure 2.9: Algorithm for determining an $\mathcal{S}$-configuration.

Determine an initial FF configuration: Different initial FF configurations result in different M-graphs. The initial FF configuration also affects the sub-circuits associated with the covers. Intuitively, the closer the FFs to the $\mathcal{S}$-edges, the smaller the sub-circuits associated with the covers will be. We determine an initial FF configuration by retiming $\mathcal{C}$ twice. The purpose of the first retiming is to satisfy as many $\mathcal{S}$-edges as possible. The purpose of the second retiming is to move the FFs toward the unsatisfied $\mathcal{S}$-edges as much as possible while preserving the satisfied ones.

Figure 2.10: Dividing an $\mathcal{S}$-edge.

We now describe how to carry out the first retiming to satisfy a maximum number of $\mathcal{S}$-edges. Let G be a copy of the graph of the circuit with the corresponding edge weights. We modify G in the following way: Each $\mathcal{S}$-edge is divided into two edges s and s' as shown in Figure 2.10. The weight of s is $w(s)$ and the weight of s' is zero. We require that there is at most one FF on s in the retimed circuit. s' is introduced to store any extra FF that otherwise would have to be placed on s. The problem of maximizing the number of $\mathcal{S}$-edges that can be made to drive FFs by retiming becomes that of determining a retiming r_1 such that $w_{r_1}(s) \leq 1$ for each $\mathcal{S}$-edge s and $\sum_{s \in \mathcal{S}} w_{r_1}(s)$ is maximized. The following is the corresponding integer linear program:

$$\textbf{max} \quad \sum_{s \in \mathcal{S}} w_{r_1}(s)$$

subject to:

$$\begin{array}{lcll} w_{r_1}(e) & \geq & 0 & \text{for any edge } e \\ w_{r_1}(s) & \leq & 1 & \text{if } s \text{ is an } \mathcal{S}\text{-edge} \\ r_1(v) & = & 0 & \text{if } v \text{ is a PO or PI} \end{array}$$

Note that the variables in this program are $r(v)$'s. This integer linear program is the dual of a min-cost flow problem. Any min-cost flow algorithm can be used for its solution. After r_1 is determined and applied to the circuit, we obtain a retimed circuit, still denoted $\mathcal{C}$. If all $\mathcal{S}$-edges are satisfied, we are done. In general, this is not the case. Let u$\mathcal{S}$ ($\neq \emptyset$) denote the set of $\mathcal{S}$-edges which have not been satisfied by r_1.

Now we describe the second retiming. Let G be a copy of the graph of the circuit not including the edge weights. We modify the graph by cutting each edge in u$\mathcal{S}$ to create a new PI and a new PO, adding an extra vertex v_0, and connecting v_0 to all PIs and POs (including the newly created ones). We also connect each PI to v_0. We assign weights to the edges in G as follows: The weight of a satisfied $\mathcal{S}$-edge s is $w(s) - 1$. The weight of any other edge e from the circuit is $w(e)$. The weights of all other edges are zero. Note that G contains no negative cycle. Let $r_2(v)$ be the weight of the shortest paths from v to v_0. r_2 is a retiming and is applied to $\mathcal{C}$. It can be easily shown that in the retimed circuit, each satisfied $\mathcal{S}$-edge still drives a FF and no retiming can move any of the FFs further toward the edges in u$\mathcal{S}$.

Compute a cover in the M-graph: To determine a cover in an M-graph, we construct another directed graph called *SM-graph.* The vertices in the SM-graph are the M-edges. If the M-graph is viewed as a circuit with a FF on each M-edge, its MS-graph is simply the S-graph of this circuit. Namely, there is an edge from an M-edge u to an M-edge v in the SM-graph if there is a path containing no other M-edge from u to v in the M-graph. It can be easily verified that a cover in the M-graph corresponds to a feedback vertex set (FVS) in the corresponding MS-graph and vice versa.

A set of vertices in the MS-graph is associated with a sub-circuit — the sub-circuit associated with the M-edges corresponding to the vertices in the set. The problem now is to find a feedback vertex set in the MS-graph such that the sub-circuit associated with the set has minimum cost. The cost of a sub-circuit includes the area for the gates in the sub-circuit and the number of inputs to the sub-circuit. The number of its inputs should be included because after the FF movements are carried out, there is a FF on each of the inputs (see Figure 2.6(c)). Each vertex in the MS-graph can be assigned a cost which is the cost of the sub-circuit associated with the corresponding M-edge. It should be noted that the cost of a set of vertices is not the sum of the costs of the vertices. This is because the sub-circuits associated with the vertices in the MS-graph may overlap. To solve the problem optimally is costly since the minimum feedback vertex set problem (which is known to be NP-hard) is obviously a specially case of this problem. We, therefore, resort to heuristics. Our heuristic algorithm is similar to many heuristics used for determining a minimum feed-back vertex set. The algorithm selects one vertex at a time. Each time, a vertex with minimum value according to a function which is proportional to the cost of the vertex and inversely proportional to the profit in selecting the vertex to break cycles.

	Input: The SM-graph G of circuit $\mathcal{C}$.
	Output: $\mathcal{M}$, a FVS of G with small cost.
1.	$\mathcal{M} \Leftarrow$ the set of vertices with self-loops;
	Update G;
	Update the cost of each vertex in G;
2.	If G is empty, output $\mathcal{M}$ and stop;
3.	$v \Leftarrow$ the vertex with minimum value according to a cost function;
	$\mathcal{M} \Leftarrow \mathcal{M} \cup \{v\}$;
	Update G;
	Update the cost of each vertex in G;
4.	Goto 2;

Figure 2.11: A heuristic algorithm for determining a minimum cost cover.

A simple choice of the function could be:

$$\frac{cost(v)}{d^+(v) + d^-(v)} \tag{2.1}$$

where $cost(v)$ is the cost of the sub-circuit associated with v less the portion overlapped with the sub-circuit associated with the set of vertices that have already been included in the FVS. $d^+(v)$ and $d^-(v)$ are the in-degree and out-degree of v in the current SM-graph. $d^+(v) + d^-(v)$ is just a simple estimate on the number of cycles that can be broken if v is included in the FVS.

After a vertex is selected for inclusion in the FVS, the SM-graph is updated: Edges crossing different strongly connected components are removed and the costs of the vertices are updated. Obviously, any vertex with a self-loop has to be included in the FVS. These vertices are included in the FVS at the very beginning. Figure 2.11 shows the overall algorithm.

Simultaneously identify the logic and minimize the number of FFs: In this step, we determine a retiming of $\mathcal{C}$ aiming at minimizing the number of FFs in the resulting $\mathcal{S}$-configuration and possibly further reducing logic replication. Refer to Figure 2.6(c). The FFs on the outputs of $\mathcal{C}_1$ are moved to the inputs of $\mathcal{C}_1$ in the $\mathcal{S}$-configuration. Therefore, in order to minimize the number of FFs in the $\mathcal{S}$-configuration, we only need to minimize the number of FFs in the remaining part of $\mathcal{C}$. In the meantime, the FFs currently on the outputs of $\mathcal{C}_1$ are retained. This retiming can, again, be determined by solving an integer linear program which is the dual of a min-cost flow problem. Let r_3 be the retiming. r_3 is applied to $\mathcal{C}$.

Generate the $\mathcal{S}$-configuration: This step is shown in Figure 2.6.

2.4 Experimental results

The proposed algorithm was implemented in the C language. To test our algorithm, we also implemented an algorithm for selecting the scan FFs called **SELECT_FF**, and an algorithm for selecting the scan signals called **SELECT_SIGNAL**

We first briefly describe the algorithms used in **SELECT_FF** and **SELECT_SIGNAL**. **SELECT_FF** implements the heuristic for cutting the cycles except self-loops in the S-graph described in [43]. When no graph transformation can be applied, the vertex with the maximum sum of in-degree and out-degree is picked. **SELECT_SIGNAL** implements a heuristic for selecting a set of edges to cut the cycles in a circuit. It is almost the same as the heuristic in **SELECT_FF**. The difference is that in **SELECT_SIGNAL**, the algorithm stops if the S-graph of the remainder of the circuit contains no cycle except self-loops.

The three programs are integrated into one package, which will be referred to as PSPSS. The input to PSPSS is a circuit in *blif* format [91]. The outputs are the the $\mathcal{S}$-configuration (again in *blif* format) determined by our algorithm as well as the scan FFs of the $\mathcal{S}$-configuration determined by **SELECT_FF**.

We tested PSPSS on several LSW'91 sequential multi-level benchmark circuits [91]. The results are summarized in Table 2.1. In Table 2.1, under the column **initial circuit** we list the number of literals (lits), the number of FFs (FF), and the optimal number of scan FFs (scan) quoted from [11]; under the column **final circuit**, we list the corresponding values for the $\mathcal{S}$-configuration produced by PSPSS. Note that for the final circuits, the numbers of scan FFs are determined by **SELECT_FF**. The column **time** lists the total computation time of the integrated package on each circuit on a SPARCstation 10. Overall, our method was able to reduce the number of scan FFs significantly with moderate increase in area as measured by the number of literals and the number of FFs. As an example, for circuit s9234, the initial circuit has 7971 literals, 211 FFs, and 53 scan FFs (optimal), PSPSS produced an equivalent circuit with 8022 literals, 226 FFs, and 32 scan FFs (**SELECT_FF**). As another example, for circuit s38417, the initial circuit has 32246 literals, 1636 FFs, and 374 scan FFs (optimal), PSPSS produced an equivalent circuit with 32250 literals, 1618 FFs, and 135 scan FFs (**SELECT_FF**).

circuit	Initial circuit			Final circuit			time
	lits	FF	scan	lits	FF	scan	sec.
s344	269	15	5	269	17	4	0.4
s349	273	15	5	273	17	4	0.4
s382	306	21	9	306	22	3	0.5
s400	352	21	9	352	22	3	0.6
s444	320	21	9	320	22	3	0.5
s1423	1164	74	21	1164	86	8	5.4
s9234	7971	211	53	8022	226	32	158.6
s13207*	11234	638	58	11333	489	34	233.0
s15850*	13658	534	88	13767	572	41	374.6
s38417	32246	1636	374	32250	1618	135	2680.6

*For this circuit we use the scan signals provided by Chakradhar and Dey.

Table 2.1: Experimental results.

Table 2.2 compares the increases in literals and FFs in the equivalent circuits produced by the algorithm in [12] (PROPS) with that produced by our package (PSPSS) for four large benchmark circuits . As can be seen, the area increases for the equivalent circuits produced by our method are much less. For example, for circuit s9234, in the equivalent circuit produced by PROPS the numbers of literals and FFs are increased by 207 and 48, respectively. However, in the circuit produced by PSPSS the respective increases are 51 and 15. It should be noted that PROPS and PSPSS may use different scan signals for s9234 and s38417.

circuit	lits incr.		FF incr.	
	PROPS	PSPSS	PROPS	PSPSS
s9234	207	51	48	15
s13207	206	99	-51	-149
s15850	253	109	35	38
s38417	31	4	-80	-18

Table 2.2: Comparison between PROPS [12] and PSPSS.

2.5 Summary

In this chapter, we consider the problem of structurally modifying a circuit to make a pre-selected set of signals FF-driving signals so that they can be scanned while preserving the set of cycles being broken. We present a new approach for solving this problem. Based on the new approach we design an efficient algorithm. Unlike a previous algorithm which inherently has no control over the area overhead incurred during the modification, our algorithm explicitly minimizes the area overhead. The algorithm has been implemented and encouraging results were obtained. In the future, we would like to study the problem of selecting signals in a circuit considering both the effects of minimizing the number of scan FFs and reducing the area overhead incurred later. Another seemingly promising approach is to use an ATPG tool to direct signal selection as the one proposed by Chickermane and Patel for selecting scan FFs [17]. We believe this could further reduce scan FFs without sacrificing fault coverage.

Chapter 3

Resynthesis and Retiming for Cycle Time Reduction

Sequential circuit synthesis is a rather complex process, and it is difficult to control the timing performance of a circuit at the early synthesis stage. Consequently, an optimization step to speed-up a sequential circuit is often needed in order to meet the timing requirement. Timing optimization of sequential circuits has received considerable attention [5, 13, 56, 27, 45, 52, 53, 75]. We first briefly summarize some of the past efforts in this area.

An important method for timing optimization of sequential circuits is called *retiming* [45]. Retiming is the process of repositioning the FFs in a sequential circuit without changing its logic. Efficient algorithms for retiming a circuit to satisfy a pre-specified cycle time, if possible, was proposed in [45]. However, the effectiveness of retiming is limited because it does not modify the logic of a circuit.

Combinational logic timing optimization methods such as [77, 80] can be used to optimize the combinational logic between FFs in a sequential circuit. In the conventional approach, combinational logic between FFs are resynthesized so that combinational path delays between FFs are all upper-bounded by the cycle time. Though such an approach is useful, it lacks a global view in two ways. First, signal dependencies across FF boundaries are not exploited. Second, the requirement that all combinational path delays be upper-bounded by the cycle time is unnecessarily stringent since a sequential circuit can be retimed to achieve a given cycle time even though some of its combinational path delays exceed it.

Several methods have been proposed to overcome the weaknesses of a direct application of combinational logic timing optimization methods. In [56, 26, 52], methods were proposed to exploit signal dependencies across FF boundaries through retiming. To expose large combinational parts of a sequential circuit, the concepts of negative FFs and peripheral retiming were introduced in [52]. Later, peripheral retiming was used to optimize the performance of a special class of sequential circuits called pipelined logic circuits [53]. Recently, two new methods for generating timing constraints which are used to direct the optimization of the combinational logic between FFs were proposed [13, 27]. The effect of retiming is taken into consideration in these new methods.

To combine the key features of the prior methods, we want to have a method for generating resynthesis timing constraints in the presence of negative FFs. A major contribution in this chapter is the introduction of a framework for generating resynthesis timing constraints in the presence of negative FFs. Besides allowing the possibility of exploiting signal dependencies across FF boundaries, an additional advantage of allowing the presence of negative FFs is that internal signals (signals that are not primary outputs or FF inputs) can be included in the global generation of timing constraints the same way primary outputs and FF inputs are. Including internal signals in the resynthesis process not only helps to reduce the area penalty incurred during resynthesis, but also gives the resynthesis method more control over the combinational timing optimizer.

Our proposed method generates the resynthesis timing constraints globally. It first extracts an abstraction of a sequential circuit which describes the resources, namely, the FFs (positive as well as negative), and the demands, namely, the combinational delays, in the sequential circuit. Then, the resources are distributed globally so that the demands can be satisfied as much as possible. An integer linear program is used to direct the resource distribution to minimize the unsatisfied path delays. Resynthesis is then carried out to take care of the unsatisfied path delays.

3.1 Preliminaries and problem definition

The problem addressed in this chapter is the following: Given a circuit and a target cycle time ϕ, retime and resynthesize the circuit so that the resulting circuit has cycle time ϕ. We assume a gate delay is an integer which is a multiple of a common unit among the gates.

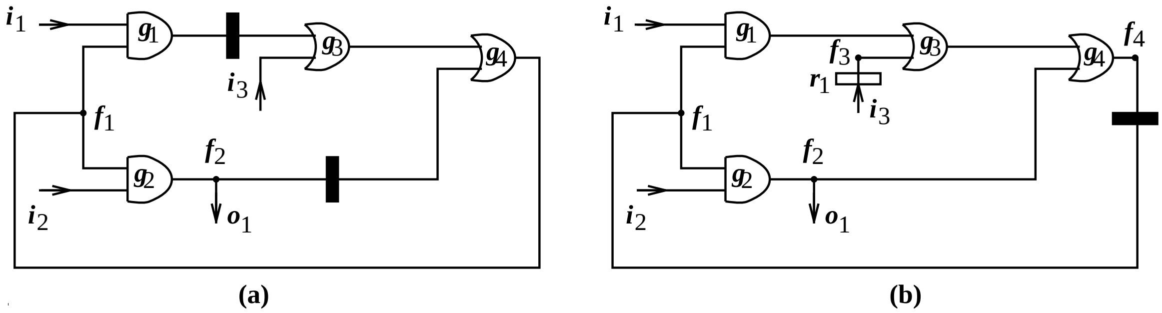


Figure 3.1: (a) A sequential circuit and (b) a retimed circuit.

It was noted in [52] that negative FFs are necessary in order to expose large combinational logic of a sequential circuit. We will see that they are also necessary for timing optimization. The cycle time of the circuit in Figure 3.1(a) is 3 assuming that each gate has unit delay. It can be shown that any combination of the resynthesis and legal retiming steps cannot reduce the cycle time. If we introduce a negative FF r_1 by retiming both g_3 and g_4 all by -1 as shown in Figure 3.1(b). This retimed circuit can be resynthesized and then retimed to a cycle time equal to 2 as will be demonstrated later.

Given a sequential circuit, we construct a corresponding unit-delay circuit[1] as follows: For each gate with a d unit delay, we change its delay to one unit and add $d-1$ unit-delay buffers to the output of the gate. The functionality of the unit-delay circuit is the same as that of the original circuit. The following result explains why we are interested in the unit-delay circuit. Its proof is straightforward.

Theorem 3.1 *If the corresponding unit-delay circuit can be retimed to cycle time ϕ, then the original circuit can be retimed to cycle time $\phi + d_{max} - 1$, where d_{max} is the largest gate delay in the original circuit.* □

[1] A circuit in which all gates have unit delay.

It can be easily shown the bound in Theorem 3.1 is best possible. The unit-delay circuit is useful because of the following necessary and sufficient condition on whether a unit-delay synchronous circuit can be retimed to cycle time ϕ. This theorem can be proved by the method in [45].

Theorem 3.2 *A unit-delay circuit can be retimed to cycle time ϕ iff*

(1) *for any path, P, from a PI to a PO, $d(P) \leq (w(P)+1)\phi$;*

(2) *for any cycle, C, in the circuit, $d(C) \leq w(C)\phi$.*

□

3.2 A timing optimization algorithm

We shall present, in this section, our timing optimization algorithm. We begin with an overview of the algorithm. The key steps of the algorithm will be detailed in Section 3.2.1 and Section 3.2.2.

If all FFs in a sequential circuit are removed, we obtain a combinational network. The inputs of the network are the PI's and the FF outputs of the circuit. The outputs of network are the PO's and the FF inputs of the circuit. Our algorithm will generate a set of timing constraints for the network in the form of arrival times at the inputs and required times at the outputs of the combinational network. Though we do not include internal signals in the definition of the combinational network, their addition is very simple. For those internal signals we have chosen for inclusion, we can simply let them pass through a positive as well as a negative FF.

A high-level description of the algorithm is as follows:

Step 1. *Determine the positions of the FFs.* The objective is to expose signal dependencies across FF boundaries, which, in turn, enhances the resynthesis potentials of the resulting combinational network.

Step 2. *For the combinational network obtained in Step 1 and a given cycle time ϕ, generate a set of resynthesis timing constraints for the network.* A set of resynthesis timing constraint specifies arrival times at the inputs and required times at the outputs of the combinational network. The constraints generated have the following properties: (i) If all the constraints are satisfied during resynthesis, the resulting sequential circuit can be retimed to within the largest gate delay from the the given cycle time; (ii) The total amount of resynthesis work is minimum.

Step 3. *Resynthesize the combinational network to satisfy the timing constraints computed in the Step 2.*

Step 4. *Retime the resynthesized sequential circuit to minimize the cycle time.*

For Step 1, the methods in [26, 52] can be used. For our current implementation, we minimize the number of FFs inside the circuit (excluding the PI's and PO's) by retiming. Therefore, we can only introduce negative FFs at the the PI's and PO's. For the resynthesis of combinational network (Step 3), we use the *speed_up* program in **SIS** [74, 77]. As for Step 4, we use the *retime* program in **SIS**. The key step of the timing optimization algorithm is Step 2 — generating the resynthesis timing constraints with the desired properties. The details of this step are presented in Section 3.2.1 and Section 3.2.2.

3.2.1 A framework for timing constraint generation

In this subsection we present a general framework for the generation of timing constraints. Any set of constraints generated within this framework guarantees that the resynthesized sequential circuit can be retimed to within the largest gate delay from the given cycle time if the constraints are all satisfied by resynthesis. Timing constraint generation in this framework can be viewed as a "slack" distribution process. Slack distribution in turn is carried out globally by "moving" the slacks from one place to another.

For a sequential circuit, we define a directed graph G, called the *skeleton graph* (of the circuit). The vertices are the PI's, the PO's, and the signals input to or output from a FF. The edges of G are divided into two classes: *FF edges* and *path edges*. e is a FF edge in the skeleton graph if e is an edge in the circuit such that $w(e) \neq 0$. There is a path edge $e = (u, v)$ in the skeleton graph if there is a combinational path from u to v in the circuit. As an example, the graph shown in Figure 3.2 is the skeleton graph of the circuit in Figure 3.1(b). The two dashed edges are FF edges, they are edges in the circuit also. All other edges are path edges. A skeleton graph is an abstraction of the sequential circuit: All edges with FFs are retained and all combinational paths between two FFs is abstracted to be the path edge with the same starting and ending vertices. If P is a path in the circuit, there is a unique corresponding path P' in G. P' can be constructed by replacing each maximal combinational subpath with the corresponding path edge in G. Since the FF positions have already been fixed when we generate the timing constraints, the skeleton graph of a resynthesized circuit is the same as that of the circuit before resynthesis.

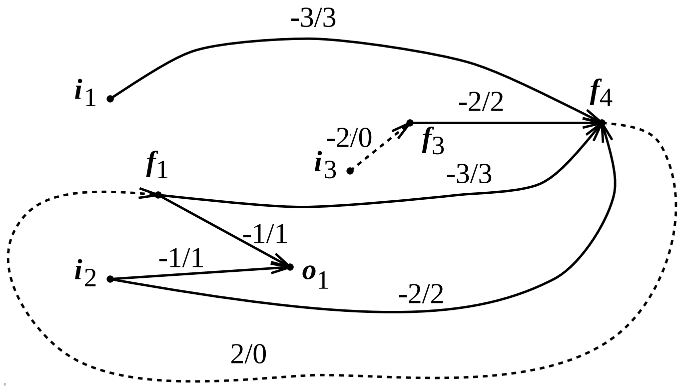


Figure 3.2: A skeleton graph, the numbers on each edge are s/D values.

We introduce two functions, D and s, on the edges of G to describe the initial resources and demands distribution in the circuit. They are defined as follows:

$$D(e) = \begin{cases} 0 & \text{if } e \text{ is a FF edge} \\ \text{maximum path delay among all} & \\ \text{combinational path from } u \text{ to } v & \text{if } e = (u,v) \text{ is a path edge} \end{cases}$$

$$s(e) = \begin{cases} w(e)\phi & \text{if } e \text{ is a FF edge} \\ -D(e) & \text{if } e \text{ is a path edge} \end{cases}$$

D is called the *delay function* and s is called the *slack function*. The delay function represents the path delays between the inputs and the outputs of the combinational network of the circuit. The slack function represents the initial slacks (resoures as well as demands) presented in the circuit when the required cycle time is ϕ. At the beginning, a slack ϕ is given to each positive FF, a slack $-\phi$ is given to each negative FF. In other words, we initially lump all resources in the circuit to the FF edges. Because of this all combinational path delays become demands initially. For the skeleton graph shown in Figure 3.2, the numbers beside each edge are its s and D values in the form s/D for $\phi = 2$. The following result is evident from the definitions of $D(e)$ and $s(e)$.

Lemma 3.1 *Let P' be a path in G, then $D(P') + s(P') = w(P')\phi$ where $w(P')$ denotes the sum of the weights of all FF edges on P' in the initial circuit.*

As was said before, the initial slack function denotes the initial distribution (configuration) of resources and demands in the circuit. We now introduce an operation to redistribute the resources and demands, which is called *slack redistribution.* A slack redistribution is a mapping L from the vertices of G to the integers such that

$$\begin{aligned} s_L(e) &= s(e) + L(v) - L(u) && \text{where } e = (u, v) \\ L(v) &= 0 && \text{if } v \text{ is a PO} \\ L(v) &= -\phi && \text{if } v \text{ is a PI} \end{aligned}$$

Slack redistribution looks very much like the retiming operation. Its purpose is to move the slacks around in the skeleton graph, but to keep the total slack on each cycle and each path from a PI to a PO an invariant. Let D' denote the delay function of a resynthesized circuit. The following result exhibits the importance of a slack redistribution:

Theorem 3.3 *If $D'(e) \leq D(e) + s_L(e)$ for any edge e in the skeleton graph G, then the corresponding unit-delay circuit of the resynthesized circuit can be retimed to cycle time ϕ.*

Proof: Let P be a path from a PI to a PO in the resynthesized circuit and $d'(P)$ be its delay. Let P' be the path corresponding to P in G. Then,

$$d'(P) \leq D'(P') \leq D(P') + s_L(P')$$

According to the definition of slack redistribution, $s_L(P') = \phi + s(P')$. Thus,

$$\begin{aligned} d'(P) &\leq D(P') + \phi + s(P') \\ &= (w(P') + 1)\phi \qquad \text{(by Lemma 3.1)} \\ &= (w(P) + 1)\phi \end{aligned}$$

□

Let C be a cycle in the resynthesized circuit and $d'(C)$ be its delay. Let C' be the cycle corresponding to C in G. Then, similarly,

$$d'(C) \leq D'(C') \leq D(C') + s_L(C')$$

According to the definition of slack redistribution, $s_L(C') = s(C')$. Thus,

$$\begin{aligned} d'(C) \quad \leq \quad & D(C') + s(C') \\ & = w(C')\phi \qquad \text{(by Lemma 3.1)} \\ & = w(C)\phi \end{aligned}$$

□

Theorem 3.3 will be used to generate the path delay bounds for the resynthesis of the combinational network. To be specific, for a path edge $e = (u, v)$, we require that the maximum combinational path delay from u to v, $D'(e)$, be upper-bounded by $D(e)+s_L(e)$ after resynthesis. Thus, $s_L(e)$ denotes the amount of anticipated delay reduction in the circuit. (This is the reason we call s_L slack function.) If after resynthesis, the upper-bounds for all path edges are satisfied, according to Theorem 3.3, the corresponding unit-delay circuit of the resynthesized circuit can be retimed to cycle time ϕ. Then, by Theorem 3.1 the resynthesized circuit can be retimed to within the largest gate delay of ϕ.

Timing information input to a combinational logic timing optimizer is often specified in the form of arrival times at the inputs and required times at the output of a combinational network. We need to translate the path delay bounds into arrival times and required times. For the combinational network of the sequential circuit, let v be an output and u be an input. Then $e = (u, v)$ is a path edge in G. The path delay constraint is

$$\begin{aligned} D'(e) \quad \leq \quad & D(e) + s_L(e) \\ & = D(e) + s(e) + L(v) - L(u) \\ & = L(v) - L(u) \end{aligned}$$

Thus, to realize all the path delay constraints, we assign the arrival times and required times for the combinational network as follows: The required time at an output v is $L(v)$; the arrival time at an input u is $L(u)$.

3.2.2 Generation of timing constraints

Within our framework, the problem of timing constraint generation is reduced to that of determining a slack redistribution. Different slack redistributions yield different sets of resynthesis timing constraints, some may be satisfiable, some may not. We now consider how to determine a set of resynthesis timing constraints that is more likely to be satisfiable. Let us examine the path delay constraint $D'(e) \leq D(e) + s_L(e)$, where $e = (u, v)$ and $D(e)$ (resp. $D'(e)$) is the largest combinational path delay from u to v in the original (resp. resynthesized) circuit. If $s_L(e) < 0$, the path delay from u to v must be reduced by at least $-s_L(e)$. On the other hand, if $s_L(e) \geq 0$, the path delay from u to v need not be reduced. In fact, it can tolerate an increase up to $s_L(e)$. In the extreme case, we have the following result:

Theorem 3.4 *The corresponding unit-delay circuit of a sequential circuit can be retimed to cycle time ϕ (without any resynthesis) iff there is a slack redistribution L such that $s_L(e) \geq 0$ for any path edge e in G.*

Proof: ($\Longleftarrow$) Directly from Theorem 3.2.

($\Longrightarrow$) Take s as the weight function for the edges in G. From the fact that $d(C) \leq w(C)\phi$ for any cycle C in the circuit, it is easy to see that G contains no negative cycle. For a vertex v in G such that v is not a PI (of the circuit), let $L(v)$ denote the length of a shortest path among all paths from v to the PO's. If v is a PI, simply let $L(v) = -\phi$. It can be easily verified that L is a slack redistribution such that $s_L(e) \geq 0$ for any edge e in G. $\square$

Intuitively, the larger the sum of all negative slacks in the skeleton graph is, the more difficult the corresponding resynthesis task becomes. Thus, to increase the potential of a successful resynthesis, it is desirable to redistribute the slacks to minimize the sum of all negative slacks in the skeleton graph. Also note that not all edges can have negative slacks. For example, none of the FF edges can have a negative slack since they cannot be resynthesized. Some path edges may have such a small delay that further reduction is very unlikely. In general, we can classify all edges in the skeleton graph as resynthesizable (for those that can be resynthesized) and nonresynthesizable (for those that cannot be resynthesized).

Let t_e denote the amount of negative slack on edge e. Obviously, $s_L(e) + t_e \geq 0$ and $t_e \geq 0$. So, we have the following integer linear program which minimizes the sum of all negative slacks in the skeleton graph:

$$\textbf{minimize} \quad \textstyle\sum_e t_e$$

subject to:

$$\begin{array}{cccl} s_L(e) + t_e & \geq & 0 & \\ t_e & \geq & 0 & \text{if } e \text{ is resynthesizable} \\ s_L(e) & \geq & 0 & \text{if } e \text{ is nonresynthesizable} \\ L(v) & = & 0 & \text{if } v \text{ is a PO} \\ L(v) & = & -\phi & \text{if } v \text{ is a PI} \end{array}$$

Solving the above integer linear program will determine a slack redistribution, which in turn determines a set of resynthesis timing constraints for the combinational network. The above linear program can be further enhanced. For example, we can put an upper-bound on t_e in case e can only tolerate a limited amount of reduction.

3.3 An example

We now present an example to illustrate our algorithm. For the circuit shown in Figure 3.1(a), the minimum cycle time that can be achieved through any combination of the resynthesis and retiming setps is 3 if no negative FFs are introduced. We now show how its cycle time can be reduced to 2. After minimizing the number of FFs inside the circuit, we have the new FF positions as shown in Figure 3.1(b). Note that there is a negative FF r_1 on primary input i_3. Figure 3.2 shows its skeleton graph along with the s and D values. Obviously, only edges $e_1 = (i_1, f_4)$, $e_2 = (f_3, f_4)$, $e_3 = (f_1, f_4)$, and $e_4 = (i_2, f_4)$ can possibly be resynthesized. We have the following integer linear program for slack redistribution:

$$\textbf{minimize} \quad t_{e_1} + t_{e_2} + t_{e_3} + t_{e_4}$$

subject to:

$$
\begin{aligned}
L(f_4) - L(i_1) - 3 + t_{e_1} &\geq 0 \\
L(f_4) - L(f_3) - 2 + t_{e_2} &\geq 0 \\
L(f_4) - L(f_1) - 3 + t_{e_3} &\geq 0 \\
L(f_4) - L(i_2) - 2 + t_{e_4} &\geq 0 \\
L(f_3) - L(i_3) - 2 &\geq 0 \\
L(f_1) - L(f_4) + 2 &\geq 0 \\
L(o_1) - L(f_1) - 1 &\geq 0 \\
L(o_1) - L(i_2) - 1 &\geq 0 \\
L(o_1) &= 0 \\
L(i_1) = L(i_2) = L(i_3) &= -2
\end{aligned}
$$

Solving the integer linear program, we have

$$
\begin{aligned}
&L(o_1) = 0; \\
&L(i_1) = L(i_2) = L(i_3) = -2; \\
&L(f_4) = 1; L(f_3) = 0; \\
&L(f_1) = -1.
\end{aligned}
$$

The slacks in the skeleton graph after redistribution are shown in Figure 3.3. As can be seen, e_2 and e_3 must be reduced by one unit each. e_4 can tolerate one unit increase. e_1 need not be reduced and it cannot tolerate any increase either.

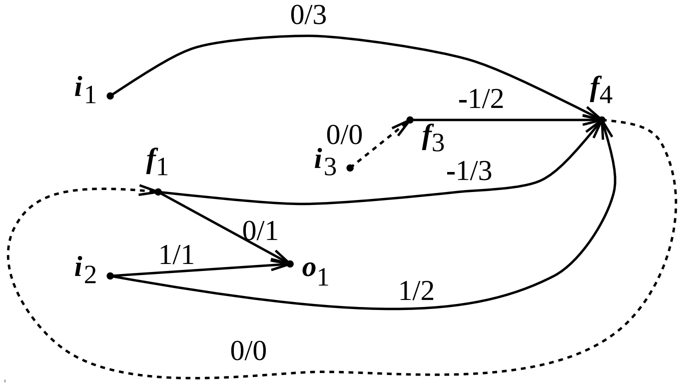


Figure 3.3: The slacks after redistribution.

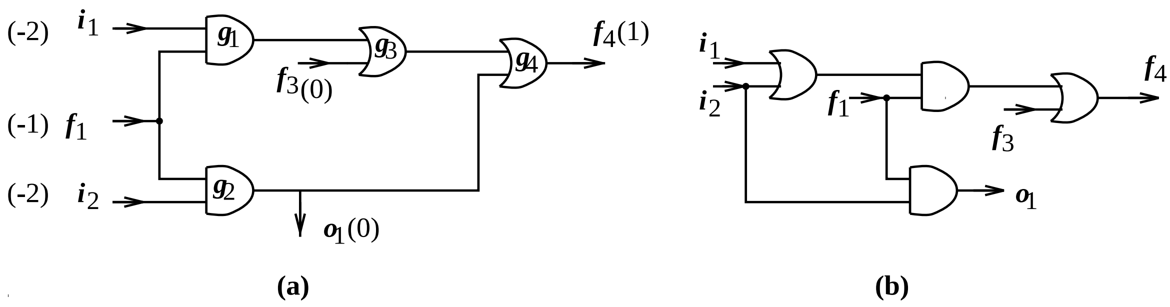


Figure 3.4: The network and a resynthesized one satisfying all timing constraints.

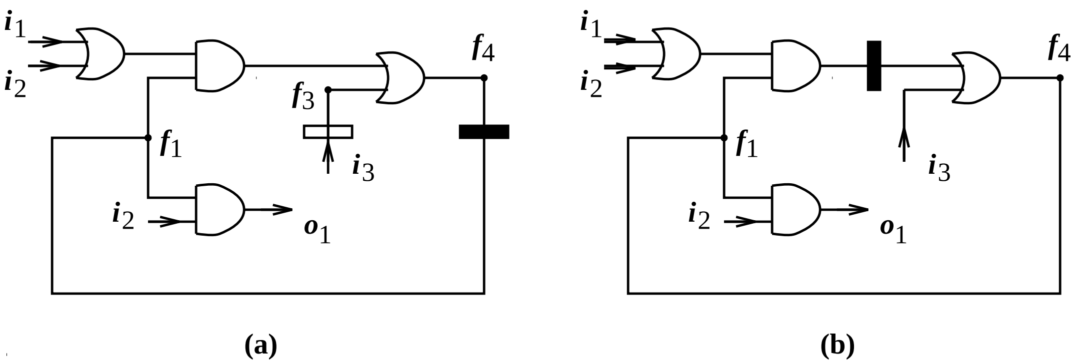


Figure 3.5: The resynthesized circuit and its retimed one.

The combinational network is shown in Figure 3.4(a). Also shown in Figure 3.4(a) are the arrival and required times. The resynthesized network that satisfies the timing requirement is shown in Figure 3.4(b). Figure 3.5(a) shows the resynthesized circuit and Figure 3.5(b) shows the retimed circuit of the resynthesized circuit which achieves the desired cycle time 2.

3.4 Experimental results

We implemented a version of the proposed algorithm in C and tested it on a number of MCNC benchmark sequential circuits. Our implementation has four steps: reposition the FFs, generate resynthesis timing constraints, speed up the combinational network according to the timing constraints generated and a final retiming. For repositioning the FFs, we currently use a very simple method. We minimize the total number of FFs inside the circuit (excluding the PI's and PO's) by retiming, that is, we only allow negative FFs at the inputs and the outputs of the circuit, but no negative FFs inside the circuit. To minmize the number of FFs in the circuit, we used a very fast mincost flow program from Stanford. To solve the integer linear program for generating timing contraints, we use the commerical linear programming package **LINDO**. For combinational network resynthesis and the final retiming, we use the *speed_up* and *retime* programs in **SIS**. We use the simple unit-fanout delay model. Our current implementation does not generate timing constraints for internal signals.

Table 3.1 and Table 3.2 summarize our experimental results. Each initial circuit is derived from the corresponding MCNC benchmark sequential circuit by using *tech_decomp -a 2* command in **SIS**. Among the MCNC benchmark sequential circuits we experimented with, negative FFs can be introduced by our current implementation only in six of them. Table 3.1 shows the experimental results for these six circuits. Under column **Initial** we list the number of literals (lits), the number of FFs (FFs), and the cycle time (ct) of each initial circuit. Column **Ret + Sp** lists the parameters of the resulting circuit by applying *retime* then *speed_up* in **SIS** to each initial circuit. Similarly, column **Sp + Ret** lists the parameters of the resulting circuit by applying *speed_up* then *retime* in **SIS** to each initial circuit. Column **Ours** lists the the parameters of the resulting circuit by applying our algorithm to each initial circuit. For *speed_up* we used the *-d 6* option in all the experiments. For *retime*, we used the option to minimize the cycle time.

circuit	**Initial**			**Ret + Sp**			**Sp + Ret**			**Ours**		
	lits	FFs	ct	lits	FFs	ct	lits	FFs	ct	lits	FFs	ct
s382	300	21	19.2	287	52	9.6	262	41	9.6	271	43	9.0
s400	320	21	19.0	283	32	9.8	320	39	10.0	275	31	9.4
s444	342	21	20.4	325	36	9.8	362	59	10.8	285	32	9.4
s641	342	19	36.6	306	19	34.4	482	17	24.6	550	34	19.0
s713	367	19	36.0	420	19	23.8	420	17	22.4	728	28	18.8
s1423	1019	74	82.6	1537	76	37.2	1366	85	38.2	1976	129	35.0

Table 3.1: Experimental results for circuits with negative FFs introduced.

We also tested a set of benchmark sequential circuits for which our current implementation cannot introduce negative FFs. The experiment setting is the same as that for the previous one. The results are shown in Tabel 2 with the columns having the same meaning as those in Table 3.1.

circuit	**Initial**			**Ret + Sp**			**Sp + Ret**			**Ours**		
	lits	FFs	ct	lits	FFs	ct	lits	FFs	ct	lits	FFs	ct
s349	241	15	20.0	227	25	14.2	265	18	13.8	270	21	12.6
s420	343	16	16.8	339	33	14.0	387	16	14.8	349	39	12.4
s526	507	21	15.0	423	55	10.6	450	51	10.8	474	71	9.6
s820	939	5	33.0	705	6	25.4	719	6	22.8	761	10	20.8
s832	967	5	33.6	723	6	25.6	737	6	23.0	761	11	21.0
mm9a	1063	27	76.4	1306	27	50.8	1306	27	50.8	2038	29	47.6

Table 3.2: Experimental results for circuits without negative FFs introduced.

For all the tested circuits, repositioning FFs and generating timing constraints only took a couple of seconds. As can be seen from the experimental results our algorithm consistently obtained the best cycle time. As for the number of literals and number of FFs, our algorithm is comparable to **Ret + Sp** and **Sp + Ret** in most of the examples. However, there are several cases, our algorithm showed substantial increases in both counts. Our experience seems to suggest that the larger the reduction in cycle time, the heavier the area penalty is. For example, for circuit s641 our algorithm can further reduce its cycle time from 19 to 16 with 683 literals and 41 FFs in the resulting circuit. It will be very interesting to study the trade-off between cycle time and area.

3.5 Summary

In this chapter, we present a method for timing optimization of synchronous sequential circuits. The proposed method first generates a set of timing constraints, then uses the timing constraints to direct a combinational timing optimizer to resynthesize the combinational network of the sequential circuit. We propose a general framework for timing constraint generation. Within this framework, negative FFs are allowed in the circuits. This is an important feature because negative FFs are often introduced in exploit signal dependencies across FF boundaries. Our prelimnary experimental results indicate the approach is promising.

There are several problems we would like to investigate in the future. One obvious problem is to examine other methods for introducing negative FFs in order to fully exploit the potential of our timing constraint generation method. We also would like to carry out research on how to include internal signals in the resynthesis process directly to gain more control over the combinational timing optimizers so that the timing constraints can be better utilized.

Chapter 4

Technology Mapping of Sequential Circuits for LUT-based FPGAs

Field programmable gate arrays (FPGAs) has evolved rapidly as an important ASIC technology. Important features of FPGAs are low manufacturing cost for low volume designs, short design cycle, and reprogrammability. These features make FPGAs particularly attractive for such applications as design prototyping and hardware emulation.

In this chapter we confine ourselves to look-up table (LUT) based FPGA architectures [89]. A LUT-based FPGA consists of an array of programmable logic blocks (PLBs) together with programmable interconnections. The core of a PLB is a k-input LUT (k-LUT) which can implement any combinational logic with up to k inputs and a single output, where k is a positive integer ranging usually from 3 to 9. There are also a few flip-flops (FFs) in each PLB which can be connected to the inputs and outputs of the LUT to realize sequential behavior.

The technology mapping problem for LUT-based FPGAs is to produce, for a given circuit, an equivalent circuit comprising of k-LUTs. This problem has been studied extensively. However, almost all mapping algorithms were designed for combinational circuits. Mapping algorithms for combinational circuits have been proposed for different optimization criteria: optimizing the performance [15, 21, 23, 20, 34, 55, 60, 72, 90], minimizing the number of LUTs [31, 32, 33, 39, 58, 59, 88], improving the routability [6, 73], and combinations of multiple objectives [22, 71]. We mention here particularly FlowMap, a mapping algorithm by Cong and Ding [21].

For a k-bounded combinational circuit, FlowMap is able to produce a mapping solution with minimum delay under the unit delay model.

The conventional approach to technology mapping of sequential circuits is to use a mapping algorithm for combinational circuits to map the combinational logic between the FFs. Specifically, the FFs in a sequential circuit are removed to obtain a combinational network. Then the combinational network is mapped. Finally, the FFs are connected back (to the mapping solution of the combinational network) to generate a mapping solution for the sequential circuit. This approach has, however, two shortcomings: (i) It fails to consider the signal dependencies across FF boundaries, and (ii) it does not exploit the possibility of exposing the combinational logic between the FFs in different ways, through retiming and replication — two functionality-preserving transformations.

Recently, researchers have begun to investigate the mapping problem for sequential circuits directly. Murgai *et al.* [57] proposed mapping algorithms with the objective of minimizing the number of LUTs. They, however, assume that the positions of the FFs are fixed as in the conventional approach. Weinmann and Rosenstiel [84] proposed a heuristic mapping algorithm for loopless sequential circuits.

A. Motivations

The conventional approach has two shortcoming: (i) It fails to consider the signal dependencies across FF boundaries, and (ii) it does not exploit the possibility of exposing the combinational logic between the FFs in different ways, for example, through retiming. Actually, we notice that even with an optimal mapping algorithm for combinational circuits, the the conventional approach may not produce optimal mapping solutions for sequential circuits. As an example, consider the circuit in Figure 4.1(a). Suppose $k = 4$. If we do not retime the circuit, it can be easily verified that either c or d must be an input to the LUT for output o_1. As a result, any mapping solution has cycle time at least 2. One possible mapping solution is shown in Figure 4.1(a) by the dashed lines. This mapping solution has not only cycle time 2, but also 4 LUTs. On the other hand, if we move both f_1 and f_2 backward as shown in Figure 4.1(b), we have the mapping solution with one LUT shown in Figure 4.1(c), where the LUT is formed by the logic delineated by the dashed lines in Figure 4.1(b). Notice that this mapping solution has cycle time 1.

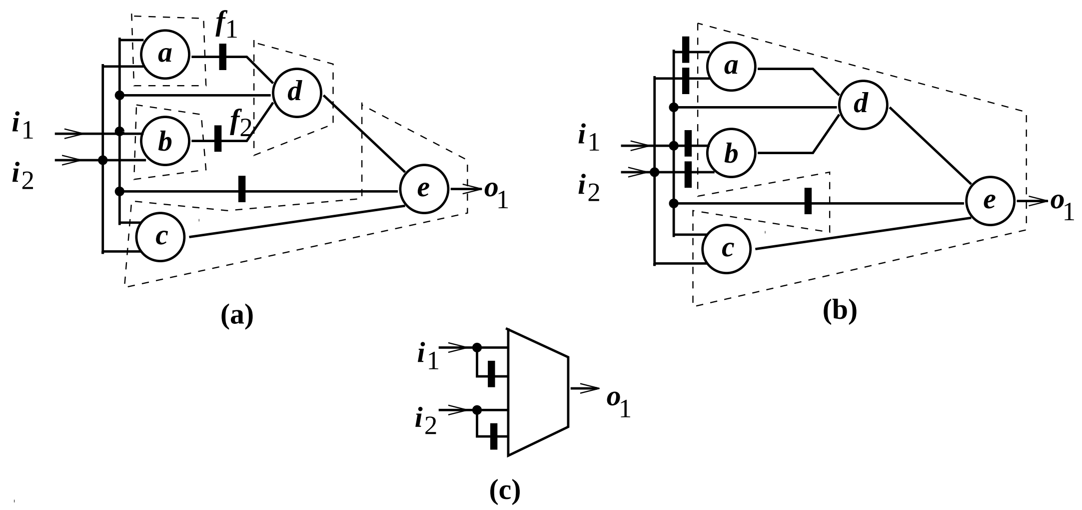


Figure 4.1: Necessity of retiming.

From the above example, we know that retiming should be integrated into technology mapping in order to obtain the best mapping solutions. We further notice that to fully exploit the potential of retiming, logic replication is also necessary. Replication can help produce mapping solutions which are otherwise impossible. Consider the circuit in Figure 4.2(a), Assume $k = 4$. Suppose we are not allowed to replicate any node. If we do not retime the circuit, either d, e, or f must be an input to the LUT for o_1. Consequently, any mapping solution must have cycle time at least 2. On the other hand, suppose, for example, that we move FF f out of x. A so-called negative FF will be introduced on y. This means that y will be broken in the corresponding combinational circuit. Again, it can be verified that any mapping solution will have cycle time at least 2. However, if we duplicate a (to become a_1 and a_2), b (to become b_1 and b_2), and c (to become c_1 and c_2), then retime a_2, b_2, and c_2 all by 1 as shown in Figure 4.2(b), we have the mapping solution with one LUT in Figure 4.2(c), where the LUT is formed by the logic delineated by the dashed lines in Figure 4.2(b). Note that the mapping solution in Figure 4.2(c) has cycle time 1.

B. Contributions

In this chapter we examine the technology mapping problem for sequential circuits for LUT-based FPGAs in the most general setting. Conceptually, our problem formulation can be described by the diagram in Figure 4.3. Namely, the mapping solution space consists of all the

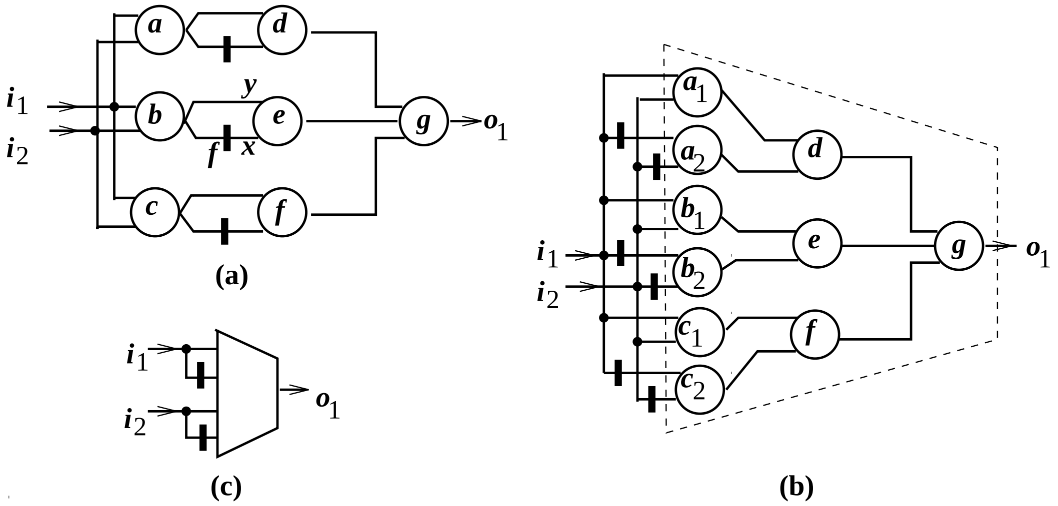


Figure 4.2: Necessity of logic replication.

circuits that can be obtained by retiming and replicating the given initial circuit, then mapping the combinational logic between the FFs, following with another retiming and replication. Note that in the conventional approach, Step 1 and Step 3 in Figure 4.3 are missing. The objective of the technology mapping problem addressed here is to obtain a mapping solution with minimum cycle time, that is, the maximum delay between any two successive FFs in the mapping solution is minimum. We will present an efficient algorithm for solving this problem. Specifically, we will present an algorithm that can produce a minimum cycle time mapping solution for a loopless sequential circuit, under the unit delay model. Furthermore, the algorithm is extended to the general delay model. In that case, it produces a mapping solution with provably good cycle time.

We should emphasize at this point that what Figure 4.3 shows is the definition of the solution space, not the steps that our algorithms take to solve the problem. Actually to try to obtain a mapping solution by carrying out the steps in the figure sequentially is exactly what an algorithm based on the conventional approach does, and it inevitably arrives at a sub-optimal solution since there is no way to know what retiming and replication to use before mapping the combinational logic between the FFs. Another way to understand our algorithms is that they combine retiming and replication with mapping combinational logic seamlessly and globally to obtain the best mapping solutions.

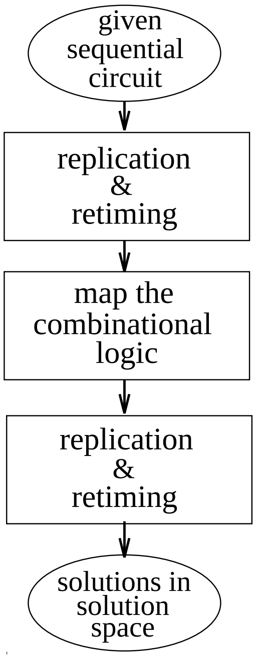


Figure 4.3: Solution space after integrating retiming and replication.

It is easy to see that the solution space in our formulation is enormous since there are too many ways to retime and replicate a circuit. One of our contributions is to identify a much smaller subset of the solution space that is sufficient for obtaining a minimum cycle time mapping solution. An important issue in technology mapping for LUT-based FPGAs is the formation of LUTs. Though this issue is relatively simple for combinational circuits, it is complicated for sequential circuits because of retiming and replication. Another contribution is to derive a method to forming LUTs in sequential circuits by introducing the concept of expanded circuits. Our algorithm is also based on the general idea of node labeling [42]. In this chapter, we introduce a labeling scheme that takes into consideration both the delays and FFs in a circuit. We believe that the results presented here constitute a significant step toward the understanding of technology mapping for LUT-based FPGAs.

4.1 Preliminaries and problem definition

Let v be a node in N. We use N_v to denote the subcircuit induced by the nodes that can reach v. The output of N_v is the output of v. We use $Ein(v)$ to denote the set of incoming edges of

v, $Eout(v)$ to denote the set of outgoing edges of v, and f_v to denote the output function of v. For a path p in N, let $d(p)$ denote the total delay on p and $w(p)$ denote the sum of the weights of the edges (i.e., the total number of FFs) on p.

If we remove all the FFs from N, we obtain a combinational circuit. The inputs of the combinational circuit are the PIs of N and the outputs of the removed FFs, and its outputs are the POs of N and the inputs of the removed FFs. We call the resulting circuit the *combinational circuit* of N.

If r is a retiming of N, we use N_r to denote the circuit obtained by applying r to N. Let $f_v^{(i)}$ denote function f_v delayed by i clock cycles (i.e., $f_v^{(i)}$ at clock cycle $t+i$ is equal to f_v at clock cycle t). It follows that the output function at v is $f_v^{(i)}$ in N_r.

The *cycle time* $\Phi(N)$ of circuit N is the maximum delay on the combinational paths (paths without FFs) in N, provided that there is no negative FF in N; otherwise, we define $\Phi(N) = +\infty$. The *optimal cycle time* $\Phi_{opt}(N)$ of N is defined to be the minimum cycle time of all the circuits that can be obtained by retiming N. Namely,

$$\Phi_{opt}(N) = \min\{\Phi(N_r) \mid r \text{ is a retiming of } N\}.$$

A retiming r such that $\Phi(N_r) = \Phi_{opt}(N)$ is called an *optimal retiming*. Efficient algorithms for computing an optimal retiming are presented in [45].

Now we state the technology mapping problem for LUT based FPGAs formally. Refer again to Figure 4.3. Suppose N is a circuit in a generic gate representation. We assume that N is *k-bounded*, i.e., $|Ein(v)| \leq k$ for any node v in N. Let N' be a circuit obtained from N using replication and retiming. Let N'' be a mapping solution of the combinational circuit of N', and N''' be the circuit obtained by connecting the removed FFs back to N'' and followed by another retiming and replication[1]. Then N''' is a *mapping solution* of N. The technology mapping problem addressed here is as follows:

Problem 1 *Find a mapping solution with minimum cycle time.*

Finally, we list several graph-theoretic concepts. Let G be a DAG with one sink but possibly several sources. We use $V(G)$ and $E(G)$ to denote the set of nodes and the set of edges of G,

[1] Later, we will see that this replication is unnecessary.

respectively. A *cut* $(X, \overline{X})$ is a partition of the nodes in G such that the sink is in $\overline{X}$ and all the sources are in X. The *edge-set* $E(X, \overline{X})$ of the cut is the set of edges from X to $\overline{X}$, the *node-set* $V(X, \overline{X})$ of the cut is the set of nodes in X which are connected to one or more nodes in $\overline{X}$. The *cone* of the cut is the subgraph induced by $\overline{X}$. If $|V(X, \overline{X})| \leq k$, $(X, \overline{X})$ is called a *k-feasible* cut, or k-cut for short.

4.2 Formation of LUTs

We recall that in a combinational circuit, a k-LUT for a node v is formed by the cone of a k-cut in N_v [21]. In this section, we are going to derive a similar result for sequential circuits.

In our problem formulation, a sequential circuit can conceptually be retimed and replicated arbitrarily before the FFs are removed. As a result, there is no fixed circuit for LUT formation. To overcome the difficulty, we actually study an equivalent problem in which only a subset of special mapping solutions is considered. Let N' be a mapping solution of N with minimum cycle time. Obviously the output functions of the LUTs in N' are from the set $\{f_v^{(i)} \mid v \text{ is a node in } N \text{ and } i \text{ is an integer}\}$. We define a retiming r of N' as follows: $r(\mathcal{L}) = -\rangle$ for a LUT $\mathcal{L}$ in N' with output function $f_v^{(i)}$. The output functions of the LUTs in the retimed circuit $N_r^{'}$ are then from the following set

$$\{f_v \mid v \text{ is a node in } N\}. \tag{4.1}$$

Namely, they are the signals in the original circuit N. By definition, $N_r^{'}$ itself is a mapping solution of N. We also have $\Phi_{opt}(N_r^{'}) = \Phi_{opt}(N') = \Phi(N')$. A mapping solution with the output functions of the LUTs from the set in (4.1) will be referred to as a *simple mapping solution*. By the above analysis, we have the following result:

Theorem 4.1 *There is a simple mapping solution whose optimal cycle time is equal to the minimum cycle time of all mapping solutions.* □

Because of the above result, instead of studying Problem 1 we will study the following problem:

Problem 2 *Find a simple mapping solution whose optimal cycle time is minimum.*

After finding a simple mapping solution with minimum optimal cycle time, we can compute an optimal retiming of the mapping solution and then retime the mapping solution according to the optimal retiming to obtain a mapping solution with minimum cycle time.

Since we now restrict ourselves to simple mapping solutions of N, we only need to study how to form LUTs for (the output functions of) the nodes in N. A LUT $\mathcal{L}$ for v (or its copies) in a simple mapping solution is obviously the cone of a cut in a circuit that can be obtained from N_v by replication and retiming. Of course, the edges inside the cone must have weight zero. Let $(X, \overline{X})$ be the cut. An edge $e' = (u', x)$ in edge-set $E(X, \overline{X})$ can be characterized by a 3-tuple (u, w, e), where u' is a copy of u in N, e' is a copy of e in N, and w is the weight of e'. (Note that the output function of u' is f_u since $\mathcal{L}$ is in a simple mapping solution.) The *characteristic set* $cs(\mathcal{L})$ is the set of all 3-tuples for the edges in $E(X, \overline{X})$.

With $cs(\mathcal{L})$ we can determine the number of inputs to LUT $\mathcal{L}$ easily. Suppose e'_1 and e'_2 are two edges in $E(X, \overline{X})$ and their respective 3-tuples are (u_1, w_1, e_1) and (u_2, w_2, e_2). If both $u_1 = u_2$ and $w_1 = w_2$, e'_1 and e'_2 generate the same input to $\mathcal{L}$. On the other hand, if $u_1 \neq u_2$ or $w_1 \neq w_2$ e'_1 and e'_2 generate different inputs to $\mathcal{L}$. Thus, the number of inputs to $\mathcal{L}$ is the number of 3-tuples whose first or second components are different.

To form a LUT for a node v, the straightforward approach is to use the cone of a cut in N_v. Then FFs within the cone are moved to the boundaries of the cone using retiming and logic replication. Beside the problem that the number of inputs to the final LUT is not directly related to the node-size of the the cut, this approach, however, may not generate all possible LUTs for v. For instance, for the circuit in Figure 4.4(a) if we replicate a (to become a_1 and a_2) and b (to become b_1 and b_2), then retiming b_2 by 1, we have the 3-LUT shown in Figure 4.4(c). This LUT is formed by the logic delineated by the dashed lines in Figure 4.4(b). Note that this LUT cannot be derived without replication first.

We now introduce the concept of *expanded circuits*. We will show that for a node v, it suffices to examine all k-LUTs formed by the k-cuts in the expanded circuit $\mathcal{E}(N_v)$ for v. $\mathcal{E}(N_v)$ is constructed by replicating the nodes in N_v and it has the property that all paths from any given node to v have the same number of FFs.

Let $r(u, v)$ denote the set $\{w(p) \mid p$ is a path from u to v in $N\}$. That is, $r(u, v)$ is the set of different path weights for all the paths from u to v in N. $\mathcal{E}(N_v)$ is obtained by successively replicating the nodes in N_v in the following fashion: The nodes in N_v are examined in a

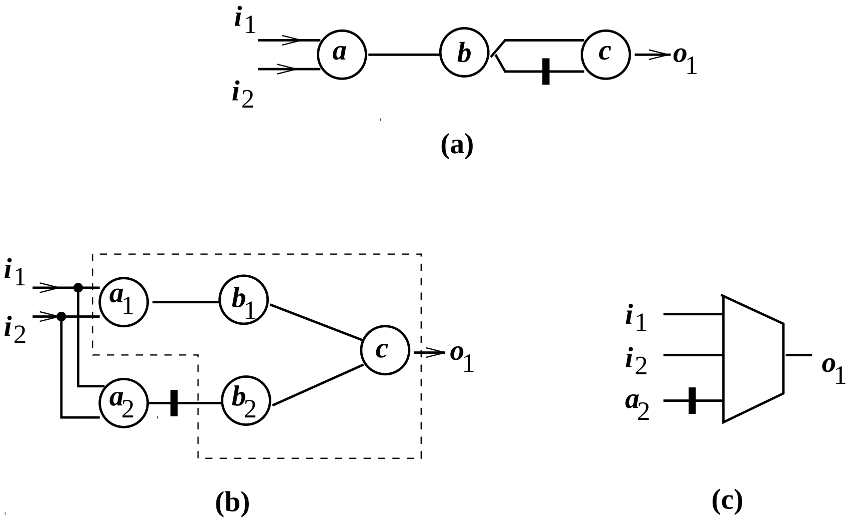


Figure 4.4: A LUT formed only through replication.

topological order starting from v. At the beginning, $N_v^{\{v\}}$ is the same as N_v except we rename v as $v^{(0)}$. Suppose we have constructed N_v^T and x is in T for each edge (u, x) in N_v, then $N_v^{T \cup \{u\}}$ is obtained from N_v^T by replicating u into $|r(u, v)|$ copies as follows:

(1) For each $w \in r(u, v)$, introduce a node $u^{(w)}$.

(2) For each edge $(u, x^{(w_2)})$ with weight w_1 in N_v^T, add an edge $(u^{(w)}, x^{(w_2)})$ with weight w_1 where $w = w_1 + w_2$.

(3) For each edge (y, u) with weight w_1 in N_v^T, add an edge $(y, u^{(w)})$ with weight w_1 for every $w \in r(u, v)$.

(4) Remove node u and the edges incident to it.

The following result can be easily shown by induction.

Lemma 4.1 *For any $u \in T$ and any $w \in r(u, v)$, the output function of $u^{(w)}$ is f_u, and any path from $u^{(w)}$ to $v^{(0)}$ has exactly w FFs.* □

When T consists of all the nodes in N_v, the resulting circuit is $\mathcal{E}(N_v)$. Note that when replicating a PI u, all copies of u are assumed to have the same output function f_u. As an example, for gate g in the circuit in Figure 4.2(a), Figure 4.5 shows its expanded circuit. The following corollary of Lemma 4.1 shows that $\mathcal{E}(N_v)$ indeed has the desired property.

Lemma 4.2 *For each $u^{(w)}$ in $\mathcal{E}(N_v)$, its output function is f_u, and any path from $u^{(w)}$ to $v^{(0)}$ has exactly w FFs.* $\square$

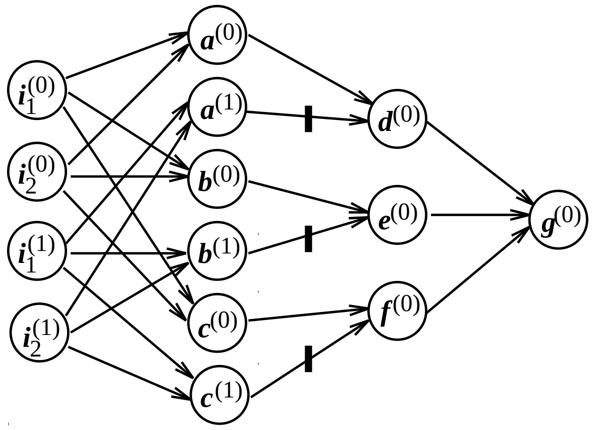


Figure 4.5: Expanded circuit.

A more direct way to view $\mathcal{E}(N_v)$ is that for each path from u to v in N_v:

$$u = u_1 e_1 u_2 e_2 \cdots e_t u_{t+1} = v,$$

there is a unique corresponding path in $\mathcal{E}(N_v)$:

$$u_1^{(w_1)} e_1' u_2^{(w_2)} e_2' \cdots e_t' u_{t+1}^{(0)},$$

where $w_i = \sum_{i \leq j \leq t} w(e_j)$, and e_i' is a copy of e_i with its weight being $w(e_i)$, $1 \leq i \leq t$.

We now estimate the sizes of $\mathcal{E}(N_v)$. Hereafter, we will use n to denote $|V(N)|$, the number of nodes in N. Since N is k-bounded, the number of edges in N is $O(kn)$. If we let R denote the maximum of $|r(u,v)|$ for all pairs of u and v, obviously $|V(\mathcal{E}(N_v))| \leq R|V(N)|$ and $|E(\mathcal{E}(N_v))| \leq R|E(N)|$. In practice, the weight of an edge is usually either 0 or 1. Consequently, the maximum element in $r(u,v)$ is less than n. This means $R \leq n$. Therefore, $|V(\mathcal{E}(N_v))| = O(n^2)$ and $|V(\mathcal{E}(N_v))| = O(kn^2)$.

Let $(X, \overline{X})$ be a k-cut in $\mathcal{E}(N_v)$. For each $u^{(w)}$ in $\overline{X}$, we retime it by w; all other nodes are assigned a retiming value zero. Of course, $(X, \overline{X})$ is still a cut in the retimed circuit. From Lemma 4.2, we know that the edges between the nodes in $\overline{X}$ have weight zero, i.e., no FFs. As a result, the cone of $(X, \overline{X})$ is a LUT for v. Let $\mathcal{L}$ denote the LUT. Suppose $e' = (u_i^{(w_i)}, u_{i+1}^{(w_{i+1})})$ in $E(X, \overline{X})$ is a copy of e in N. In the retimed circuit, the weight of e' is $w(e) + w_{i+1} = w_i$. Therefore, the 3-tuple for e' is (u_i, w_i, e). Consequently,

$$cs(\mathcal{L}) = \{(u_i, w_i, e) \mid (u_i^{(w_i)}, x) \text{ in } E(X, \overline{X}) \text{ is a copy of } e \text{ in } N\}. \tag{4.2}$$

Thus, the number of inputs to $\mathcal{L}$ is equal to $|V(X, \overline{X})|$. Consequently, $\mathcal{L}$ is a k-LUT.

Lemma 4.3 *From a k-cut $(X, \overline{X})$ in $\mathcal{E}(N_v)$, we can derive a k-LUT $\mathcal{L}$ for v satisfying* (4.2). $\square$

On the other hand, let $\mathcal{L}$ be a k-LUT for v. We remove each edge $e' = (u_i^{(w_i)}, x)$ from $\mathcal{E}(N_v)$ if tuple (u_i, w_i, e) is in $cs(\mathcal{L})$, where e' is a copy of e. Then all paths from PIs to $v^{(0)}$ in $\mathcal{E}(N_v)$ must be broken. Let $\overline{X}$ be the set of nodes that have paths to $v^{(0)}$ after the edge removal. $(X, \overline{X})$ then is a cut in $\mathcal{E}(N_v)$. Furthermore, $V(X, \overline{X}) \subseteq \{u_i^{(w_i)} \mid (u_i, w_i, e) \in cs(\mathcal{L})\}$. Let $\mathcal{L}'$ be the k-LUT derived from cut $(X, \overline{X})$. From Lemma 4.3, we have $cs(\mathcal{L}') \subseteq cs(\mathcal{L})$.

Lemma 4.4 *Given a k-LUT $\mathcal{L}$ for v, there is a k-cut $(X, \overline{X})$ in $\mathcal{E}(N_v)$ such that the k-LUT $\mathcal{L}'$ for v derived from the k-cut satisfying $cs(\mathcal{L}') \subseteq cs(\mathcal{L})$.* $\square$

We will see that to solve Problem 2, we only need to consider those k-LUTs whose characteristic sets are minimal under the subset relation. Therefore, when examining k-LUTs for v, we only need to consider the k-LUTs that can be derived from the k-cuts in $\mathcal{E}(N_v)$.

4.3 An optimal cycle time mapping algorithm

In this section we present our algorithm for solving Problem 2, assuming the unit delay model, namely, PEs have one unit delay and interconnections have zero delay. As was mentioned earlier, we assume that the initial circuit N is loopless and k-bounded. The algorithm is based on solving the following problem, which is the decision version of Problem 2:

Problem 3 *Given a target cycle time c, find a simple mapping solution whose optimal cycle time is c or less, if such a mapping solution exists.*

If we have an algorithm for solving Problem 3, we can use binary search to solve Problem 2. In the remainder of this section, we are going to present an algorithm for solving Problem 3. Our algorithm employs a labeling procedure and is based on network flow techniques.

We first consider the problem of determining, for a given circuit M and an integer c, whether M can be retimed to cycle time c or less. To this end, we construct a graph whose topology is the same as that of M, but with the edge weights being redefined as follows: For an edge $e = (u, v)$, the new weight is $-c \cdot w(e) + 1$ if v is a PE; otherwise, the new weight is $-c \cdot w(e)$. The *l-value* $l(v)$ of v is defined to be the maximum weight of the paths from the PIs to v according to the new edge weights. We have,

Theorem 4.2 *$\Phi_{opt}(M) \leq c$ iff $l(v) \leq c$ for each PO v.*

Proof: Let G denote the reweighted graph and $w_G(e)$ denote the new edge weight of e.

($\Longrightarrow$) Suppose there is a PO v such that $l(v) > c$. Let p be a path from PI u to v with $l(v) = w_G(p)$. Obviously, $w_G(p) = -c \cdot w(p) + d(p)$. Thus, $-c \cdot w(p) + d(p) > c$, so $d(p) > (w(p) + 1)c$. To have a cycle time c on p, we have to have $\lceil \frac{d(p)}{c} \rceil - 1 \geq w(p) + 1$ FFs on p by pigeonhole principle. This is obviously impossible since retiming cannot increase the number of FFs on any path from a PI to a PO.

($\Longleftarrow$) Let r be the following retiming:

$$r(v) = \begin{cases} 0 & v \text{ is a PI or PO} \\ \lceil \frac{l(v)}{c} \rceil - 1 & \text{otherwise.} \end{cases}$$

We are going to show that M_r has cycle time c or less.

First, we show that r is a legal retiming. Let $e = (u, v)$ be an edge in M. We have $l(v) \geq l(u) + w_G(e)$. There are four cases according to whether u and v are PEs, POs, or PIs. All the case can be proved similarly. Here we only show the case that u is a PE and v is a PO. In this case, we have $c \geq l(v) \geq l(u) + w_G(e) = l(u) - c \cdot w(e)$. Dividing both sides by c, we have $1 \geq \frac{l(u)}{c} - w(e)$. Therefore,

$$r(v) = 0 \geq r(u) - w(e), \text{ i.e., } w(e) + r(v) - r(u) \geq 0.$$

Namely, e has non-negative weight in M_r.

To show M_r has cycle time c or less, we only need to show that for any path p such that $d(p) \geq c+1$, there is at least one FF on it in M_r. Obviously, we can assume that both end-points of p are PEs. We then have,

$$l(v) \geq l(u) + w_G(p) = l(u) - c \cdot w(p) + d(p) - 1.$$

Therefore,

$$r(v) \geq r(u) - w(p) + \frac{d(p) - 1}{c} \geq r(u) - w(p) + 1,$$

so, $w(p) + r(v) - r(u) \geq 1$. Namely, there is at least one FF on p in M_r. □

Remark: It is clear that replicating a circuit does not change the l-values of the POs. From Theorem 4.2, we have that replication cannot change the optimal cycle time of a circuit. Consequently, we can replace the second "replication & retiming" by "retiming" in Figure 4.3.

For a loopless circuit, the l-values can be determined by a topological traversal of the circuit. Let $v_1, v_2, \cdots, v_t$ be a topological ordering of the nodes in the circuit starting from the PIs. We can determine the l-values of the nodes recursively as follows:

$$\begin{array}{ll} l(v_i) = 0 & v_i \text{ is a PI;} \\ l(v_i) = \max_{e=(u,v_i)}\{l(u) - c \cdot w(e) + 1\} & v_i \text{ is a PE;} \\ l(v_i) = l(u) - c \cdot w(e) & e = (u, v_i) \text{ is the edge to PO } v_i. \end{array}$$

Our algorithm for solving Problem 3 has two phases: the labeling phase and the mapping phase. In the labeling phase, we compute the label for each node which is the minimum l-value of the k-LUTs for the node among all possible mapping solutions. With the minimum l-values, we can determine whether there is a mapping solution whose optimal cycle time is c or less according to Theorem 4.2. If the mapping solution exists, we generate one such solution in the mapping phase, using the information produced during the labeling phase. In the next two subsections, we present the details of the two phases, respectively.

4.3.1 The labeling phase

For a node v in N, let $l^{opt}(v)$ denote the minimum l-value of the k-LUTs for v among all mapping solutions. We determine $l^{opt}(v)$ for each node v in N in this phase of the algorithm.

Suppose $\mathcal{L}$ is a k-LUT for v. The l-value of $\mathcal{L}$ then is:

$$\max\{l(u) - c \cdot w + 1 \mid (u, w, e) \in cs(\mathcal{L})\},$$

where $l(u)$ is the l-value for u. To minimize the l-value for v, we let $l(u) = l^{opt}(u)$. Moreover, we can assume that $cs(\mathcal{L})$ is minimal under the subset relation. By the analysis in Section 4.2, we have

$$l^{opt}(v) = \min_{(X,\overline{X})} \left(\max\{l^{opt}(u) - c \cdot w + 1 \mid (u^{(w)}, x, e') \in E(X, \overline{X})\} \right), \qquad (4.3)$$

where $(X, \overline{X})$ is a k-cut in $\mathcal{E}(N_v)$.

From (4.3) it is obvious that we can compute the minimum l-values for the nodes in N in topological order starting from the PIs. At the beginning, for each PI v, $l^{opt}(v) = 0$. Suppose we now want to determine $l^{opt}(v)$ for a non-PI node v, given that the minimum l-values for its predecessors have already been determined. To determine $l^{opt}(v)$, we again consider the decision version, namely the following problem:

Problem 4 *Determine whether $l^{opt}(v) \leq L$ for a given integer L.*

We use network flow techniques to determine whether $l^{opt}(v) \leq L$. The flow network G is constructed from $\mathcal{E}(N_v)$ as follows:

(1) For each node $u^{(w)}$ in $\mathcal{E}(N_v)$ except $v^{(0)}$, we introduce two nodes $u_h^{(w)}$ (*head*) and $u_t^{(w)}$ (*tail*) in G, and connect them (from the tail to the head) with a *bridging edge*. For convenience, we view $v^{(0)}$ as both $v_h^{(0)}$ and $v_t^{(0)}$.

(2) For each edge $(x^{(w_1)}, y^{(w_2)})$ where $y \neq v$, there is an edge $(x_h^{(w_1)}, y_t^{(w_2)})$ in G.

(3) A super-source s is introduced and is connected to $u_t^{(w)}$ for each PI u and each w in $r(u, v)$.

(4) The capacities on the edges in G are assigned as follows: If $l^{opt}(u) - c \cdot w + 1 \leq L$, the bridging edge for $u^{(w)}$ has unit capacity. All other edges have infinite capacity.

The capacity of a cut in a flow network is the sum of the capacities of the edges in the edge-set of the cut. Suppose G has a cut $(X', \overline{X'})$ with capacity less than or equal to k. Let

$\overline{X}$ be the nodes in $\mathcal{E}(N_v)$ whose heads and tails are all in $\overline{X'}$. It is easy to see that $(X, \overline{X})$ is a k-cut in $\mathcal{E}(N_v)$ such that $l^{opt}(u) - c \cdot w + 1 \leq L$ for each $(u^{(w)}, x, e')$ in $E(X, \overline{X})$. Therefore, we have $l^{opt}(v) \leq L$. The arguments hold in the reverse direction.

Lemma 4.5 *$l^{opt}(v) \leq L$ iff G has a cut with capacity less than or equal to k.* □

Based on the classical Max-flow Min-cut Theorem, we can use the augmenting path algorithm for solving the max-flow problem to determine whether G has a cut with capacity less than or equal to k in $O(k \cdot |E(G)|) = O(k^2 n^2)$ time. Though $l^{opt}(v)$ can now be determined using binary search based on Lemma 4.5, we actually only need to solve Problem 4 once. This is because the following result similar to Lemma 2 in [21]:

Lemma 4.6 *$l^{opt}(v) = L_v - 1$ or L_v, where*
$L_v = \max\{l^{opt}(u) - c \cdot w(e) + 1 \mid e = (u, v)$ is an edge in $N_v\}$.

Proof: Since N is k-bounded, there are at most k nodes connecting to $v^{(0)}$ in $\mathcal{E}(N_v)$. Let $\overline{X} = \{v^{(0)}\}$, then $(X, \overline{X})$ is a k-cut in $\mathcal{E}(N_v)$. The l-value for v resulted from this k-cut is exactly L_v. Therefore $l^{opt}(v) \leq L_v$. On the other hand, suppose $e = (u, v)$ is an edge in N_v. Then $(u^{(w_1)}, v^{(0)})$ is an edge in $\mathcal{E}(N_v)$ where $w_1 = w(e)$. Let $(X, \overline{X})$ denote the k-cut such that the resulting l-value for v is minimum. If $u^{(w_1)}$ is in X, then we have $(u^{(w_1)}, v^{(0)}) \in E(X, \overline{X})$. Consequently, $l^{opt}(v) \geq l^{opt}(u) - c \cdot w_1 + 1$. If $u^{(w_1)}$ is in $\overline{X}$, let $\overline{X'}$ (X') be the set of nodes in $\overline{X}$ (X) which have a path to $u^{(w_1)}$ and $\overline{X''} = \{x^{(w-w_1)} \mid x^{(w)} \in \overline{X'}\}$ $(X'' = \{x^{(w-w_1)} \mid x^{(w)} \in X'\})$, then it can be verified that $(X'', \overline{X''})$ is a k-cut in $\mathcal{E}(N_u)$. Therefore,

$$
\begin{aligned}
l^{opt}(u) &\leq \max\{l^{opt}(x) - c \cdot (w - w_1) + 1 \mid (x, w - w_1, e') \in E(X'', \overline{X''})\} \\
&= \max\{l^{opt}(x) - c \cdot w + 1 \mid (x, w, e') \in E(X', \overline{X'})\} + c \cdot w_1 \\
&\leq \max\{l^{opt}(x) - c \cdot w + 1 \mid (x, w, e') \in E(X, \overline{X})\} + c \cdot w_1 \\
&= l^{opt}(v) + c \cdot w_1,
\end{aligned}
$$

Thus, $l^{opt}(v) \geq l^{opt}(u) - c \cdot w_1$. In any case, $l^{opt}(v) \geq l^{opt}(u) - c \cdot w_1$ for any u such that (u, v) is in N_v. Therefore, $l^{opt}(v) \geq L_v - 1$. □

To determine $l^{opt}(v)$, we only need to check whether $l^{opt}(v) \leq L_v - 1$. If so, $l^{opt}(v) = L_v - 1$; otherwise, $l^{opt}(v) = L_v$. Based on these discussions, we have the following result:

Theorem 4.3 *The minimum l-values of all nodes in N can be determined in $O(k^2 n^3)$ time.* □

4.3.2 The mapping phase

The purpose of this phase is to generate a mapping solution with cycle time c or less if there is one. Note that after obtaining the minimum l-values for the POs in N, we can check whether there is a mapping solution whose optimal cycle time is c or less easily according to Theorem 4.2.

This phase is relatively simple. The first step is to generate a simple mapping solution whose optimal cycle time is c or less. To do this, we keep two lists D and U. D is the set of nodes in N for which we have generated k-LUTs and U is the set of nodes whose k-LUTs are inputs to some k-LUTs in D and have not yet been generated. At the beginning, D consists of the PIs and U consists of the POs. At each iteration, a node v in U is removed and added to D. Let the k-LUT that realizes $l^{opt}(v)$ be $\mathcal{L}_v$ as was determined in the labeling phase. Then, for each (u, w, e) in $cs(\mathcal{L}_v)$ we create an edge from $\mathcal{L}_u$ to $\mathcal{L}_v$ with weight w, and add u to U if it is not in D or U. This process stops when U becomes empty. Let N' denote the resulting mapping solution. Note that for each node v in N there is at most one $\mathcal{L}_v$ in N'. In other words, replication is only used in the formation of the LUTs. The final D could be a proper subset of $V(N)$. For those nodes not in the final D, their outputs are not used by any other k-LUTs in N'.

To obtain a mapping solution with cycle time c or less, we can compute an optimal retiming of N' and apply the retiming to N'. However, from the proof of Theorem 4.2 we can simply apply the following retiming to N':

$$r(\mathcal{L}_v) = \begin{cases} 0 & v \text{ is a PI or PO} \\ \lceil \frac{l^{opt}(v)}{c} \rceil - 1 & \text{otherwise.} \end{cases}$$

The retimed solution is guaranteed to have cycle time c or less, provided that the l-value for each PO is less than or equal to c.

4.4 Extension to the general delay model

FPGA technology mapping under the general delay model was again only studied for combinational circuits. Mathur and Liu [55] proposed a very effective heuristic algorithm. Later, Yang and Wong [90] gave an optimal algorithm. Here, we extend our algorithm in the previous section to the general delay model. The extended algorithm can produce a mapping solution with provably good cycle time for loopless sequential circuits.

In the general delay model, both PEs and interconnections have associated delay values. We will use $d(x)$, where x is either a PE or an edge, to denote the delay of x. Without loss of generality, we assume that the delay of an edge cannot be further divided. In other words, FFs can only be placed right after or right before a PE. For otherwise, we can certainly divide any dividable edge by introducing dummy nodes beforehand.

The delays in a mapping solution come from two sources: the delays of the k-LUTs and the delays of the interconnections. We assume that all k-LUTs have the same delay and the delay value will be denoted $d_{\square}$. For an edge with w_1 FFs from LUT $\mathcal{L}_u$ to LUT $\mathcal{L}_v$ in a mapping solution, its delay is assumed to be the maximum of the delays of all the edges in the initial circuit that get mapped to this edge, namely, $\max\{d(e) \mid (u, w_1, e') \in cs(\mathcal{L}_v)\}$.

The algorithm is based on the generalized l-values which are defined in the same way as the l-values for the unit delay model except the following change: For an edge $e = (u, v)$ in the graph that has the same topology as circuit M, its weight is $-c \cdot w(e) + d(e) + d(v)$ instead. The generalized l-value $l(v)$ of v is again defined to be the maximum weight of the paths from the PIs to v according to the new edge weights. The following result, similar to Theorem 4.2, is the basis of the algorithm and it can also be proved similarly:

Theorem 4.4 *(i) If there is a PO v such that $l(v) > c$, then $\Phi_{opt}(M) > c$; (ii) if $l(v) \le c$ for each PO v, $\Phi_{opt}(M) < c + \max\{d(x) \mid x$ is a node or an edge in $M\}$.* □

The algorithm for the general delay model is essentially the same as the one for unit delay model except that we use the generalized l-values. Namely, Eq. (3) is replaced by the following one:

$$l^{opt}(v) = \min_{(X, \overline{X})} \left(\max\{l^{opt}(u) - c \cdot w + d(e) + d_{\square} \mid (u^{(w)}, x, e') \in E(X, \overline{X})\} \right), \tag{4.4}$$

where $(X, \overline{X})$ is a k-cut in $\mathcal{E}(N_v)$.

To determine $l^{opt}(v)$, we, again, consider its decision version, namely, determining whether or not $l^{opt}(v) \le L$ for a given integer L. The decision version can be determined by the network flow techniques using a flow network which is a slight modification of the one for unit delay model, as proposed by Yang and Wong [90]. Specifically, the modification is as follows: Change (4) in the construction of the previous network to be that all bridging edges have unit capacity and all other edges have infinite capacity, and in (2), replace $x_h^{(w_1)}$ by $x_t^{(w_1)}$ if $l^{opt}(u) - c \cdot w + d(e) + d_\Box > L$ and $(x^{(w_1)}, y^{(w_2)})$ is a copy of edge e in N. Analogous to Theorem 4.3, we have the following result:

Theorem 4.5 *The minimum l-values of all nodes in N can be determined in $O(k^2 n^3 \log(kn))$ time.* $\Box$

After all the minimum l-values are determined, if one or more of the POs have l-values larger than c, it is certain that there is no mapping solution with cycle time c or less. On the other hand, if the l-value for all the POs are less than or equal to c, the mapping solution obtained during the mapping phase is guaranteed to have cycle time less than $c + \delta$, where δ is the maximum among $d_\Box$ and the delays of the edges in the mapping solution.

To obtain a mapping solution with cycle time not far from the optimal cycle time (denoted ϕ_{opt}), we can use binary search to search for a real number c_ϵ (for any given $\epsilon > 0$) with the following property: At least one of the POs will have a l-value larger than $c_\epsilon - \epsilon$ when the target cycle time is $c_\epsilon - \epsilon$, but all the POs have l-values less than or equal to c_ϵ if c_ϵ is the target cycle time. From Theorem 4.4, we know the mapping solution obtained using c_ϵ as the target cycle time then has cycle time less than $\phi_{opt} + \epsilon + \delta$. By making ϵ arbitrarily small, we can obtain a mapping solution whose cycle time is at most δ more than the optimal cycle time.

4.5 Experimental results

We implemented our technology mapping algorithm (referred to as SeqMap) and carried out some experiments. In this section, we describe our experiments and summarize the results.

Since there are no benchmark loopless sequential circuits, we derived our test examples from the multi-level combinational and sequential benchmark circuits in the LGSynth91 suite from MCNC. For a combinational benchmark circuit, we added to it one stage of pipeline (adding one FF at each PI). The resulting sequential circuit was then retimed to its optimal cycle time to obtain a test example. For a sequential benchmark circuit, we removed a set of gates to cut the loops, then again retimed it to its optimal cycle time to obtain a test example. Some standard **SIS** commands such as *speed_up, sweep*, and *tech_decomp* were used in the construction of the test examples.

For comparison, we implemented a technology mapping algorithm based on the conventional approach called ComMap. ComMap maps a sequential circuit by first removing all the FFs. Then, it maps the resulting combinational logic using FlowMap — a delay optimal technology mapping algorithm for combinational circuits by Cong and Ding [21]. Finally, the removed FFs are connected back to form a mapping solution of the original circuit. To make the comparison fair, ComMap further retimes the resulting mapping solution to its optimal cycle time. The resulting circuit is then the final output of ComMap. In our current implementation, neither ComMap nor SeqMap contains any post-processing operations for LUT reduction.

test	**Initial**			**ComMap**			**SeqMap**		
example	#gates	#FFs	CT	#LUTs	#FFs	CT	#LUTs	#FFs	CT
C3540	1078	56	19	477	53	6	471	58	5
C2670	996	173	12	386	171	4	364	157	4
cm85a	48	8	4	15	7	2	13	10	1
comp	134	12	6	56	10	3	52	14	2
count	264	53	6	140	29	3	119	40	2
cu	62	12	4	26	13	2	23	17	1
f51m	226	22	5	61	13	3	55	17	2
i10	2774	330	21	1348	308	7	1287	341	6
pm1	73	24	3	36	18	2	28	23	1
term1	735	88	8	306	45	4	275	54	3
x2	59	13	4	23	11	2	17	15	1
mult32a	533	33	6	68	33	2	66	34	1
s1196	671	19	16	306	19	5	309	22	5
s5378	2779	430	21	747	411	5	572	309	4
s9234	5597	498	27	1164	483	6	911	338	5
s15850	9785	944	54	2157	944	7	1618	694	7
total				7316		63	6180		50

Table 4.1: Experimental results.

We tested both ComMap and SeqMap on a set of example circuits using 5-LUTs. The results are summarized in Table 4.1. In Table 4.1, under column **initial** we list the number of gates, the number of FFs, and the cycle time (CT) of each test example; under column **ComMap** (resp. **SeqMap**), we list the number of LUTs, the number of FFs, and the cycle time of the mapping solution produced by ComMap (resp. SeqMap) for each example. Note that the cycle times of the mapping solutions produced by SeqMap are optimal. From the table, it is clear that quite often ComMap produced sub-optimal solutions. It can also be seen that the mapping solutions produced by SeqMap usually have fewer numbers of LUTs than that produced by ComMap. This was also expected since SeqMap may extend across FF boundaries to form LUTs, while the signal dependencies across FF boundaries are totally ignored by ComMap. The running time of SeqMap was about 2-3 times that of ComMap for each of the test examples.

4.6 Summary

In this chapter, we have studied the FPGA technology mapping problem for sequential circuits. The problem is studied in the most general setting, except we do not use the functionality information of the gates. Under our formulation, the mapping solution space is much larger than what the conventional approach (based on a mapping algorithm for combinational circuits) is able to explore.

We have presented an optimal cycle time mapping algorithm for loopless sequential circuits. (This algorithm can also be used for circuits with loops by properly removing some of the FFs to break the loops first.) Another way to understand our algorithm is that it combines retiming and replication with mapping combinational logic globally, while in the conventional approach, they are carried out in sequence (refer to Figure 4.3). Our algorithm has been implemented. Experimental results further show the superiority of our algorithm over the conventional one.

Currently, we are trying to extend our results to sequential circuits with loops. We are also considering using the ideas developed here for LUT minimization.

Chapter 5

Area Minimization for General Floorplans

Floorplanning is an important stage in the physical design of VLSI circuits. Because of its complexity, floorplanning is usually carried out in two separate steps. The first step is to determine the floorplan that specifies the relative positions of the subcircuits on a chip. Since each subcircuit may have several possible layout alternatives with different dimensions, the second step is to select a layout alternative for each subcircuit so that the total chip area is minimized while the floorplan determined in the first step is held fixed. In this chapter, we study the second step which is referred to as the area minimization problem. There is an extensive literature on the area minimization problem [19, 47, 62, 68, 78, 83, 85, 87, 93, 94]. Among the many results, Stockmeyer showed that area minimization for general floorplans is strongly NP-complete [78].

Area minimization algorithms can be divided into two classes, those that can handle only a special class of floorplans [62, 68, 78, 83], and those that can handle general floorplans [19, 85, 87]. On the one end, there is an efficient polynomial algorithm for slicing floorplans by Otten and Stockmeyer [62, 78]. On the other end, Wimer *et al.* proposed an algorithm for general floorplans based on the branch-and-bound method [85]. In this chapter we are interested in area minimization for general floorplans. Area minimization for general floorplans is important for two major reasons. First, general floorplans are often encountered in practice. Second, a key step in area minimization for hierarchical floorplans is to determine the nonredundant realizations of the non-slicing primitive superblocks (or partitioning patterns) [19, 68, 83].

We propose two area minimization methods for general floorplans. Both methods are based on a generalization of the concept of cutlines. Using the generalized cutlines, we are able to partition a floorplan even when it is not slicing. Unlike the branch-and-bound based algorithm in [85], our methods explicitly exploit the structural information of the floorplans. In particular, our methods reduce naturally to the polynomial algorithm for slicing floorplans when the given floorplans are slicing. In general, for floorplans that are "approximately" slicing, our methods are quite efficient. Such a property makes our methods particularly attractive since non-slicing floorplans often only "deviate" slightly from slicing floorplans in practice. (We shall further explain this point later.)

5.1 Preliminaries and problem definition

A *floorplan* is a dissection of an enveloping rectangle into rectangular (*basic*) *blocks* such that no four blocks share a common point, i.e., there is no + - junction as illustrated in Figure 5.1. The enveloping rectangle represents the total chip area while the blocks represent areas for the subcircuits. A floorplan specifies the topological relationship (left-right and above-below) among the subcircuits.

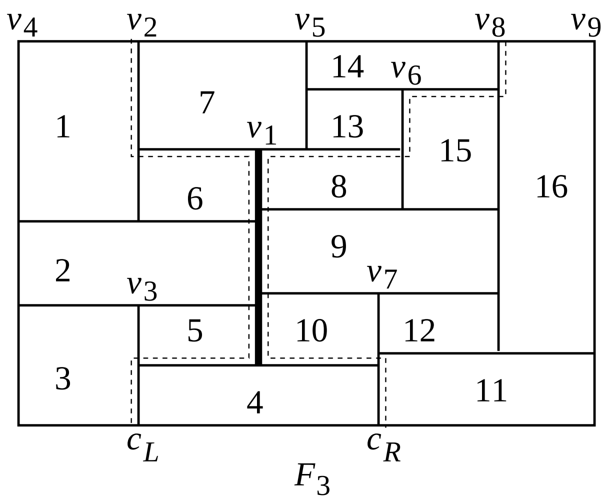


Figure 5.1: A floorplan.

A layout alternative of a subcircuit is called a *realization* of the corresponding block. A *realization* of a floorplan is obtained by selecting a realization for each block and arrange them according to the floorplan. In this chapter we assume that the shapes of blocks and floorplans are all rectangular. A realization (of a block or a floorplan), therefore, has two dimensions, the width and height of the smallest rectangle that can accommodate the realization. The area minimization problem can be formally stated as follows: *Given a floorplan and a set of realizations for each of its blocks, determine a realization of the floorplan which has the minimum area.*

Let r be a realization (of a block or a floorplan), we use $w(r)$ to denote its width and $h(r)$ to denote its height. For two different realizations r_1 and r_2, r_1 is said to *dominate* r_2 if the conditions $w(r_1) \leq w(r_2)$ and $h(r_1) \leq h(r_2)$ are satisfied. (Note that two realizations having the same width and height are considered to be the same during area minimization.) For a set of realizations, a realization in the set is said to be *redundant* if it is dominated by another realization in the set. Otherwise, the realization is said to be *nonredundant*. A set of nonredundant realizations obviously has the following properties: (i) No two realizations in the set have the same width or height, and (ii) If the realizations are sorted in increasing order according to one dimension, they will be arranged in decreasing order according to the other dimension. These properties suggest the following simple algorithm for determining the nonredundant realizations of a set of given realizations: Sort all realizations in the set in increasing order according to the width (if two realizations have the same width, remove the one with larger height), then inspect the realizations in this order one by one and retain only those which keep the height in decreasing order.

It is obvious that as far as a minimum area realization of a floorplan is concerned, redundant realizations of blocks and the floorplan are irrelevant. Thus, the problem we actually consider in this chapter is a more general one: *Given a floorplan and a set of nonredundant realizations for each block, find the set of nonredundant realizations of the floorplan.* Since the set of nonredundant realizations of a floorplan contains all the minimum area realizations, determination of this set also gives the designer the additional freedom to choose, among all the minimum area realizations of the floorplan, the one with a preferred aspect ratio. More importantly, finding the set of nonredundant realizations instead of only a minimum area realization is essential for the application of the methods to hierarchical area minimization [19, 68, 83].

A *V-channel* (resp. *H-channel*) is a maximal vertical line segment (resp. horizontal) in a floorplan. For example, v_1 (thickened) in Figure 5.1 is a V-channel. A very useful representation of a floorplan is a pair of polar graphs (directed acyclic graphs with a source and a sink), called the *V-graph* and *H-graph* [47, 63]. The vertices of the V-graph (resp. H-graph) are the V-channels (resp. H-channels) in the floorplan. Corresponding to each block there is an arc connecting the two V-channels (resp. H-channels) that contain the two opposite vertical (resp. horizontal) sides of the block in the V-graph (resp. H-graph). This arc will be labeled with the name of the block. As a convention, we choose the orientations of the arcs to be from left to right in the V-graph, and from top to bottom in the H-graph. Figure 5.2 shows the two polar graphs of the floorplan in Figure 5.1. It is well known that both V-graph and H-graph of a floorplan can be embedded in the plane with no crossing arcs such that their sources and sinks are in the exterior (unbounded) face. Furthermore, they are duals of each other, and each pair of dual arcs have the same label [9, 47, 63]. Two polar graphs having this relationship are called dual polar graphs. If D is a polar graph, we shall use D^* to denote the dual of D.

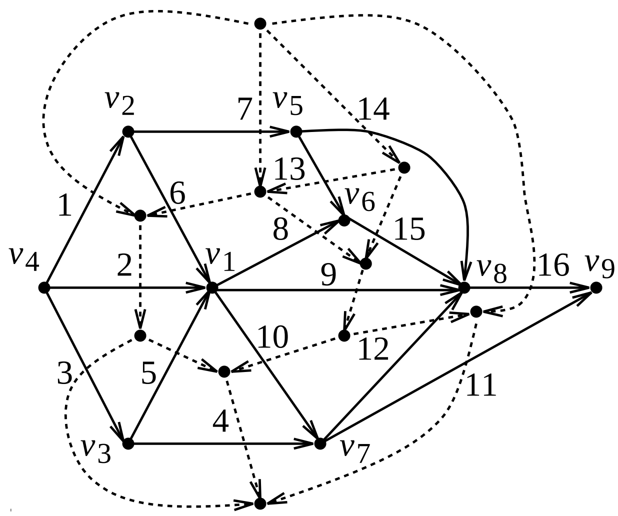


Figure 5.2: V-graph (solid) and H-graph (dashed).

A *selection* for a set of blocks is a function that selects a realization for each block in the set. Let σ be a selection for a set $\mathcal{B}$. For a block B in $\mathcal{B}$, $\sigma(B) = (w, h)$ means that σ selects realization (w, h) for block B. Let σ_1 be a selection for $\mathcal{B}_1$ and σ_2 be a selection for $\mathcal{B}_2$ such that $\mathcal{B}_1 \cap \mathcal{B}_2 = \emptyset$, $\sigma_1 \vee \sigma_2$ denotes the combined selection for $\mathcal{B}_1 \cup \mathcal{B}_2$. A realization of a floorplan is simply a selection for the set of all the blocks in the floorplan. The dimensions of a realization can be determined as follows: A weight equal to the width (height) of the realization selected for a block is assigned to its corresponding arc in the V-graph (H-graph). The width (height) of the realization is the length of a longest path from the source to the sink in the V-graph (H-graph). In general, we use $l_D^\sigma(p)$ to denote the length of path p in graph D, and $l_D^\sigma(u_1, u_2)$, where u_1 and u_2 are vertices of D, to denote the length of a longest path from u_1 to u_2, with weights assigned to the arcs according to selection σ.

Let D be a polar graph with source s and sink t. We use $V(D)$ to denote the vertex set and $A(D)$ to denote the arc set of D. A *full path* in a polar graph is a directed path from the source to the sink. For example, {1, 6, 8, 15, 16} is a full path in the V-graph in Figure 5.2. A set of arcs, E, in D is called a *cut* if there is a partition $\{S, \overline{S}\}$ on $V(D)$ such that $s \in S$, $t \in \overline{S}$, E is the set of arcs from the vertices in S to the vertices in $\overline{S}$, and there is no arc from a vertex in $\overline{S}$ to a vertex in S. For example, {7, 6, 2, 5, 4} is a cut in the V-graph in Figure 5.2 with $S = \{v_2, v_3, v_4\}$.

We have the following results concerning full paths and cuts in the pair of dual polar graphs D and D^* of a floorplan:

Lemma 5.1 *Let $E \subseteq A(D)$. Then E is a full path in D iff E is a cut in D^*.*

Proof: A direct consequence of the duality relationship between D and D^*. □

Lemma 5.2 *If $e_1, e_2 \in A(D)$ and there is a path that contains e_1 and e_2, then there is no cut that contains e_1 and e_2.*

Proof: A direct consequence of the definition of a cut. □

By *coalescing* [51] a set of vertices of an arc-labeled directed graph we mean to delete all arcs among the vertices, identify them as a single vertex, and let the resulting vertex inherit all the incoming and outgoing arcs (retaining their labels) of the vertices in the set.

A vertex v in D induces a partition of D into three arc-disjoint subgraphs: $L_{v,D}$, the subgraph induced by the arcs on the directed paths from s to v; $R_{v,D}$, the subgraph induced by the arcs on the directed paths from v to t; $C_{v,D}$, the subgraph induced by the remaining arcs. Let $LC = V(L_{v,D}) \cap V(C_{v,D}) - \{s, t\}$, $RC = V(R_{v,D}) \cap V(C_{v,D}) - \{s, t\}$ when D and v are understood from the context. We further define two polar graphs $D_L(v)$ and $D_R(v)$.

$D_L(v)$ (resp. $D_R(v)$) is obtained from D by coalescing all vertices in $R_{v,D}$ (resp. $L_{v,D}$). If the resulting vertices are denoted v in both graphs, then $D_L(v)$ has source s and sink v and $D_R(v)$ has source v and sink t. As an example, for the V-graph (called D) in Figure 5.2, the three subgraphs induced by v_1 are shown in Figure 5.3. Figure 5.4 shows $D_L(v_1)$ and $D_R(v_1)$. In addition, $LC = \{v_2, v_3\}$ and $RC = \{v_6, v_7, v_8\}$.

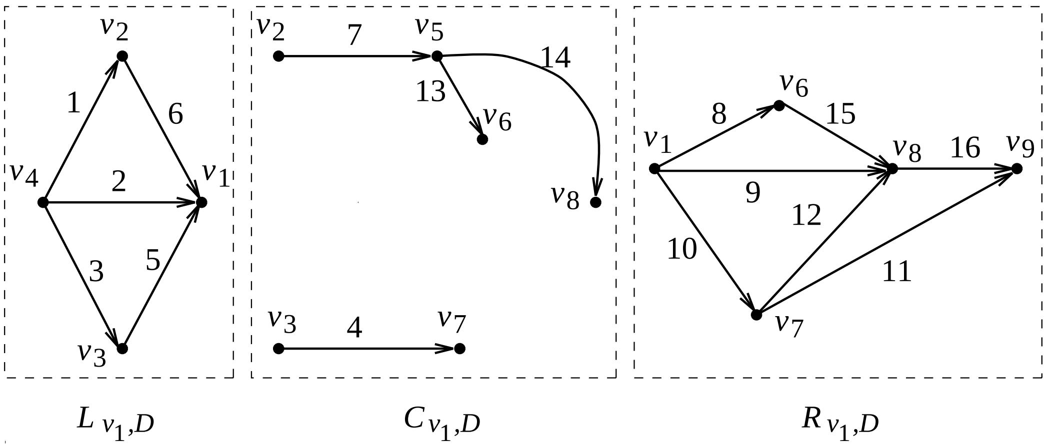


Figure 5.3: The partition of the V-graph (D) in Figure 5.2 induced by v_1.

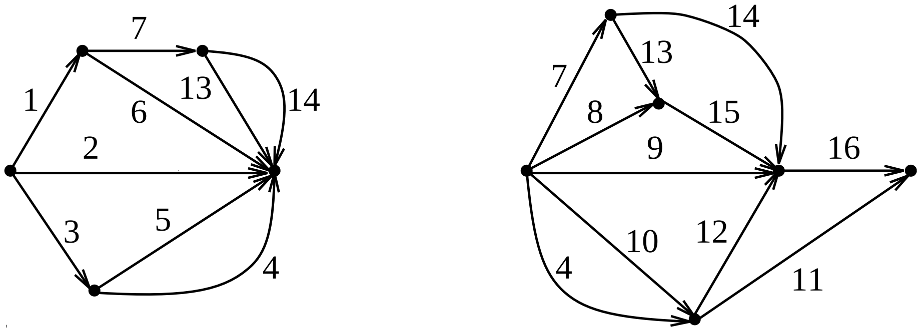


Figure 5.4: $D_L(v_1)$ and $D_R(v_1)$ for the V-graph in Figure 5.2.

Finally, we show the following result the implication of which will be given in the next section.

Theorem 5.1 *Let v be a vertex in D. Then, the set of full paths in D^* is equal to the union of the set of full paths in $(D_L(v))^*$ and the set of full paths in $(D_R(v))^*$.*

Proof: Let P_1 be the set of full paths in D^* and P_2 be the union of those of $(D_L(v))^*$ and $(D_R(v))^*$. Let s and t be the source and sink of D, respectively.

Let $E \in P_2$. Without loss of generality (w.l.o.g. for short), we may assume that E is a full path in $(D_L(v))^*$. Consequently, E is a cut in $D_L(v)$ by Lemma 5.1. Let $\{S, V(D_L(v)) - S\}$, where $s \in S$ and $v \notin S$, denote the partition induced by cut E on $V(D_L(v))$. It is easy to see that E is also a cut in D with $\{S, V(D) - S\}$ being the corresponding partition on $V(D)$. Therefore, E is a full path in D^* by Lemma 5.1, i.e., $E \in P_1$.

On the other hand, let $E \in P_1$. By Lemma 5.1, E is a cut in D. Let $\{V(D) - S, S\}$, where $s \notin S$ and $t \in S$, denote the partition induced by E on $V(D)$. By Lemma 5.2, either $E \cap A(L_{v,D}) = \emptyset$ or $E \cap A(R_{v,D}) = \emptyset$, since for any $e_1 \in A(L_{v,D})$ and $e_2 \in A(R_{v,D})$ there is a path in D that contains both e_1 and e_2. W.l.o.g. we assume $E \cap A(L_{v,D}) = \emptyset$. It is obvious that E is a cut in $D_R(v)$ with $\{V(D_R(v)) - S, S\}$ being the corresponding partition on $V(D_R(v))$. Therefore, E is a full path in $(D_R(v))^*$, i.e., $E \in P_2$. $\square$

5.2 Generalized cutlines

The polynomial algorithm for slicing floorplans is based on the fact that there is always a channel that divides a slicing floorplan into two (sub)floorplans (if, of course, the floorplan has more than one block). A channel with this property is called a *cutline*. Our methods are based on a generalization of the concept of cutlines. For simplicity, we shall use D to denote the V-graph of a floorplan from now on. It is easy to see that this convention does not place any restriction on our results since if we turn a floorplan by 90° counter-clockwise, the roles of H-graph and V-graph are interchanged.

Let us first recall Stockmeyer's result in [78]. Consider the (partially drawn) floorplan F_1 in Figure 5.5(a). The vertical cutline v partitions the floorplan into two floorplans L_1 and R_1 as shown in Figure 5.5(b). The key result is that all the nonredundant realizations of F_1 can be derived from the nonredundant realizations of L_1 and R_1. Moreover, the number of nonredundant realizations of F_1 is at most the sum of those of L_1 and R_1. This result leads to the efficient area minimization algorithm for slicing floorplans since the result can be applied to both L_1 and R_1 recursively in the case that F_1 is slicing.

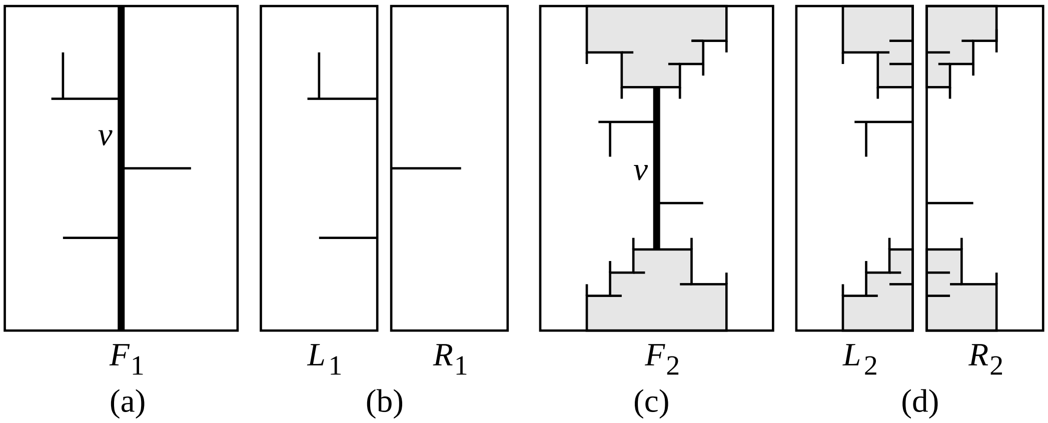


Figure 5.5: Illustration of a cutline and a generalized cutline.

However, if the floorplan considered is non-slicing, there may be no cutline that partitions the floorplan into two floorplans. A major contribution is the introduction of the concept of *generalized cutlines* and prove a result similar to Stockmeyer's. A generalized cutline is induced by a channel. The main idea can be illustrated intuitively first as follows: Consider the (partially drawn) floorplan F_2 in Figure 5.5(c). The vertical channel v induces a partition of F_2 into three parts as shown in Figure 5.5(c) — the shaded one and the two unshaded ones on either side of v. Two floorplans L_2 and R_2 are derived from the partition as shown in Figure 5.5(d): L_2 consists of the left unshaded part of F_2 together with the shaded part, and R_2 consists of the right unshaded part of F_2 together with the shaded part. (A precise description on how L_2 and R_2 are constructed will be given later. They will be referred to as $\mathcal{F}_L(v)$ and $\mathcal{F}_R(v)$, respectively.) If each block in the shaded part has only one realization, we can prove that the nonredundant realizations of F_2 can be derived from the L-nonredundant (defined later) realizations of L_2 and the R-nonredundant (defined later) realizations of R_2. Moreover, the number of nonredundant realizations of F_2 is at most the sum of the number of L-nonredundant realizations of L_2 and the number of R-nonredundant realizations of R_2. This result can be viewed as a generalization of Stockmeyer's result since it is reduced to Stockmeyer's result when v is a cutline (in that case, the shaded part of F_2 is empty). Just like Stockmeyer's result for slicing floorplans, our result leads to a recursive area minimization algorithm for general floorplans (**AMP-1** in the next section). In practice, L_2 and/or R_2 could be slicing even though F_2 is not. In general, if only a small number of generalized cutlines are used in the recursive applications of our result to partition a floorplan (that is what we mean by "approximately" slicing), our method is quite efficient. Besides this direct generalization of Stockmeyer's method, we also present a variation of the method which is more efficient in the worst case.

We now turn to the formal description of our generalization. Let $\mathcal{F}$ be a floorplan and v be a V-channel in $\mathcal{F}$. The *generalized cutline* induced by v consists of two paths c_L and c_R in $\mathcal{F}$ which are defined as follows: c_L (resp. c_R) consists of all the line segments in $\mathcal{F}$ that are traversed in the following steps:

(i) We move along v from the lower end of v all the way up until v meets an H-channel, then we move left (resp. right) along the H-channel until a V-channel is met. We then move up along the V-channel until an H-channel is met again. We shall move upward and leftward (resp. rightward) until the upper boundary of the enveloping rectangle is reached.

(ii) We move along v from the upper end of v all the way down until v meets an H-channel, then we move left (resp. right) along the H-channel until a V-channel is met. We then move down along the V-channel until an H-channel is met again. We shall move downward and leftward (resp. rightward) until the lower boundary of the enveloping rectangle is reached.

As an example, the dashed lines in Figure 5.1 indicate the generalized cutline c_L and c_R induced by the V-channel v_1. A generalized cutline c_L and c_R partitions $\mathcal{F}$ into three parts: The blocks to the left of c_L (exactly corresponding to the arcs in $L_{v,D}$ defined in the last section), the blocks to the right of c_R (exactly corresponding to the arcs in $R_{v,D}$), and the blocks between c_L and c_R (exactly corresponding to the arcs in $C_{v,D}$). Let $\mathcal{F}_L(v)$ (resp. $\mathcal{F}_R(v)$) be the floorplan obtained by removing all blocks in $R_{v,D}$ (resp. $L_{v,D}$) and making c_R (resp. c_L) a vertical straight line by pushing the blocks between c_L and c_R to the left (resp. right) without altering the relative positions among them. $D_L(v)$ and $D_R(v)$ defined in the last section are exactly the V-graphs of $\mathcal{F}_L(v)$ and $\mathcal{F}_R(v)$, respectively. For example, Figure 5.6 shows the floorplans $\mathcal{F}_L(v_1)$ and $\mathcal{F}_R(v_1)$ corresponding to the generalized cutline c_L and c_R in Figure 5.1. It can be verified that $D_L(v_1)$ and $D_R(v_1)$ in Figure 5.4 are their V-graphs.

If there is a vertical cutline that partitions a floorplan into a left sub-floorplan and a right sub-floorplan, then the height of a realization of the floorplan is equal to the height of either the induced realization of the left sub-floorplan or the induced realization of the right sub-floorplan

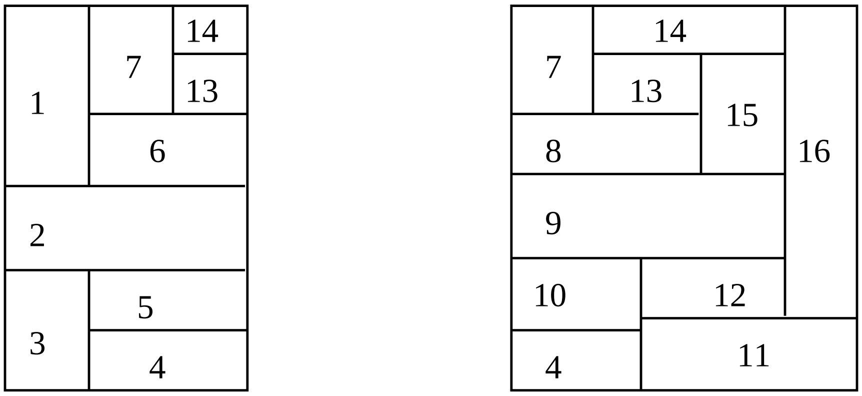


Figure 5.6: $\mathcal{F}_L(v_1)$ and $\mathcal{F}_R(v_1)$ for the floorplan in Figure 5.1.

whichever is larger. Or, in terms of polar graphs, the H-graph of the floorplan is equal to the H-graph of the left sub-floorplan and the H-graph of the right sub-floorplan connected in *parallel* [81]. Theorem 5.1 in last section is a generalization of this observation to a generalized cutline. What Theorem 5.1 says is that D^* can be viewed as $(D_L(v))^*$ and $(D_R(v))^*$ connected in parallel so far as the full paths of D^* are concerned. Or, in terms of floorplans, $\mathcal{F}$ can be viewed as a floorplan that has a vertical cutline with $\mathcal{F}_L(v)$ to the left and $\mathcal{F}_R(v)$ to the right as far as the height of $\mathcal{F}$ is concerned. In order to consider both the width and height of $\mathcal{F}$, we further generalize the concept of domination.

Definition 5.1 *Let $V' \subseteq V(D) - \{s, t\}$. Let σ_1 and σ_2 be two realizations of $\mathcal{F}$. σ_1 is said to L-dominate (resp. R-dominate) σ_2 with respect to (w.r.t. for short) V', if (1) σ_1 dominates σ_2 and (2) $l_D^{\sigma_1}(s, v) \leq l_D^{\sigma_2}(s, v)$ (resp. $l_D^{\sigma_1}(t, v) \leq l_D^{\sigma_2}(t, v)$) for any $v \in V'$.*

Plainly speaking, σ_1 L-dominates σ_2, if not only does σ_1 dominate σ_2, but also the length of a longest path from the left boundary of σ_1 to any channel v in V', $l_D^{\sigma_1}(s, v)$, is smaller than or equal to the corresponding quantity for σ_2, $l_D^{\sigma_2}(s, v)$. L-domination and R-domination are essentially the same concept since one becomes the other if the orientations of the arcs in D are reversed. L-domination is also a generalization of the conventional notion of domination because they become the same when $V' = \emptyset$. Similar to the conventional notion of domination, we can define the notions of *L-redundant*, *L-nonredundant*, *R-redundant*, and *R-nonredundant*.

Let $\mathcal{B}$ be the set of blocks of $\mathcal{F}$ which are on the directed paths from s to the vertices in V'. If $\sigma_1(B) = \sigma_2(B)$ for any $B \in \mathcal{B}$, part (2) of Definition 1 is vacuously satisfied. Consequently, *Procedure* **L-R**$(D, \mathcal{F}, V')$

1. $\mathcal{B} \leftarrow$ the set of blocks on the directed paths from s to vertices in V';

2. $L \leftarrow \emptyset$;

3. For each selection σ for $\mathcal{B}$ do
/* This step will be executed only once if $\mathcal{B} = \emptyset$. */

3.1 *temp* $\leftarrow$ the set of nonredundant realizations of $\mathcal{F}$ with the blocks in $\mathcal{B}$ having the realizations specified by σ and all the other blocks having their original sets of realizations;
/* May need to call an area minimization algorithm. */

3.2 $L \leftarrow$ the set of L-nonredundant realizations of $L \cup temp$;

4. Return L.

Figure 5.7: Procedure **L-R**.

to check whether σ_1 L-dominates σ_2, we only need to check whether σ_1 dominates σ_2. This simple observation suggests a straightforward method for finding the set of L-nonredundant realizations of a floorplan w.r.t. V': For each selection σ for $\mathcal{B}$ determine the corresponding set of nonredundant realizations of $\mathcal{F}$ with each block in $\mathcal{B}$ having the realization specified by σ. Then, remove all L-redundant realizations from the union of the sets of nonredundant realizations of $\mathcal{F}$ over all possible choices of σ for $\mathcal{B}$ according to the definition itself. To do so, just imagine a realization σ_1 of $\mathcal{F}$ as a "multi-dimensional rectangle" with $l_D^{\sigma_1}(s, v)$ for $v \in V'$ as distinct dimensions (in addition to width and height), then use a generalization of the method for determining the nonredundant realizations of a set of given realizations. The procedure **L-R** in Figure 5.7 is a direct rendition of this idea.

Let σ be a selection for $A(C_{v,D})$, we define a dummy selection σ_0 for $A(C_{v,D})$ such that for each block B in $A(C_{v,D})$, $\sigma_0(B) = (0, h)$ if $\sigma(B) = (w, h)$. The following result enables us to determine the redundant selections for $A(L_{v,D})$ and $A(R_{v,D})$.

Theorem 5.2 *Let σ be a selection for $A(C_{v,D})$. Let $x \vee \sigma_0$ and $y \vee \sigma_0$ be two realizations of $D_L(v)$ such that $x \vee \sigma_0$ L-dominates $y \vee \sigma_0$ w.r.t. LC. Let $\sigma_0 \vee z$ be a realization of $D_R(v)$. Then $x \vee \sigma \vee z$ dominates $y \vee \sigma \vee z$.*

Proof: Let $h_{x\vee\sigma\vee z}$ be the height of realization $x \vee \sigma \vee z$. Similarly, we define $w_{x\vee\sigma\vee z}$, $h_{x\vee\sigma_0}$ etc. By Theorem 5.1, we have

$$h_{x\vee\sigma\vee z} = max\{h_{x\vee\sigma_0}, h_{\sigma_0\vee z}\} \text{ and } h_{y\vee\sigma\vee z} = max\{h_{y\vee\sigma_0}, h_{\sigma_0\vee z}\}.$$

Because $x \vee \sigma_0$ L-dominates $y \vee \sigma_0$, $h_{x\vee\sigma_0} \leq h_{y\vee\sigma_0}$. Therefore, $h_{x\vee\sigma\vee z} \leq h_{y\vee\sigma\vee z}$.

We consider now the widths of the realizations. The proof proceeds by showing that for each full path p in D, $l_D^{x\vee\sigma\vee z}(p) \leq w_{y\vee\sigma\vee z}$. We may assume that $p = p_L \cup p_{CR}$, where $p_L \subseteq A(L_{v,D})$ is a path from s to some $u \in LC \cup \{v\}$, and $p_{CR} \subseteq A(C_{v,D}) \cup A(R_{v,D})$ is a path from u to t. Thus,

$$l_D^{x\vee\sigma\vee z}(p) = l_D^{x\vee\sigma\vee z}(p_L) + l_D^{x\vee\sigma\vee z}(p_{CR})$$

$$\leq l_{D_L(v)}^{x\vee\sigma_0}(s,u) + l_D^{\sigma\vee z}(p_{CR}) \leq l_{D_L(v)}^{y\vee\sigma_0}(s,u) + l_D^{\sigma\vee z}(p_{CR}) \leq w_{y\vee\sigma\vee z}.$$

□

Remark: If we interchange the roles of L and R, the symmetric result is also valid.

The reason to introduce σ_0 is that we do not know how the width of the realization of each block in $A(C_{v,D})$ is to be distributed between $\mathcal{F}_L(v)$ and $\mathcal{F}_R(v)$. Note that each block in $A(C_{v,D})$ appears in both $\mathcal{F}_L(v)$ and $\mathcal{F}_R(v)$. This is also the reason that we introduce the concepts of L-domination and R-domination which enable us to ignore some of the selections for $A(L_{v,D})$ and $A(R_{v,D})$ even when the width of the realization of each block in $A(C_{v,D})$ is disregarded. With Theorems 5.1 and 5.2, we have the following generalization of Stockmeyer's result [78]:

Theorem 5.3 *Let σ be a selection for $A(C_{v,D})$. Let S_σ denote the set of nonredundant realizations of $\mathcal{F}$ with each block in $A(C_{v,D})$ having the realization specified by σ. Denote by S_L (S_R) the set of L-nonredundant (R-nonredundant) realizations of $\mathcal{F}_L(v)$ ($\mathcal{F}_R(v)$) w.r.t. LC (RC) with each block in $A(C_{v,D})$ having the realization specified by σ_0. Then S_σ can be derived from S_L and S_R and $|S_\sigma| \leq |S_L| + |S_R| - 1$.*

Proof: From Theorem 5.2, it is clear that

$$S_\sigma \subseteq \{x \vee \sigma \vee y \mid x \vee \sigma_0 \in S_L \text{ and } \sigma_0 \vee y \in S_R\}.$$

Furthermore, by Theorem 5.1, the height of $x \vee \sigma \vee y$, where x is a selection for $A(L_{v,D})$ and y is a selection for $A(R_{v,D})$, is equal to the larger one between the height of $x \vee \sigma_0$ and the height of $\sigma_0 \vee y$, so $|S_\sigma| \leq |S_L| + |S_R| - 1$. □

Remark: If v is a cutline, then $A(C_{v,D}) = \emptyset$ and $LC = RC = \emptyset$. Therefore, S_L (S_R) is simply the set of nonredundant realizations of $\mathcal{F}_L(v)$ ($\mathcal{F}_R(v)$). In this case, this result is reduced to that of Stockmeyer.

5.3 The new methods

With the concept of generalized cutlines, we can now present our area minimization methods. Our first area minimization method is a direct application of Theorem 5.3, a generalization of Stockmeyer's result. In this method, the selections for the blocks in $A(C_{v,D})$ are considered one by one. For each selection σ for $A(C_{v,D})$, S_L and S_R are determined by calling procedure **L-R** which in turn may call an area minimization algorithm to determine sets of nonredundant realizations.

We now consider how to determine S_σ, the set of nonredundant realizations of $\mathcal{F}$ with each block in $A(C_{v,D})$ having the realization specified by σ, from S_L and S_R. One simple way is to pair off the realizations in S_L with the realizations in S_R. (Note that for a pair $x \vee \sigma_0$ and $\sigma_0 \vee y$, the corresponding realization of $\mathcal{F}$ is $x \vee \sigma \vee y$.) Then calculate the dimensions for the realizations formed by the pairs and remove the redundant ones. Procedure **L-R-merge** presented later in Figure 5.9 is a more efficient algorithm for doing so. However, it should be note that the worst-case time complexity of **L-R-merge** is still $O(|S_L||S_R|)$, the same as direct pairing off.

This area minimization method called **AMP-1** is presented in Figure 5.8. The time complexity of this method is $O(\Sigma T_L T_R)$, where Σ is the number of σ's, and T_L and T_R are the times it takes to generate S_L and S_R, respectively.

We next present a slightly different area minimization algorithm which has a better worst-case performance. We begin with a brief outline of the method. Suppose there is only one realization for each block in $A(L_{v,D})$ and $A(C_{v,D})$. In other words, we are given a selection x for $A(L_{v,D})$ and a selection σ for $A(C_{v,D})$. According to **AMP-1** we need to determine the corresponding S_L and S_R. It is obvious that $S_L = \{x \vee \sigma_0\}$. However, to determine S_R could be

AMP-1($\mathcal{F}$)

1. $L \leftarrow \emptyset$;

2. Choose a channel v in $\mathcal{F}$ (suppose v in V-graph D);

3. For each selection σ for $A(C_{v,D})$ do
/* If v is a cutline, this step will be executed only once. */

3.1. Apply procedure **L-R** to determine S_L and S_R;

3.2. $temp \leftarrow$**L-R-merge**$(S_L, S_R, \sigma, D, v, \mathcal{F})$;

3.3. $L \leftarrow$ the set of nonredundant realizations of $L \cup temp$;
/* Using the procedure described in Section 5.1. */

4. Return L.

Figure 5.8: Area minimization method **AMP-1**.

time-consuming. One the other hand, since x and σ have been given, we can distribute the widths of the realizations of the blocks in $A(C_{v,D})$ between $\mathcal{F}_L(v)$ and $\mathcal{F}_R(v)$ as follows: The intuitive idea is to push the realizations of the blocks in $A(C_{v,D})$ into $\mathcal{F}_L(v)$ so long as the width of the resulting realization of $\mathcal{F}_L(v)$ is bounded by the width of $x \vee \sigma_0$ and place the remaining parts of the realizations in the corresponding blocks in $\mathcal{F}_R(v)$. We use σ_1 to denote the portions of the realizations of the blocks in $A(C_{v,D})$ that are pushed into $\mathcal{F}_L(v)$ and σ_2 to denote the portions that are left in $\mathcal{F}_R(v)$. σ_1 and σ_2 can be defined precisely as follows: For each $B \in A(C_{v,D})$, if $\sigma(B) = (w, h)$, then $\sigma_1(B) = (w', h)$ and $\sigma_2(B) = (w - w', h)$, where $w' = min\{w, max\{l_D^{x\vee\sigma}(s, v) - l_D^{x\vee\sigma}(s, u), 0\}\}$ with u being the channel that B's left-boundary is on. The next result (Theorem 5.4) shows that if the realizations of the blocks in $A(C_{v,D})$ are distributed this way, for any selection y for $A(R_{v,D})$, the width of the realization $x \vee \sigma \vee y$ of $\mathcal{F}$ is equal to the sum of the width of $x \vee \sigma_1$ and the width of $\sigma_2 \vee y$. First, we prove the following lemma.

Lemma 5.3 *Let $u \in V(D_L(v))$, then $l_D^{x\vee\sigma}(s, v) \geq l_{D_L(v)}^{x\vee\sigma_1}(s, u)$.*

Proof: (By induction) If $u \in V(L_{v,D}) - \{v\}$, it is obvious that the inequality is true since any path from s to u in $D_L(v)$ can be extended to become a path from s to v in D. Suppose the inequality holds for all predecessors[1] of u. Let $p = p_1 \cup \{B\}$ be a longest path from s to u in $D_L(v)$, where p_1 is a path from s to u_1 and B is the block corresponding to arc (u_1, u) and

[1] u_1 is said to be a predecessor of u if (u_1, u) is an arc.

$\sigma(B) = (w, h)$. If $B \in A(L_{v,D})$, then $p \subseteq A(L_{v,D})$, so it is obvious that $l_D^{x\vee\sigma}(s,v) \geq l_{D_L(v)}^{x\vee\sigma_1}(p)$. Otherwise,

$$l_{D_L(v)}^{x\vee\sigma_1}(p) \leq l_{D_L(v)}^{x\vee\sigma_1}(s,u_1) + w' \leq l_{D_L(v)}^{x\vee\sigma_1}(s,u_1) + max\{l_D^{x\vee\sigma}(s,v) - l_D^{x\vee\sigma}(s,u_1), 0\}$$

$$= max\{l_D^{x\vee\sigma}(s,v) - l_D^{x\vee\sigma}(s,u_1) + l_{D_L(v)}^{x\vee\sigma_1}(s,u_1), l_{D_L(v)}^{x\vee\sigma_1}(s,u_1)\}.$$

Since w' is less than or equal to w, it is obvious that $l_D^{x\vee\sigma}(s,u_1) \geq l_{D_L(v)}^{x\vee\sigma_1}(s,u_1)$. According to the hypothesis, we also have $l_{D_L(v)}^{x\vee\sigma_1}(s,u_1) \leq l_D^{x\vee\sigma}(s,v)$. Therefore, $l_{D_L(v)}^{x\vee\sigma_1}(p) \leq l_D^{x\vee\sigma}(s,v)$. This concludes the proof. □

Theorem 5.4 *For any selection y for $A(R_{v,D})$,*

$$l_D^{x\vee\sigma\vee y}(s,t) = l_{D_L(v)}^{x\vee\sigma_1}(s,v) + l_{D_R(v)}^{\sigma_2\vee y}(v,t) \quad and \quad l_{D_L(v)}^{x\vee\sigma_1}(s,v) = l_D^{x\vee\sigma}(s,v) \quad (= l_{D_L(v)}^{x\vee\sigma_0}(s,v)).$$

Proof: According to the way σ_1 and σ_2 are defined, it is easy to see that

$$l_D^{x\vee\sigma\vee y}(s,t) \leq l_{D_L(v)}^{x\vee\sigma_1}(s,v) + l_{D_R(v)}^{\sigma_2\vee y}(v,t).$$

Now we consider the other direction of the inequality. According to Lemma 5.3 and the fact that $l_D^{x\vee\sigma}(s,v) \leq l_{D_L(v)}^{x\vee\sigma_1}(s,v)$, we have

$$l_{D_L(v)}^{x\vee\sigma_1}(s,v) = l_D^{x\vee\sigma}(s,v) \tag{5.1}$$

Let p_1 be a longest full path in $D_L(v)$ with weights assigned to the arcs according to selection $x \vee \sigma_1$. Let $p_2 = p_3 \cup p_4$ be a longest full path in $D_R(v)$ with weights assigned to the arcs according to selection $\sigma_2 \vee y$, where $p_3 \subseteq A(C_{v,D})$ is a directed path from v to u_2 such that $|p_3|$ is the minimum among all longest full paths in $D_R(v)$ and $p_4 \subseteq A(R_{v,D})$ is a directed path from u_2 to t, for some $u_2 \in RC$. W.l.o.g. we assume $p_1 \subseteq A(L_{v,D})$ (because of (1)). Therefore, if $p_3 = \emptyset$, then $p_1 \cup p_2$ is also a full path in D, we are done. If $p_3 \neq \emptyset$, according to the way p_3 was chosen, it must be the case that $l_{D_R(v)}^{\sigma_2\vee y}(p_3) > 0$. Let B be the first block along p_3 that has non-zero width according to σ_2. Let the left-boundary of B be on channel u_1 and the right-boundary of B be on channel u_2 in $\mathcal{F}$. Denote by $\mathcal{B}(u_1)$ the set of all the blocks on the paths from v to u_1 in $D_R(v)$. Let $p_3 = p_7 \cup \{B\} \cup p_8$, where $p_7 = p_3 \cap \mathcal{B}(u_1)$. It is obvious that for any $B' \in \mathcal{B}(u_1)$, the realization of B' specified by σ_2 has zero width (for, otherwise, p_2 would not be a longest full path in $D_R(v)$). Therefore, $l_D^{x\vee\sigma}(s,u_1) = l_{D_L(v)}^{x\vee\sigma_1}(s,u_1)$. Let $p_5 \cup p_6$ be the path that attains the length $l_{D_L(v)}^{x\vee\sigma_1}(s,u_1)$ with $p_5 \subseteq A(L_{v,D})$ and $p_6 \subseteq \mathcal{B}(u_1)$. Then,

$$l_{D_L(v)}^{x\vee\sigma_1}(p_5 \cup p_6 \cup \{B\}) = l_{D_L(v)}^{x\vee\sigma_1}(p_5 \cup p_6) + w' = l_{D_L(v)}^{x\vee\sigma_1}(s,u_1) + max\{l_D^{x\vee\sigma}(s,v) - l_D^{x\vee\sigma}(s,u_1), 0\}$$

$$\geq l_D^{x\vee\sigma}(s,v) - l_D^{x\vee\sigma}(s,u_1) + l_{D_L(v)}^{x\vee\sigma_1}(s,u_1) = l_D^{x\vee\sigma}(s,v).$$

Combining the above with Lemma 5.3, we obtain

$$l_{D_L(v)}^{x\vee\sigma_1}(p_5 \cup p_6 \cup \{B\} \cup p_8) = l_D^{x\vee\sigma}(s,v).$$

Thus,

$$l_{D_L(v)}^{x\vee\sigma_1}(p_1) + l_{D_R(v)}^{\sigma_2\vee y}(p_2) = l_D^{x\vee\sigma}(s,v) + l_{D_R(v)}^{\sigma_2\vee y}(p_2)$$

$$= l_{D_L(v)}^{x\vee\sigma_1}(p_5 \cup p_6 \cup \{B\} \cup p_8) + l_{D_R(v)}^{\sigma_2\vee y}(p_6 \cup \{B\} \cup p_8 \cup p_4).$$

By the definitions of σ_1 and σ_2,

$$l_{D_L(v)}^{x\vee\sigma_1}(p_6 \cup \{B\} \cup p_8) + l_{D_R(v)}^{\sigma_2\vee y}(p_6 \cup \{B\} \cup p_8) = l_D^{x\vee\sigma\vee y}(p_6 \cup \{B\} \cup p_8).$$

Therefore $l_{D_L(v)}^{x\vee\sigma_1}(p_1) + l_{D_R(v)}^{\sigma_2\vee y}(p_2) = l_D^{x\vee\sigma\vee y}(p_5 \cup p_6 \cup \{B\} \cup p_8 \cup p_4) \leq l_D^{x\vee\sigma\vee y}(s,t)$. So the theorem follows. □

Remark: If we swap L and R, replace s by t, and left by right in Theorem 5.4, the symmetric result is also true.

Let **V-merge**(S_1, S_2) denote the procedure which combines S_1 and S_2, the sets of realizations of the left and right (sub)floorplans of a vertical cutline, to generate the set of nonredundant realizations of the floorplan in [78][2]. By Theorem 5.1 and Theorem 5.4, $\mathcal{F}$ can be viewed as a floorplan formed by $\mathcal{F}_L(v)$ and $\mathcal{F}_R(v)$ with a vertical cutline. Let σ be a given selection for $A(C_{v,D})$, and X be a set of selections for $A(L_{v,D})$ such that every selection in X defines the same σ_1 and σ_2. Then, the set of nonredundant realizations of $\mathcal{F}$ with the form $x \vee \sigma \vee y$ for any x in X and any selection y for $A(R_{v,D})$, is the set of realizations generated by **V-merge**(S_1, S_2)

[2]The procedure for horizontal cutlines should be used if the generalized cutline is induced by an H-channel.

Procedure **L-R-merge**$(S_L, S_R, \sigma, D, v, \mathcal{F})$

/* D is the V-graph of $\mathcal{F}$. The output is the set of nonredundant realizations of $\mathcal{F}$ with each block in $A(C_{v,D})$ having one realization specified by σ. */

1. Group the realizations in S_L so that all realizations in the same group have the same value for $l^{x\vee\sigma_0}_{D_L(v)}(s,v) - l^{x\vee\sigma_0}_{D_L(v)}(s,u)$ for each $u \in LC$, suppose $S_L = S_{1,1} + S_{1,2} + \cdots + S_{1,m}$;
/* Note that $m = 1$ if v is a cutline. */

2. Group the realizations in S_R so that all realizations in the same group have the same value for $l^{\sigma_0\vee y}_{D_R(v)}(v,t) - l^{\sigma_0\vee y}_{D_R(v)}(u,t)$ for each $u \in RC$, suppose $S_R = S_{2,1} + S_{2,2} + \cdots + S_{2,n}$;
/* Note that $n = 1$ if v is a cutline. */

3. For $1 \le i \le m, 1 \le j \le n$, $L_{i,j} \leftarrow$ **V-merge**$(S_{1,i}, S_{2,j})$;
/* If pair $x \vee \sigma_0$ and $\sigma_0 \vee y$ is selected by **V-merge**, $x \vee \sigma \vee y$ is actually put into $L_{i,j}$. */

5. $L \leftarrow$ the set of nonredundant realizations of $\cup_{i,j} L_{i,j}$;
/* Using the procedure described in Section 5.1. */

6. Return L.

Figure 5.9: Procedure **L-R-merge**.

where $S_1 = \{x \vee \sigma_0 | x \in X\}$ and S_2 is the set of nonredundant realizations of $D_R(v)$ with each block in $A(C_{v,D})$ having one realization specified by σ_2. This is the main idea in our second area minimization method. The next result helps to find those X's whose elements define the same σ_1 and σ_2 in S_L (or S_R). The intuitive idea here is that if x_1 and x_2, two selections for $A(L_{v,D})$, have the same amount of room in the horizontal dimension that can be used for the realizations of the blocks in $A(C_{v,D})$, then x_1 and x_2 define the same σ_1 and σ_2.

Theorem 5.5 *Let x_1 and x_2 be two selections for $A(L_{v,D})$. If $l^{x_1\vee\sigma_0}_{D_L(v)}(s,v) - l^{x_1\vee\sigma_0}_{D_L(v)}(s,u) = l^{x_2\vee\sigma_0}_{D_L(v)}(s,v) - l^{x_2\vee\sigma_0}_{D_L(v)}(s,u)$ for any $u \in LC$, then x_1 and x_2 define the same σ_1 and σ_2.*

Proof: This result is quite obvious. □

Remark: If we replace L by R, s by t, the resulting statement is also valid.

Theorem 5.5 also leads to the procedure **L-R-merge** (Figure 5.9). Note that if v is a cutline, this algorithm is exactly the same as **V-merge**.

Our second area minimization algorithm **AMP-2** is presented in Figure 5.10. The main step is Step 3 which examines the selections for $A(C_{v,D})$ one by one. For each selection for $A(C_{v,D})$, **AMP-2** calls **L-R** to determine S_L (Step 3.1), then groups the realizations in S_L according

AMP-2($\mathcal{F}$)

1. $L \leftarrow \emptyset$;

2. Choose a channel v in $\mathcal{F}$ (suppose v is in V-graph D);

3. <u>For</u> each selection σ for $A(C_{v,D})$ <u>do</u>
/* Step 3 will be executed only once if $A(C_{v,D}) = \emptyset$ */

3.1. Apply procedure **L-R** to determine S_L;
/* Or S_R if we interchange the roles of R and L in this algorithm. Recursion may occur at this step. */

3.2. Group the realizations in S_L so that all realizations in the same group have the same value for $l^{x \vee \sigma_0}_{D_L(v)}(s, v) - l^{x \vee \sigma_0}_{D_L(v)}(s, u)$ for each $u \in LC$, suppose $S_L = S_{1,1} + S_{1,2} + \cdots + S_{1,m}$;
/* Note that $m = 1$ if v is a cutline. */

3.3. $L' \leftarrow \emptyset$;

3.4. <u>For</u> $i \leftarrow 1$ to m <u>do</u>

3.4.1. Compute σ_2;
/* Note that all realizations in $S_{1,i}$ define the same σ_2. */

3.4.2. $temp1 \leftarrow$ the set of nonredundant realizations of $D_R(v)$ with each block in $A(C_{v,D})$ having one realization specified by σ_2;
/* Recursion may occur at this step. */

3.4.3. $temp2 \leftarrow$ **V-merge** $(S_{1,i}, temp1)$;
/* If pair $x \vee \sigma_0$ and $\sigma_0 \vee y$ is selected by **V-merge**, $x \vee \sigma \vee y$ is actually put into $temp2$ */

3.4.4. $L' \leftarrow$ the set of nonredundant realizations of $L' \cup temp2$;

3.5. $L \leftarrow$ the set of nonredundant realizations of $L \cup L'$;

4. <u>Return</u> L.

Figure 5.10: Area minimization method **AMP-2**.

to Theorem 5.5 (Step 3.2) so that all realizations in the same group defining the same σ_1 and σ_2. Next it considers the groups of S_L one by one (Step 3.4). For each group $S_{1,i}$ **AMP-2** first computes σ_2 (Step 3.4.1), then determines $temp1$, the set of nonredundant realizations of $D_R(v)$ with each block in $A(C_{v,D})$ having one realization specified by σ_2 (Step 3.4.2), then uses one of Stockmeyer's procedures to combine $S_{1,i}$ and $temp1$ (Step 3.4.3). In **AMP-2**, L is the set of all realizations generated for $\mathcal{F}$. Note that **AMP-2** will also degenerate to Stockmeyer's algorithm if the floorplan being considered is slicing. It is easy to see that the time complexity of **AMP-2** is $O(\Sigma T_L T_R^{'})$ where Σ is the number of σ's, T_L is the time it takes to generate S_L, and $T_R^{'}$ is the time it takes to generate the set of nonredundant realizations of $\mathcal{F}_R(v)$. Obviously, $T_R^{'} \leq T_R$, so, **AMP-2** is faster that **AMP-1**.

In both **AMP-1** and **AMP-2**, there are steps in which there are choices to be made. For example, which channel is used to induce a generalized cutline? Should S_L or S_R be used in **AMP-2**? Even though these decisions do not affect the correctness of the methods, they do affect the efficiency of the methods very much. Good decisions can make the methods much faster. We do not know any iron-clad rule that can be used for these decisions. A general guideline that can be followed is to use cutlines instead of generalized cutlines whenever possible and try to select a generalized cutline so that the number of selections considered in Step 3 of **L-R, AMP-1**, and **AMP-2** is minimized if no cutline is available.

5.4 Examples

In this section we present several examples to illustrate the worst-case time and space costs when **AMP-2** is applied to determine the nonredundant realizations of a floorplan. In all examples, we assume that each basic block has k realizations. Thus, the time and space costs will be functions of k. Similar to Stockmeyer's algorithm, the execution of **AMP-2** generates a partition tree. A partition tree shows the floorplan hierarchy according to which the original floorplan is partitioned by **AMP-2**. To simplify the figures, we shall not show any further partitioning when a floorplan in a partition tree is slicing because, in that case, **AMP-2** is the same as Stockmeyer's algorithm.

Example 1: Consider floorplan F_3 in Figure 5.1. Figure 5.11 shows a partition tree. In Figure 5.11, the number in each block denotes the name of the block, and if the realization of a block has been fixed, we put a "*" in the block. (The same convention will be used in all later examples.) In the first step, suppose v_1 is chosen to partition F_3 (into L_3 and R_3). **AMP-2** considers the selections for $A(C_{v_1,D}) = \{4, 7, 13, 14\}$, the set of blocks enclosed by the generalized cut line (indicated by the two dashed lines), one by one (k^4 of them altogether). For each selection σ for $A(C_{v_1,D})$, i.e. fixing a realization for each of blocks 4, 7, 13, and 14, **AMP-2** calls **L-R** to determine S_L, the set of L-nonredundant realizations of L_3 w.r.t. $\{v_2, v_3\}$. To determine S_L, **L-R** considers all the combinations of realizations of blocks 1 and 3, which are the blocks on the paths from the source, v_4, of D to v_2 or v_3, one by one. For each combination, **L-R** determines the set of nonredundant realizations of L_3, which can be done in $O(k)$-time and $O(k)$-space since L_3 is slicing. Thus, S_L can be found in $O(k^3 \log k)$-time and $O(k^3)$-space. Next **AMP-2** considers the realizations in S_L one by one[3]. For each realization in S_L, **AMP-2** determines the set of nonredundant realizations of floorplan R_3 with each block in $A(C_{v_1,D})$ having the realization specified by the σ_2 as defined in last section. This can be done by calling **AMP-2** recursively. Notice that the leaf floorplans in Figure 5.11 are all slicing. For each of these leaf floorplans, the number of nonredundant realizations is bounded by the total number of realizations of its blocks. The worst-case sizes of the other sets of (nonredundant or L-nonredundant) realizations shown in Figure 5.11 can be bounded in a bottom-up fashion from those of the slicing leaf floorplans. These size bounds are listed beside the corresponding floorplans in Figure 5.11. It is not difficult to see that in a partition tree the time and space costs at a node (a floorplan) where the size of the set of realizations is at most t are in the worst-case $O(t \log t)$ and $O(t)$, respectively. Therefore, the total time and space costs of running **AMP-2** on F_3 are $O(k^{10} \log k)$ and $O(k^{10})$, respectively. These costs are bounded very conservatively. In practice, the actual time and space costs should be much better than the worst-case bounds since F_3 has at most $O(k^5)$ nonredundant realizations. Even for the worst-case bounds, they are substantially better than those of an exhaustive method which are $O(k^{16} \log k)$ and $O(k^{16})$ in the worst case.

Example 2: For floorplan F_4, Figure 5.12 shows a partition tree. In this case, only one generalized cutline is used, and **AMP-2** leads to an $O(k^3 \log k)$ time and $O(k^3)$ space algorithm, which has the same time and space costs as that of the algorithm in [83]. However, this algorithm

[3]Since we are interested in determining the worst-case performance, we assume that each realization in S_L forms a group itself. Namely, we assume $m = |S_L|$ in **AMP-2**.

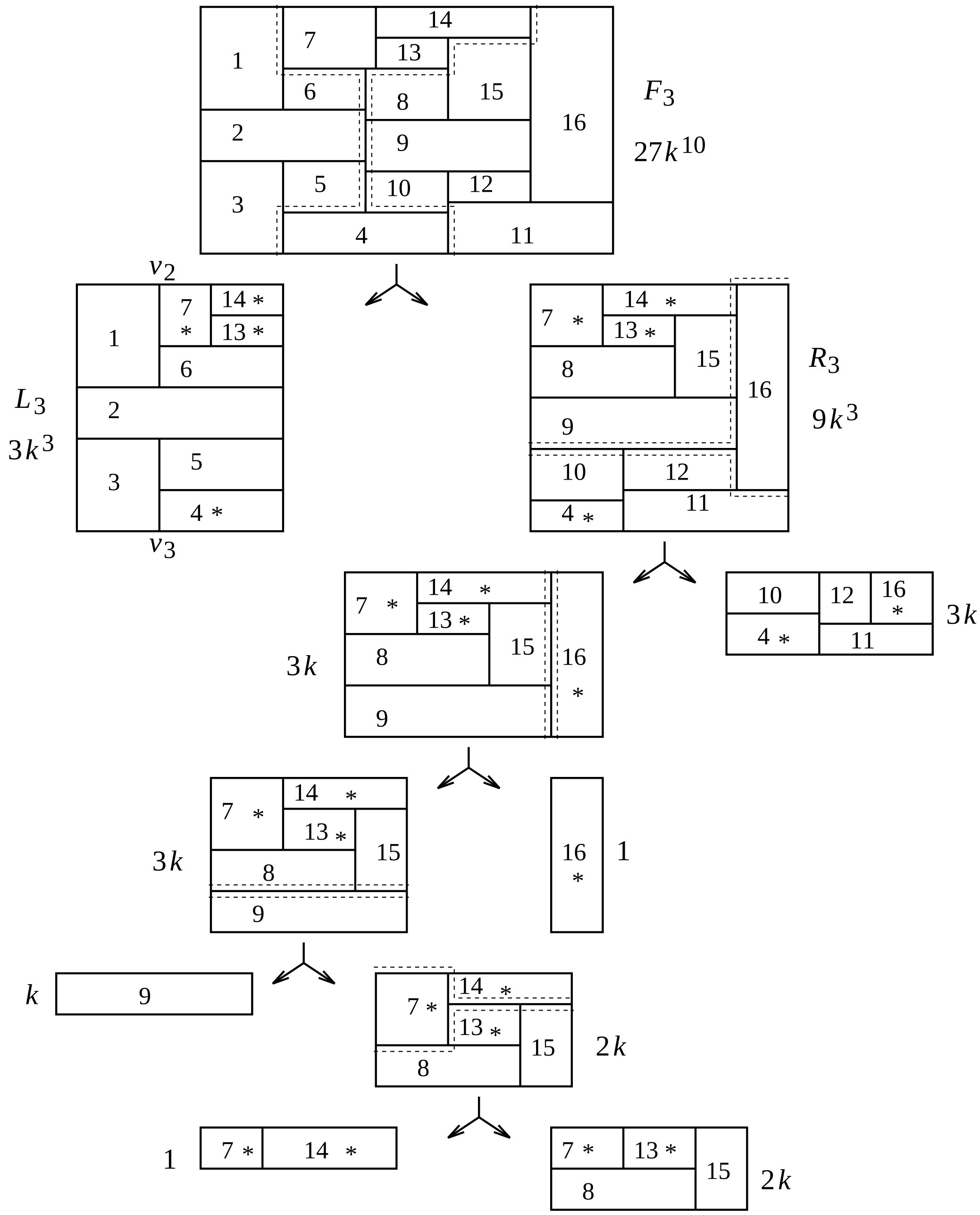


Figure 5.11: A partition tree for floorplan F_3. Here, for floorplan L_3, seek the set of L-nonredundant realizations w.r.t. $\{v_2, v_3\}$; for all other floorplans, seek the sets of nonredundant realizations.

is much simpler than the one in [83]. ([68] presents an algorithm which runs in $O(k^2 \log k)$ time and $O(k^2)$ space for this floorplan.)

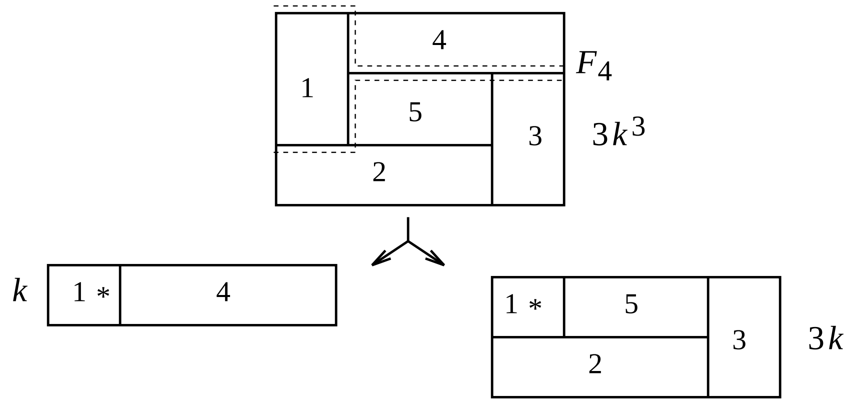


Figure 5.12: A partition tree for floorplan F_4. Here, for both floorplans, seek the sets of nonredundant realizations.

Example 3: For floorplan F_5, Figure 5.13 shows a partition tree. **AMP-2** leads to an $O(k^3 \log k)$ time and $O(k^3)$ space algorithm.

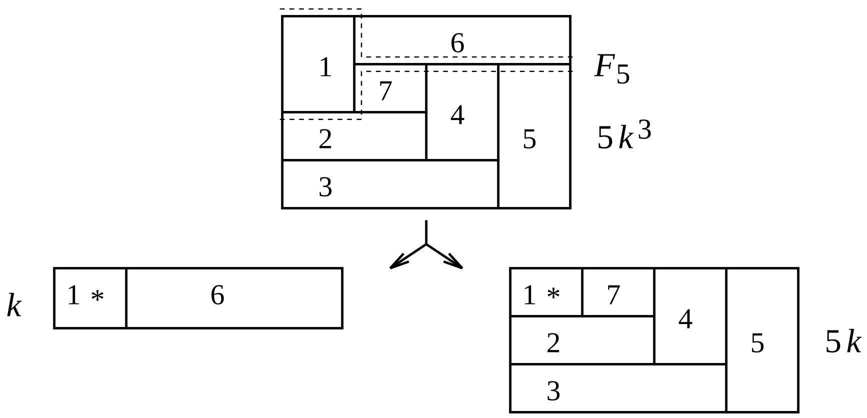


Figure 5.13: A partition tree for floorplan F_5. Here, for both floorplans, seek the sets of nonredundant realizations.

Example 4: For floorplan F_6, Figure 5.14 shows a partition tree. **AMP-2** leads to an $O(k^6 \log k)$ time and $O(k^6)$ space algorithm.

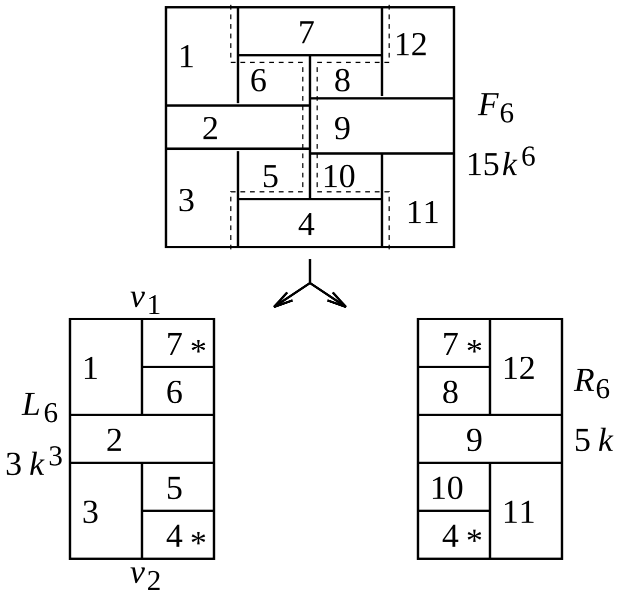


Figure 5.14: A partition tree for floorplan F_6. Here, for floorplan L_6, seek the set of L-nonredundant realizations w.r.t. $\{v_1, v_2\}$; for R_6, seek the set of nonredundant realizations.

5.5 Summary

Two methods for area minimization for general floorplans have been presented. Both methods explicitly take into account the structural information of the floorplan being considered, and render the well-known area minimization algorithm for slicing floorplans due to Otten and Stockmeyer as a degenerate case. Our experience shows that they are quite efficient for a large class of floorplans.

There are several open problems. An obvious one is to find good rules to guide the methods in making the decisions mentioned at the end of Section 5.3. Another important one is to reduce the number of iterations in Step 3 of **L-R, AMP-1**, and **AMP-2**. Another possible research direction is to apply the methods to area minimization for hierarchical floorplans.

Chapter 6

Area Minimization for Hierarchical Floorplans

In practice, floorplans are often constructed either by top-down partitioning or by bottom-up clustering. In a top-down construction we begin with the group consisting of all the modules, and at each step, a group of modules is partitioned into at most p groups of modules for some constant p. This process is continued until each group consists of a single module. A floorplan obtained this way is called a hierarchical floorplan (of order-p). A hierarchical floorplan (of order-p) can be described by a p-ary tree corresponding to the partitioning process. (Although a hierarchy can always be extracted from any floorplan [19], the degrees of the corresponding tree for a non-hierarchical floorplan, in general, cannot be upper-bounded by a constant.) Often, not all patterns that partition a group of modules into p or fewer groups are allowed. Instead, only a selected set of partitioning patterns is allowed in order to reduce the the size of the search space during floorplan design [49]. Since the cost of the partitioning process increases rapidly as p increases, the value of p is usually confined to be a small integer. In most cases, $p \leq 5$ [25, 47, 48].

In this chapter we study the area minimization problem for hierarchical floorplans. Of particular interest are hierarchical floorplans of order-5. The complexity of the area minimization problem for hierarchical floorplans was an open problem [48]. We will show that the problem is **NP**-complete. To be precise, we show that the area minimization problem for hierarchical floorplans constructed from a set of patterns that contains at least one non-slicing pattern is

NP-complete. This is true even for balanced hierarchical floorplans. Since there are pseudo-polynomial area minimization algorithms for hierarchical floorplans [82], the area minimization problem for hierarchical floorplans is only weakly **NP**-complete as opposed to that for non-hierarchical floorplans which is strongly **NP**-complete.

We next present a new algorithm for determining the nonredundant realizations of a wheel. Wheels are the only non-slicing patterns used in the construction of hierarchical floorplans of order-5. The worst case time and space costs of our algorithm are $O(k^2 \log k)$ and $O(k^2)$, respectively, where k is the maximum among the numbers of realizations of the blocks in a wheel. Both costs achieve a reduction by a factor of k in comparison with the previously best known algorithm in [83]. Recently, Chen and Tollis [14] developed independently an algorithm for wheels with the same time and space costs as ours. Our algorithm, however, is much simpler. Both algorithms are based on dividing the realizations of a wheel into classes, and finding possible candidates of nonredundant realizations in each class separately. However, the algorithms use different divisions as well as different approaches for determining possible candidates.

Based on the new algorithm for wheels, we design a new pseudo-polynomial area minimization algorithm for hierarchical floorplans of order-5. The time and space costs of the algorithm are $O((nM)^2 \log(nM))$ and $O(n^2M)$, respectively, where n is the number of blocks and M is an upper-bound on the dimensions of the realizations of the blocks. Though the algorithm is exponential in the worst case, it has been shown to be very efficient in practice, especially when M is not too large. Furthermore, the algorithm can be converted into an approximation scheme by rounding and scaling [37].

6.1 Preliminaries and problem definition

A set of hierarchical floorplans is defined by a *finite* set of (partitioning) patterns. Given a set of patterns , a floorplan is said to be *hierarchical* if either it is a single block, or it can be partitioned into sub-floorplans according to one of the patterns in the set and each sub-floorplan is hierarchical. As in the previous chapter, we only consider rectangular patterns. As a result, a pattern can be viewed as a floorplan with constant many blocks. A hierarchical floorplan can be represented by a tree, referred to as a *partition tree*. The leaves of the partition tree correspond to the basic blocks of the floorplan, and the internal nodes correspond to the sub-floorplans. Each internal node is labeled by the pattern that partitions the corresponding sub-floorplan into sub-floorplans. Figure 6.2 shows a hierarchical floorplan and its partition tree.

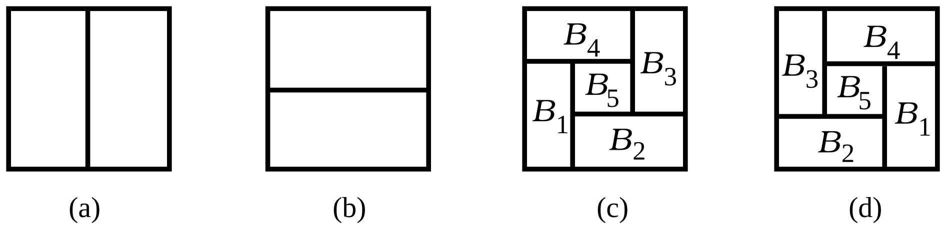


Figure 6.1: Vertical slice (a), horizontal slice (b), left wheel (c), and right wheel (d).

A *slice* is a pattern with two blocks. There are only two different slices, the vertical slice and the horizontal slice, as shown in Figure 6.1 (a) and (b). When the pattern set consists of only the two slices, the resulting hierarchical floorplans are called *slicing floorplans*. When the pattern set consists of all patterns with five or fewer blocks, the resulting hierarchical floorplans are called *hierarchical floorplans of order-5*. A *wheel* is a non-slicing pattern with five blocks. There are only two different wheels, the left wheel and the right wheel, as shown in Figure 6.1 (c) and (d). Wheels are the smallest non-slicing patterns. All other patterns with five or fewer blocks can be decomposed into slices. Consequently, hierarchical floorplans of order-5 can also be defined as those constructed from the pattern set consisting of the two slices and the two wheels. A hierarchical floorplan of order-5 and its partition tree are shown in Figure 6.2.

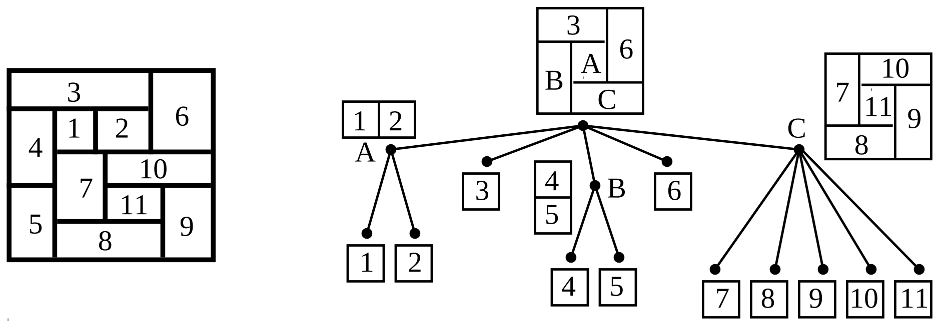


Figure 6.2: An hierarchical floorplan of order-5 and its partition tree.

The area minimization problem for hierarchical floorplans can be formally stated as follows: *Given a hierarchical floorplan F and a set of realizations for each of its blocks, determine a realization of F which has the minimum area.* We assume that the dimensions of a realization of a block are non-negative integers. The decision version of the problem can be stated as follows:
AREAMIN-$\mathcal{P}$

> **Instance:** A hierarchical floorplan F constructed from pattern set $\mathcal{P}$, a set of realizations for each block in F, and a positive integer A.
>
> **Question:** Does F have a realization with area less than or equal to A ?

When $\mathcal{P}$ consists of the two slices and the two wheel, i.e., F is a hierarchical floorplan of order-5, we denote **AREAMIN-$\mathcal{P}$** by **AREAMIN-5** for simplicity.

We will use $L(F)$ to denote the set of nonredundant realizations of F, which is either a block or a floorplan. Obviously, all the minimum area realizations of F are contained in $L(F)$.

6.2 NP-completeness results

In this section, we will show that area minimization for hierarchical floorplans is **NP**-complete. We first consider the case in which the dimensions of a realization of a block could be zero. We will show that **AREAMIN-5** is **NP**-complete. Using this result, we further show that **AREAMIN-$\mathcal{P}$** is **NP**-complete if $\mathcal{P}$ contains at least one non-slicing pattern. Next, we show how the proofs can be generalized to the case in which the dimensions of a realization of any block must be positive, as this is the case in practice. For the decision problems, only the dimensions of a realization matter. For this reason, realization r will be simply denoted by pair $(w(r), h(r))$ in this section.

Suppose that in a wheel W, B_1 and B_3 have the same set of nonredundant realizations (denoted L_1), B_2 and B_4 have the same set of nonredundant realizations (denoted L_2), and B_5 has $(0, 0)$ as its only realization. Under this assumption, we have the following result:

Lemma 6.1 *If $w \leq w'$ and $h \geq h'$ for any (w, h) in L_1 and (w', h') in L_2, then a nonredundant realization of W must select the same realization for B_1 and B_3, and the same realization for B_2 and B_4. In other words,*

$$L(W) \subseteq \{(w + w', h + h') \mid (w, h) \in L_1, (w', h') \in L_2\}.$$

Proof: Let r be a realization of W in which (w_i, h_i) is selected for B_i for $1 \leq i \leq 4$. Of course, $(0,0)$ is selected for B_5. Since $w \leq w'$ and $h \geq h'$ for any (w,h) in L_1 and (w',h') in L_2, it follows that

$$w(r) = \mathbf{max}\{w_1 + w_2, w_3 + w_4\}$$

$$h(r) = \mathbf{max}\{h_1 + h_4, h_2 + h_3\}.$$

By letting $(w_1, h_1) = (w_3, h_3) = (w, h)$ and $(w_2, h_2) = (w_4, h_4) = (w', h')$, the resulting r is $(w + w', h + h')$.

Now it suffices to prove the following claim: If $h(r) = h_1 + h_4$ (resp. $h(r) = h_2 + h_3$), then $w(r) \geq w_1 + w_4$ (resp. $w(r) \geq w_2 + w_3$). By symmetry, we only need to prove the case $h(r) = h_1 + h_4$. Suppose the claim is false, i.e., $w(r) < w_1 + w_4$. Then,

$$w_1 + w_2 \leq w(r) < w_1 + w_4 \Longrightarrow w_2 < w_4 \Longrightarrow h_2 > h_4.$$

$$w_4 + w_3 \leq w(r) < w_1 + w_4 \Longrightarrow w_3 < w_1 \Longrightarrow h_3 > h_1.$$

Now we have $h(r) \geq h_2 + h_3 > h_1 + h_4 = h(r)$, an obvious absurdity. $\square$

The problem used in proving the **NP**-completeness of **AREAMIN-5** is the following special case of **PARTITION**, which is known to be **NP**-complete [36]:

2-1-PARTITION

Instance: A list of positive integers $a_1, a_2, \ldots, a_{2n}$, where $n \geq 1$.

Question: Is there an index set I that contains exactly one of $2t+1$ and $2t+2$ for $0 \leq t \leq n-1$ such that $\sum_{i \in I} a_i = (a_1 + \cdots + a_{2n})/2$?

In the instance of **2-1-PARTITION**, if $2n$ is not a power of 2, we can append a sufficient number of 1's to the end of the list to make the total number of elements a power of 2 without changing the answer to the instance. Therefore, without loss of generality we assume $2n = 2^m$ where $m \geq 1$. We introduce the following notations:

$$\begin{aligned}
s(I) &= \textstyle\sum_{i \in I} a_i \text{ for } I \subseteq \{1, 2, \ldots, 2n\}. \\
I_{i,l} &= \{l, l+1, \ldots, l+2^i-1\}. \\
Q &= a_1 + \cdots + a_{2n}. \\
\mathcal{I}_{i,l} &= \{I \mid I \text{ contains exactly one of } l+2t \text{ and } l+2t+1 \text{ for } 0 \leq t \leq 2^{i-1}-1\}.
\end{aligned}$$

As mentioned in Chapter 2, determining the dimensions of a realization amounts to computing the maximum weights of the full paths in the V-graph and H-graph. Obviously **AREAMIN-5** is in **NP**. We now consider how to reduce the generic instance of **2-1-PARTITION** to an instance of **AREAMIN-5**. First, we define a family of floorplans $A_{i,l}$'s and the sets of realizations for their basic blocks recursively. $A_{i,l}$ is associated with the 2^i elements $a_l, a_{l+1}, \ldots, a_{l+2^i-1}$ in the instance of **2-1-PARTITION**. $A_{i,l}$ is defined in such a way that the dimensions of its nonredundant realizations reflect the index sets in $\mathcal{I}_{i,l}$. The precise definition is as follows:

(1) $A_{1,l}$, where $1 \le l \le 2^m - 1$, is a floorplan with only one block. The set of realizations of the block is $\{(a_l, a_{l+1}), (a_{l+1}, a_l)\}$.

(2) $A_{i,l}$, where $2 \le i \le m$ and $1 \le l \le 2^m - 2^i + 1$, is recursively defined by $A_{i-1,l}$ and $A_{i-1,l+2^{i-1}}$ as shown in Figure 6.3(a), where $E_{i,t}$'s are basic blocks, and $L(E_{i,1}) = L(E_{i,3}) = \{(1, Q)\}$, $L(E_{i,2}) = L(E_{i,4}) = \{(Q, 1)\}$, and $L(E_{i,5}) = \{(0, 0)\}$. $E_{i,t}$, for $1 \le t \le 4$, are referred to as *enforcement blocks* since their purpose is to enforce the condition in Lemma 6.1.

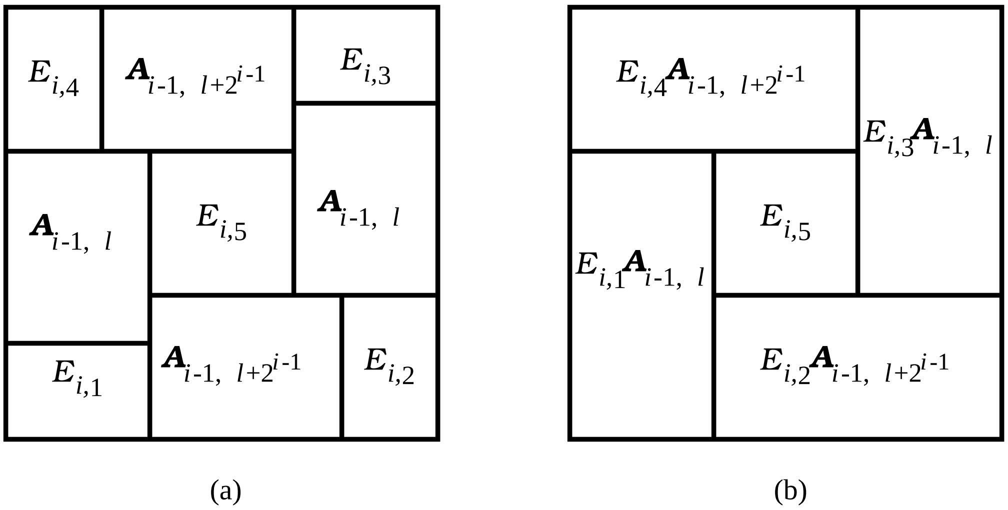


Figure 6.3: Construction of $A_{i,l}$ (a) and the wheel equivalent to $A_{i,l}$ (b).

Let us now determine b_i, the number of basic blocks in $A_{i,l}$. Obviously, we have the following recurrence:

$$\begin{cases} b_1 = 1 \\ b_i = 4b_{i-1} + 5, \quad \text{for } i > 1 \end{cases}$$

Solving this recurrence, we obtain $b_i = \frac{8}{3}4^{i-1} - \frac{5}{3}$. In particular, $b_m = \frac{8}{3}n^2 - \frac{5}{3}$.

We define a sequence of integers x_i, $i \geq 1$, as follows:

$$\begin{cases} x_1 = 0 \\ x_i = 2x_{i-1} + Q, \quad \text{for } i > 1 \end{cases}$$

Solving the recurrence, we obtain $x_i = (2^{i-1}-1)Q$. In particular, $x_m = (2^{m-1}-1)Q = (n-1)Q$.

The following result asserts that $A_{i,l}$ has the desired property.

Lemma 6.2 $L(A_{i,l}) = \{(x_i + s(I), x_i + s(I_{i,l} - I)) \mid I \in \mathcal{I}_{i,l}\}$.

Proof: (By induction on i)

Basis: For $i = 1$, the lemma simply follows from the definition of $A_{1,l}$.

Induction Step: By the induction hypothesis, we have

$$L(A_{i-1,l}) = \{(x_{i-1} + s(I_1), x_{i-1} + s(I_{i-1,l} - I_1)) \mid I_1 \in \mathcal{I}_{i-1,l}\}$$

$$L(A_{i-1,l+2^{i-1}}) = \{(x_{i-1} + s(I_2), x_{i-1} + s(I_{i-1,l+2^{i-1}} - I_2)) \mid I_2 \in \mathcal{I}_{i-1,l+2^{i-1}}\}$$

Let $E_{i,1}A_{i-1,l}$ denote the sub-floorplan formed by $E_{i,1}$ and $A_{i-1,l}$. Similarly, we define $E_{i,3}A_{i-1,l}$, $E_{i,2}A_{i-1,l+2^{i-1}}$, and $E_{i,4}A_{i-1,l+2^{i-1}}$. Then, we have

$$\begin{aligned} L(E_{i,1}A_{i-1,l}) &= L(E_{i,3}A_{i-1,l}) \\ &= \{(w(r), h(r) + Q) \mid r \in L(A_{i-1,l})\} \\ &= \{(x_{i-1} + s(I_1), x_{i-1} + Q + s(I_{i-1,l} - I_1)) \mid I_1 \in \mathcal{I}_{i-1,l}\} \end{aligned}$$

$$\begin{aligned} L(E_{i,2}A_{i-1,l+2^{i-1}}) &= L(E_{i,4}A_{i-1,l+2^{i-1}}) \\ &= \{(w(r) + Q, h(r)) \mid r \in L(A_{i-1,l+2^{i-1}})\} \\ &= \{(x_{i-1} + Q + s(I_2), x_{i-1} + s(I_{i-1,l+2^{i-1}} - I_2)) \mid I_2 \in \mathcal{I}_{i-1,l+2^{i-1}}\} \end{aligned}$$

By viewing $E_{i,1}A_{i-1,l}$, $E_{i,2}A_{i-1,l+2^{i-1}}$, $E_{i,3}A_{i-1,l}$, and $E_{i,4}A_{i-1,l+2^{i-1}}$ as basic blocks, $A_{i,l}$ is a wheel as shown in Figure 6.3(b). Moreover, it is easy to verify that the condition in Lemma 6.1

is satisfied. Thus, we have

$$\begin{aligned} L(A_{i,l}) &\subseteq \{(x_{i-1} + s(I_1) + x_{i-1} + Q + s(I_2), \\ &\qquad x_{i-1} + Q + s(I_{i-1,l} - I_1) + x_{i-1} + s(I_{i-1,l+2^{i-1}} - I_2)) \\ &\qquad\qquad \mid I_1 \in \mathcal{I}_{i-1,l}, I_2 \in \mathcal{I}_{i-1,l+2^{i-1}}\} \\ &= \{(x_i + s(I), x_i + s(I_{i,l} - I)) \mid I \in \mathcal{I}_{i,l}\} \ \ \text{(denoted } R). \end{aligned}$$

Since for any $r \in R$, $w(r) + h(r) = 2x_i + a_l + a_{l+1} + \cdots + a_{l+2^i-1}$ — independent of I, there is no redundant realization in R. Therefore, $L(A_{i,l}) = R$. □

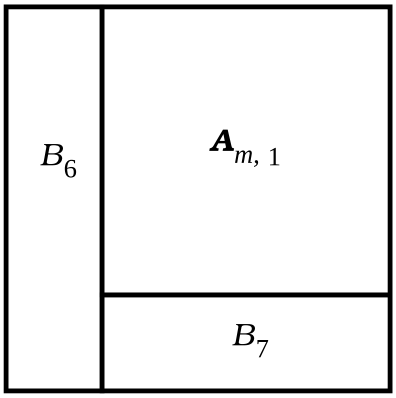


Figure 6.4: Construction of $F_{m,1}$.

Let $F_{m,1}$ be the floorplan formed by $A_{m,1}$ and two basic blocks B_6 and B_7 as shown in Figure 6.4, where B_6 has $(1, 1 + x_m + Q/2)$ as its only realization and B_7 has $(x_m + Q/2, 1)$ as its only realization. We reduce the generic instance of **2-1-PARTITION** to the instance of **AREAMIN-5** with $F = F_{m,1}$ and $A = (1 + x_m + Q/2)^2$. The reduction is polynomial because b_m is a polynomial in n, and x_m is a polynomial in n and Q. We now show that the instance of **2-1-PARTITION** has an *yes* answer iff $F_{m,1}$ has a realization with area less than or equal to $A = (1 + x_m + Q/2)^2$. We note first that $F_{m,1}$ has a realization with area less than or equal to A iff $A_{m,1}$ has a realization r with $w(r) \leq x_m + Q/2$ and $h(r) \leq x_m + Q/2$. Obviously, we can assume that r is nonredundant. By Lemma 6.2, there is an index set I in $\mathcal{I}_{m,1}$ such that

$$w(r) = x_m + s(I) \text{ and } h(r) = x_m + s(I_{m,1} - I) = x_m + Q - s(I).$$

Consequently, $w(r) \le x_m + Q/2$ and $h(r) \le x_m + Q/2$ iff $w(r) = x_m + Q/2$ and $h(r) = x_m + Q/2$, which in turn is equivalent to $s(I) = Q/2$, i.e., the instance of **2-1-PARTITION** has an *yes* answer.

According to the analysis above and the fact that $F_{m,1}$ is *node-balanced* [48], i.e., the height of the partition tree of $F_{m,1}$ is $O(\log L)$ where L denotes the number of blocks in $F_{m,1}$, we have the following result:

Theorem 6.1 *Assuming the dimensions of a realization of a block could be zero,* **AREAMIN-5** *is* **NP***-complete (even when the floorplans are balanced).* □

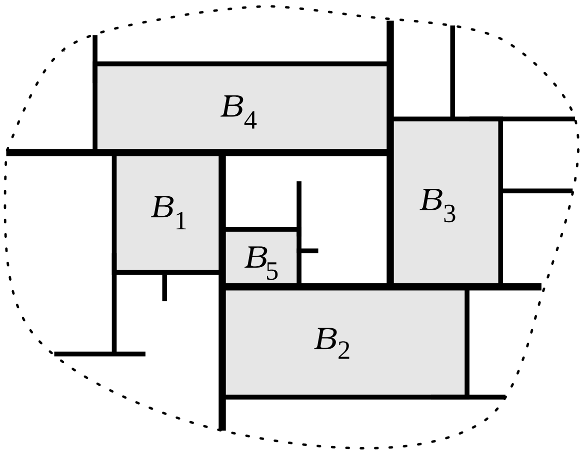


Figure 6.5: A partially drawn non-slicing partitioning pattern.

It was shown in [79] that any non-slicing pattern contains a wheel-type structure. See Figure 6.5. In Figure 6.5, if all the unshaded blocks have $(0, 0)$ as their only realizations, the pattern is equivalent to a wheel formed by the five shaded blocks (as far as the area is concerned). Also note that a non-slicing pattern can generate equivalent slices. For example, for the pattern in Figure 6.5, if all blocks except B_1 and B_4 have $(0, 0)$ as their only realizations, the pattern is equivalent to a horizontal slice formed by B_1 and B_4. Similarly, if all blocks except B_1 and B_2 have $(0, 0)$ as their only realizations, the pattern is equivalent to a vertical slice formed by B_1 and B_2. Therefore, from Theorem 6.1, we have

Theorem 6.2 *Assuming the dimensions of a realization of a block could be zero,* **AREAMIN-$\mathcal{P}$** *is* **NP***-complete if $\mathcal{P}$ contains a non-slicing pattern.* □

We now discuss the case in which the dimensions of a realization of any block must be positive integers. We will show that **AREAMIN-$\mathcal{P}$** is still **NP**-complete if $\mathcal{P}$ contains a non-slicing pattern. For this, we need to generalize Lemma 6.1. Consider the non-slicing pattern P shown in Figure 6.5. Again, we assume that B_1 and B_3 have the same set of nonredundant realizations (L_1), and B_2 and B_4 have the same set of nonredundant realizations (L_2). All other blocks have $(1,1)$ as their only realizations. Let p denote the number of blocks in P. (Note that p is a constant.) Let k_w denote the maximum of $|x|-2$ where x a full path in the V-graph of P which contains both B_1 and B_2, or both B_3 and B_4, and $|x|$ is the number of edges in x. Similarly, let k_h denote the maximum of $|y|-2$ where y a full path in the H-graph of P which contains both B_1 and B_4, or both B_2 and B_3. The following is a generalization of Lemma 6.1:

Lemma 6.3 *If (i) $w+p \leq w'$ and $h \geq h'+p$ for any (w,h) in L_1 and (w',h') in L_2, (ii) $w \geq p$ and $h \geq p$ for any (w,h) in L_1 or L_2, and (iii) $w_1 \geq w_2$ and $h_1 \leq h_2$ imply that $w_1 \geq w_2+p$ and $h_1+p \leq h_2$ if (w_1,h_1) and (w_2,h_2) are both in L_1 or in L_2, then, a nonredundant realization of P must select the same realization for B_1 and B_3, and the same realization for B_2 and B_4. In other words,*

$$L(P) \subseteq \{(w+w'+k_w, h+h'+k_h) \mid (w,h) \in L_1, (w',h') \in L_2\}.$$

The proof of this lemma is very similar to that of Lemma 6.1. We only give the intuition here. The details are left to the reader. Condition (i) is introduced so that the dimensions of the realizations in L_1 and that of those in L_2 are far apart numerically. This guarantees that the realizations for the blocks enclosed by B_1, B_2, B_3, and B_4 will have no effect on the dimensions of any nonredundant realization of P. Condition (ii) together with Condition (i) guarantees that any full path that determines the width of a realization of P must contain either both B_1 and B_2, or both B_3 and B_4. Similarly, any full path that determines the height of a realization of P must contain either both B_1 and B_4, or both B_2 and B_3. Condition (iii) is introduced to assure that the dimensions of any two realizations from the same set are far apart numerically. This guarantees that in a nonredundant realization of P, B_1 and B_3 must have the same realization, and B_2 and B_4 much have the same realization in spite of the fringe effect of those $(1,1)$ realizations of the other blocks in the pattern.

2-1-PARTITION is still the problem used in the polynomial reduction. The reduction is similar to the one used in showing the **NP**-completeness of **AREAMIN-5** we will use the same names for the various components in the reduction. In the reduction, we will assume $\mathcal{P}$ contains both vertical slices and horizontal slices. Later on, we will remark on how this assumption can be removed.

To make sure that conditions (ii) and (iii) in Lemma 6.3 are satisfied, we "magnify" the dimensions of the realizations of the blocks in the previous reduction by a factor p. To enforce condition (i), we add p to the width or height of the realization of each enforcement block $E_{i,t}$. The following is the detailed construction:

(1) $A_{1,l}$, where $1 \leq l \leq 2^m - 1$, is a one block floorplan. The set of realizations of the block is $\{(pa_l, pa_{l+1}), (pa_{l+1}, pa_l)\}$.

(2) $A_{i,l}$, where $2 \leq i \leq m$ and $1 \leq l \leq 2^m - 2^i + 1$, is recursively defined by $A_{i-1,l}$ and $A_{i-1,l+2^{i-1}}$ as shown in Figure 6.6. Here, $E_{i,1}, E_{i,2}, E_{i,3}, E_{i,4}$ are basic blocks with $L(E_{i,1}) = L(E_{i,3}) = \{(1, pQ + p + k_w)\}$, $L(E_{i,2}) = L(E_{i,4}) = \{(pQ + p + k_h, 1)\}$. All other blocks have $(1,1)$ as their only realizations. $E_{i,t}$'s are introduced to enforce condition (i) in Lemma 6.3, which is a generalization of the condition in Lemma 6.1.

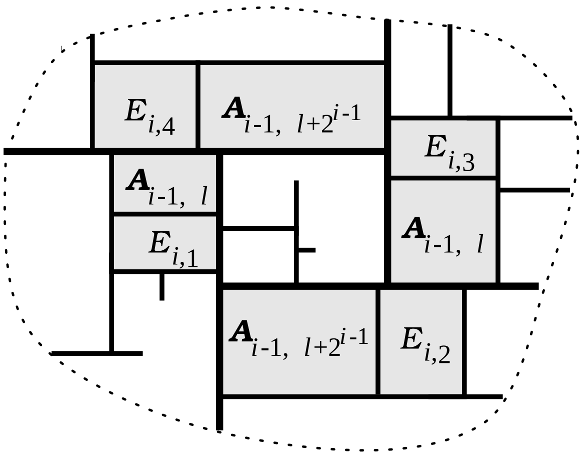


Figure 6.6: Construction of $A_{i,l}$ from a non-slicing pattern and the slices.

We redefine x_i as follows to accommodate the magnification effect:

$$\begin{cases} x_1 = 0 \\ x_i = 2x_{i-1} + pQ + p + k_w + k_h, \quad \text{for } i > 1 \end{cases}$$

The following result corresponds to Lemma 6.2 and can be proven similarly.

Lemma 6.4 $L(A_{i,l}) = \{(x_i + ps(I), x_i + ps(I_{i,l} - I)) \mid I \in \mathcal{I}_{i,l}\}$. $\square$

The remaining part of the reduction is the same as that for **AREAMIN-5** except that Q is replaced by pQ.

Finally, we discuss how to remove the assumption that $\mathcal{P}$ contains both vertical slices and horizontal slices. As was mentioned earlier, a non-slicing pattern can be turned into a slice by letting some of the blocks have $(0,0)$ as their only realizations. If zero dimensions are not allowed, we can let them have $(1,1)$ as their only realizations instead. By using the same ideas of magnifying the dimensions and adding sufficiently large constants to them, any non-slicing pattern can turned into a slice. The details are omitted.

In summary, we have shown that if $\mathcal{P}$ contains at least one non-slicing pattern, **AREAMIN-**$\mathcal{P}$ is **NP**-complete no matter whether zero dimensions are allowed or not. In other words, slicing floorplans are the only type of floorplans for which area minimization can be accomplished in polynomial time, assuming $P \neq NP$.

6.3 Nonredundant realizations of a wheel

In this section, we study the problem of determining the nonredundant realizations of a wheel. To be specific, the problem we will focus on is: *For a wheel W, determine $L(W)$, the set of nonredundant realizations of W, for given $L(B_i)$, the set of nonredundant realizations of B_i, $1 \leq i \leq 5$.* Let $k_i = |L(B_i)|$ for $1 \leq i \leq 5$, and $k = \max_i k_i$. We will present an $O(k^2 \log k)$ time and $O(k^2)$ space algorithm. The algorithm will be used in the pseudo-polynomial area minimization algorithm presented in the next section.

We assume that the realizations of a block are sorted in the order of *increasing height* (in the order of decreasing width accordingly). If they are not sorted, we can sort them before the algorithm starts, which can be done in $O(k \log k)$ time. To simplify the presentation, the realizations of a block will be called *tiles* in this section, and a realization r that selects tile b_i for B_i for $1 \le i \le 5$ will be denoted by $r = b_1 b_2 b_3 b_4 b_5$.

To generate the set of nonredundant realizations $L(W)$, our strategy is to first generate R, a superset of $L(W)$. R may contain some redundant realizations. We then remove the redundant ones from R to obtain $L(W)$, which can be done by sorting the realizations in R as described in Chapter 2.

Notice that for a realization $r = b_1 b_2 b_3 b_4 b_5$, $w(r)$ is determined by two or three blocks. So is $h(r)$. Depending on the blocks which determine the dimensions of the realizations, the nonredundant realizations can be classified into nine classes:

- Classes 1–4: Both $w(r)$ and $h(r)$ are determined by two blocks:

$$\begin{aligned} w(r) &= w(b_1) + w(b_2), \quad h(r) = h(b_2) + h(b_3); \\ w(r) &= w(b_1) + w(b_2), \quad h(r) = h(b_1) + h(b_4); \\ w(r) &= w(b_3) + w(b_4), \quad h(r) = h(b_1) + h(b_4); \\ w(r) &= w(b_3) + w(b_4), \quad h(r) = h(b_2) + h(b_3). \end{aligned}$$

- Classes 5–6: $w(r)$ is determined by two blocks and $h(r)$ is determined by three blocks:

$$\begin{aligned} w(r) &= w(b_1) + w(b_2), \quad h(r) = h(b_2) + h(b_5) + h(b_4); \\ w(r) &= w(b_3) + w(b_4), \quad h(r) = h(b_2) + h(b_5) + h(b_4). \end{aligned}$$

- Classes 7–8: $w(r)$ is determined by three blocks and $h(r)$ is determined by two blocks:

$$\begin{aligned} w(r) &= w(b_1) + w(b_5) + w(b_3), \quad h(r) = h(b_1) + h(b_4); \\ w(r) &= w(b_1) + w(b_5) + w(b_3), \quad h(r) = h(b_2) + h(b_3). \end{aligned}$$

- Class 9: Both $w(r)$ and $h(r)$ are determined by three blocks:

$$w(r) = w(b_1) + w(b_5) + w(b_3), h(r) = h(b_2) + h(b_5) + h(b_4).$$

In general, these classes are not necessarily disjoint. In describing the algorithm, we will assume that these classes are disjoint. Later, we will slightly modify the algorithm to make it work in the general situation.

Our approach is to generate a superset for each of the classes. R is simply the union of the supersets. Notice that Classes 1–4 are similar because they can be turned into each other by rotating W properly. Classes 5-8 are similar for the same reason. Therefore, all we need to do is to develop procedures for finding the supersets for Classes 1, 5, and 9, and each of the procedures has $O(k^2 \log k)$ time and $O(k^2)$ space costs.

Class 1. Nonredundant realizations the width of which is determined by B_1 and B_2, and the height of which is determined by B_2 and B_3. See Figure 6.7.

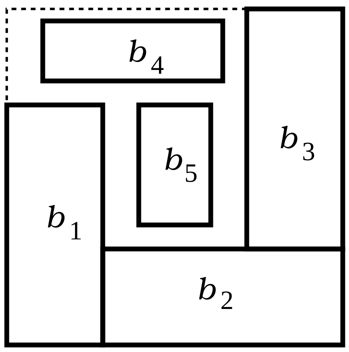


Figure 6.7: Illustration of a realization in Class 1.

We consider all the pairs of tiles $b_1 \in L(B_1)$ and $b_2 \in L(B_2)$, and for each pair b_1b_2, find a realization with minimum height. To be precise, for each pair b_1b_2, we want to find $b_3 \in L(B_3)$, $b_4 \in L(B_4)$, and $b_5 \in L(B_5)$, such that $h(b_3)$ is minimized subject to $w(r) = w(b_1) + w(b_2)$ and $h(r) = h(b_2) + h(b_3)$, where $r = b_1b_2b_3b_4b_5$. We call such an r the *I-support* for b_1b_2. It should be noted that for some pair b_1b_2, its I-support may not exist, or may be redundant. However, any nonredundant realization in Class 1 must be an I-support for some b_1b_2. Therefore, to find a superset of Class 1, we can simply find all the I-supports.

To find all the I-supports efficiently, observe that if we fix the tile for B_2 and increase the height of the tile for B_1, then the height of the tile for B_3 in the I-supports cannot decrease. In other words, if $b_1b_2b_3b_4b_5$ is the I-support for b_1b_2 and $b'_1b_2b'_3b'_4b'_5$ is the I-support for b'_1b_2 with $h(b_1) < h(b'_1)$, then $h(b_3) \leq h(b'_3)$. This is because $r = b_1b_2b'_3b'_4b'_5$ satisfies $w(r) = w(b_1) + w(b_2)$ and $h(r) = h(b_2)+h(b'_3)$. By definition, the I-support for b_1b_2 is the realization with the smallest height for B_3 while having these properties. Hence, $h(b_3) \leq h(b'_3)$.

The procedure enumerates the tiles in $L(B_2)$. For each $b_2 \in L(B_2)$, the I-supports for b_2 and the tiles in $L(B_1)$ are determined in the order of increasing height of the tiles in $L(B_1)$. For each pair $b_1 b_2$, the tiles in $L(B_3)$ are searched in the order of increasing height, starting from the one in the preceding I-support, to find b_3. The following is an outline of the procedure for Class 1.

```
begin Class 1
    for each b2 ∈ L(B2) do
        b3 ← first tile in L(B3);
        for each b1 ∈ L(B1) in increasing height do
            while search45(b1, b2, b3; b4, b5) = fail do
                if b3 is the last tile stop;
                b3 ← next tile in L(B3);
            end while
            R ← R ∪ {b1b2b3b4b5};
        end for
    end for
end Class 1
```

Subroutine $search45(b_1, b_2, b_3; b_4, b_5)$ finds $b_4 \in L(B_4)$ and $b_5 \in L(B_5)$ such that $b_4 b_5$ can be fitted into the L-shaped space left by $b_1 b_2 b_3$. To be specific, it finds b_4 and b_5 that satisfy the following conditions:

$$
\begin{aligned}
&w(b_4) = \max\{w(b) \mid b \in L(B_4), w(b) + w(b_3) < w(b_1) + w(b_2)\},\\
&w(b_5) = \max\{w(b) \mid b \in L(B_5), w(b) + w(b_3) < w(b_2)\},\\
&h(b_4) + h(b_5) < h(b_3),\\
&h(b_4) + h(b_1) < h(b_2) + h(b_3).
\end{aligned}
$$

If no such $b_4 b_5$ are found, $search45$ returns $fail$.

Clearly, $search45(b_1, b_2, b_3; b_4, b_5)$ can be done in $O(\log k)$ time and $O(1)$ space since $L(B_4)$ and $L(B_5)$ are sorted. Therefore, the time cost of the procedure is $O(k_2(k_1 + k_3)\log k)$. The space cost as well as the number of realizations generated is $O(k_1 k_2)$.

Class 5. Nonredundant realizations the width of which is determined by B_1 and B_2, and the height of which is determined by B_2, B_5, and B_4. See Figure 6.8.

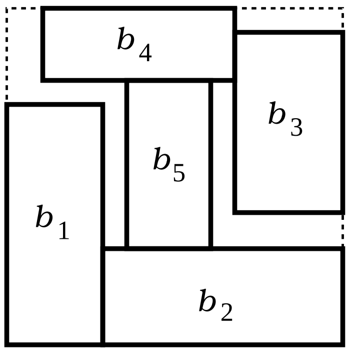


Figure 6.8: Illustration of a realization in Class 5.

We claim that a nonredundant realization r in Class 5 that contains $b_5 \in L(B_5)$ and $b_2 \in L(B_2)$ must contain $b_1 \in L(B_1)$ such that:

$$h(b_1) = \max\{h(b) \mid b \in L(B_1), h(b) < h(b_2) + h(b_5)\}. \tag{6.1}$$

Because otherwise, if r uses $b_1' \in L(B_1)$, where b_1' is not the b_1 defined above, then replacing b_1' by b_1 will reduce the width without increasing the height of r, which means that r is redundant.

We can assume that a nonredundant realization r in Class 5 that contains $b_2b_4b_5$ must contain $b_3 \in L(B_3)$ such that:

$$h(b_3) = \max\{h(b) \mid b \in L(B_3), h(b) < h(b_5) + h(b_4)\}. \tag{6.2}$$

For otherwise, suppose r uses $b_3' \in L(B_3)$, where b_3' is not the b_3 defined above, then replacing b_3' by b_3 will not increase the width nor the height of r.

We consider all the pairs of tiles $b_2 \in L(B_2)$ and $b_5 \in L(B_5)$, and for each pair b_2b_5, find a realization with minimum height. To be precise, for each pair b_2b_5, we want to find $b_1 \in L(B_1)$, $b_3 \in L(B_3)$, and $b_4 \in L(B_4)$, such that $h(b_4)$ is minimized subject to (6.1), (6.2), $w(r) = w(b_1) + w(b_2)$, and $h(r) = h(b_2) + h(b_5) + h(b_4)$, where $r = b_1b_2b_3b_4b_5$. We call such an r the *II-support* for b_2b_5. It should be noted that for some pair b_2b_5, its II-support may not exist. However, any nonredundant realization in Class 5 must be an II-support for some b_2b_5. Therefore, to find a superset of Class 5, we can simply find all the II-supports.

To find all the II-supports efficiently, observe that if we fix the tile for B_5 and increase the height of the tile for B_2, then neither of the heights of the tiles for B_1, B_4, and B_3 decrease in the II-supports. In other words, if $b_1b_2b_3b_4b_5$ is the II-support for b_2b_5 and $b'_1b'_2b'_3b'_4b_5$ is the II-support for b'_2b_5 with $h(b_2) < h(b'_2)$, then $h(b_1) \leq h(b'_1)$, $h(b_4) \leq h(b'_4)$, and $h(b_3) \leq h(b'_3)$. From (6.1), it is obvious that $h(b_1) \leq h(b'_1)$. If $h(b_4) \leq h(b'_4)$, from (6.2), obviously $h(b_3) \leq h(b'_3)$. We now only need to show $h(b_4) \leq h(b'_4)$. It can be verified that $r = b_1b_2b'_3b'_4b_5$ satisfies that $w(r) = w(b_1) + w(b_2)$ and $h(r) = h(b_2) + h(b_5) + h(b'_4)$. By definition, the II-support for b_2b_5 is the realization with the smallest height for B_4 while having these properties. As a result $h(b_4) \leq h(b'_4)$.

The procedure for Class 5 now is obvious. We enumerate the tiles in $L(B_5)$. For each $b_5 \in L(B_5)$, we determine the II-supports for b_5 and the tiles in $L(B_2)$ in the order of increasing height according to the tiles in $L(B_2)$, through determining b_1, b_4, and b_3 by linearly scanning $L(B_1), L(B_4)$, and $L(B_3)$, all in the order of increasing height, starting from those in the preceding II-support. The following is an outline of the procedure.

```
begin Class 5
    for each b5 ∈ L(B5) do
        b1 ← first tile in L(B1);
        b3 ← first tile in L(B3);
        b4 ← first tile in L(B4);
        for each b2 ∈ L(B2) in increasing height do
            if Lsearch1(b2, b5; b1) = fail goto next;
            while C(b1, b2, b3, b4, b5) = false do
                if b4 is the last tile stop;
                b4 ← next tile in L(B4);
            end while
            R ← R ∪ {b1b2b3b4b5};
    next:
        end for
    end for
end Class 5
```

Subroutine $Lsearch1(b_2, b_5; b_1)$ linearly scans $L(B_1)$ to find b_1 that satisfies (6.1), starting from the current b_1. It returns $fail$ if $h(b_1) \geq h(b_2) + h(b_5)$ for the current b_1, because then no tile in $L(B_1)$ can satisfy (6.1). $Lsearch3(b_4, b_5; b_3)$ is similar defined except it finds b_3 that satisfies (6.2). $C(b_1, b_2, b_3, b_4, b_5)$ first finds b_3 that satisfies (6.2) by calling $Lsearch3(b_4, b_5; b_3)$.

If $Lsearch3$ returns $fail$, $C(b_1, b_2, b_3, b_4, b_5)$ is $false$; otherwise, it checks whether $b_4 b_3$ can be fitted into the L-shaped space left by $b_1 b_2 b_5$. To be specific, $C(b_1, b_2, b_3, b_4, b_5)$ is the following procedure:

if $Lsearch3(b_4, b_5; b_3) = fail$ **return** $false$;
if $w(b_4) + w(b_3) \geq w(b_1) + w(b_2)$ **return** $false$;
if $w(b_5) + w(b_3) \geq w(b_2)$ **return** $false$;
return $true$

The time cost of the procedure is obviously $O(k_5(k_2 + k_1 + k_4 + k_3))$. The space cost as well as the number of realizations generated is $O(k_5 k_2)$.

Class 9. Nonredundant realizations the width of which is determined by B_1, B_5, and B_3, and the height of which is determined by B_2, B_5, and B_4. See Figure 6.9.

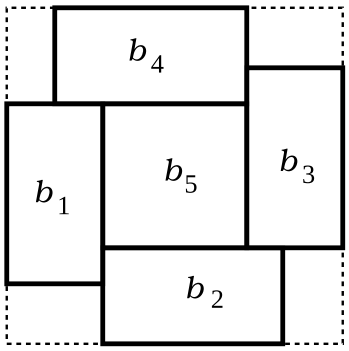


Figure 6.9: Illustration of a realization in Class 9.

By the same argument as the one used to show (6.1) in Class 5, a nonredundant realization in Class 9 that contains $b_5 \in L(B_5)$ and $b_2 \in L(B_2)$ must contain $b_1 \in L(B_1)$ such that:

$$h(b_1) = \max\{h(b) \mid b \in L(B_1), h(b) < h(b_2) + h(b_5)\}. \tag{6.3}$$

Similarly, a nonredundant realization in Class 9 that contains $b_5 b_2 b_1$ must contain $b_4 \in L(B_4)$ such that:

$$w(b_4) = \max\{w(b) \mid b \in L(B_4), w(b) < w(b_1) + w(b_5)\}, \tag{6.4}$$

and a nonredundant realization in Class 9 that contains $b_5b_2b_1b_4$ must contain $b_3 \in L(B_3)$ such that:

$$h(b_3) = \max\{h(b) \mid b \in L(B_3), h(b) < h(b_5) + h(b_4)\}. \tag{6.5}$$

A realization $b_1b_2b_3b_4b_5$ satisfying (6.3), (6.4), and (6.5) is called the *III-support* for b_5b_2. To find a superset of Class 9, we can simply find all the III-supports.

To find the III-supports efficiently, observe that if the height of the tile for B_2 increases, then neither of the heights of the tiles for B_1, B_4, and B_3 can decrease. The procedure is now obvious. The following is an outline.

```
begin Class 9
    for each b5 ∈ L(B5) do
        b1 ← first tile in L(B1);
        b3 ← first tile in L(B3);
        b4 ← first tile in L(B4);
        for each b2 ∈ L(B2) in increasing height do
            if Lsearch1(b2, b5; b1) = fail goto next;
            if Lsearch4(b1, b5; b4) = fail goto next;
            if Lsearch3(b4, b5; b3) = fail goto next;
            R ← R ∪ {b1b2b3b4b5};
    next:
        end for
    end for
end Class 9
```

Subroutines *Lsearch*1 and *Lsearch*3 have been described in Class 5. subroutine *Lsearch*4 is similarly defined except it finds b_4 that satisfies (6.5). The time cost of the procedure is $O(k_5(k_2 + k_1 + k_4 + k_3))$. The space cost as well as the number of realizations generated is $O(k_5k_2)$.

Combining all the procedures, we have an algorithm for determining $L(W)$ in $O((\sum_{i \neq j} k_ik_j) \log k)$ time and $O(\sum_{i \neq j} k_ik_j)$ space, provided that the nine classes are disjoint.

We now show how to modify the algorithm to remove the disjointness requirement. The modification is based on the idea of symbolic perturbation introduced in computational geometry [30]. To be specific, imagine that ϵ^i is added to the width and height of each tile in $L(B_i)$

for $1 \leq i \leq 5$, where $\epsilon > 0$. Let $r(\epsilon)$ denote the perturbed realization corresponding to r. By choosing a sufficiently small $\epsilon > 0$, we have

(i) No new dominance is introduced among the perturbed realizations. That is, if $w(r_1) < w(r_2)$ and $h(r_1) > h(r_2)$, then $w(r_1(\epsilon)) < w(r_2(\epsilon))$ and $h(r_1(\epsilon)) > h(r_2(\epsilon))$.

(ii) No perturbed realization is in more than one class. That is, for any realization r, $w(r(\epsilon))$ is determined by a unique set of blocks, so is $h(r(\epsilon))$.

It is obvious from (i) that the set of realizations corresponding to the nonredundant perturbed realizations is a superset of $L(W)$.

We do not actually perform the perturbation and apply the previous algorithm to the perturbed tiles. We only use the perturbation to guide the modification of the definitions of I-, II-, and III-supports as well as the algorithm accordingly.

Let us take the III-supports as an example. Consider (6.4) in the definition of III-supports. Its perturbed form is the following:

$$h(b_4) = \max\{h(b) \mid b \in L(B_4), h(b) + \epsilon^4 < h(b_1) + h(b_5) + \epsilon^1 + \epsilon^5\}.$$

For a sufficiently small $\epsilon > 0$, the ϵ terms will have any effect only if there is a tile b in $L(B_4)$ such that $h(b) = h(b_1) + h(b_5)$. Since $\epsilon^4 < \epsilon^1 + \epsilon^5$ for sufficiently small $\epsilon > 0$, such a tile b will satisfy the condition in the perturbed form of (6.4). Instead of considering the perturbed tiles, we simply modify (6.4) on the original tiles by replacing $<$ by $\leq$ as shown in the following:

$$h(b_4) = \max\{h(b) \mid b \in L(B_4), h(b) \leq h(b_1) + h(b_5)\}.$$

On the other hand, the perturbed form for (6.3) is the following:

$$h(b_1) = \max\{h(b) \mid b \in L(B_1), h(b) + \epsilon^1 < h(b_2) + h(b_5) + \epsilon^2 + \epsilon^5\}.$$

Since $\epsilon^1 > \epsilon^2 + \epsilon^5$, the ϵ terms will have no effect at all. Therefore, we still use (6.3) on the original tiles. By the same argument, we do not modify (6.5). As for the procedure for Class 9, we only need to modify *Lsearch4* to take into account the modification to (6.4).

Using a similar analysis, we can modify the definitions of I- and II-supports by properly replacing some of the $<$'s by $\leq$'s, and modify the procedures accordingly.

Theorem 6.3 *There is an algorithm for determining $L(W)$ in $O((\sum_{i \neq j} k_i k_j) \log k) = O(k^2 \log k)$ time and $O(\sum_{i \neq j} k_i k_j) = O(k^2)$ space.*

The $O(k^2)$ space bound is optimal, since the output size can be $\Omega(k^2)$. Whether the $O(k^2 \log k)$ time bound can be improved is related to the long-standing open problem of *sorting* $X+Y$, that is, whether the set $\{x_i + y_j \mid i, j = 1, \ldots, k\}$ can be sorted in $o(k^2 \log k)$ time [35]. In the following, we reduce the problem of sorting $X+Y$ to that of computing the nonredundant realizations in sorted order:

Given $X = \{x_1, \ldots, x_k\}$ and $Y = \{y_1, \ldots, y_k\}$. In a wheel W, let $L(B_5) = \{(0,0)\}$ and

$$L(B_1) = L(B_3) = \{(K + x_i, 2K - x_i) \mid i = 1, \ldots, k\},$$
$$L(B_2) = L(B_4) = \{(2K + y_i, K - y_i) \mid i = 1, \ldots, k\},$$

where $K = \max\{x_1, \ldots, x_k, y_1, \ldots, y_k\}$. From Lemma 6.1 it can be shown that

$$L(W) = \{(3K + (x_i + y_j), 3K - (x_i + y_j)) \mid i, j = 1, \cdots, k\}.$$

Therefore to produce $L(W)$ in sorted order is the same as to sort $X+Y$.

To generate $L(W)$, it seems that the order of the realizations in $L(W)$ must be determined. Another related open problem is whether the $O(k^2 \log k)$ time bound can be improved if we only want to obtain a superset of $L(W)$.

6.4 A pseudo-polynomial area minimization algorithm

In this section, we present an $O((nM)^2 \log(nM))$ time and $O(n^2 M)$ space algorithm for area minimization for a hierarchical floorplan of order-5, where n is the number of basic blocks and M is an upper bound on the dimensions of the realizations of the blocks. In fact, our algorithm generates all the nonredundant realizations instead of only a minimum area realization of the floorplan. Thus, the chip designer is afforded the freedom to choose among all the minimum area realizations one with a preferred aspect ratio. The algorithm can be modified easily to output only one minimum area realization with the same time and space costs.

Our algorithm uses the hierarchical approach found in many area minimization algorithms [19, 62, 83]. Let F be a hierarchical floorplan, and P be the pattern that partitions F into sub-floorplans $F_1, F_2, \ldots, F_t$. Let B_i be the block in P that corresponds to F_i for $1 \leq i \leq t$. The algorithm recursively determines the nonredundant realizations of F_i for $1 \leq i \leq t$, and then combines them using an algorithm for determining the nonredundant realizations of pattern P.

AreaMin(F)
    Input: A hierarchical floorplan of order-5 F, and the sets
        of nonredundant realizations of the basic blocks.
    Output: the set of nonredundant realizations of F.
    begin
        if F is a basic block **then**
            return the set of nonredundant realizations of the block;
        else $P \leftarrow$ the pattern that partitions F into sub-floorplans
            $F_1, \ldots, F_t$ with F_i corresponding to B_i in P;
            for $i = 1$ **to** t **do** $L(B_i) \leftarrow$ **AreaMin**(F_i);
            call an algorithm for P to determine $L(P)$;
            return $L(P)$
    end

In **AreaMin**, area minimization for hierarchical floorplans of order-5 is reduced to that of determining $L(P)$, the set of nonredundant realizations of pattern P, where P is a slice or a wheel. If P is a slice, we use the algorithm in [78] to determine $L(P)$. If P is a wheel, we use the algorithm in the previous section to determine $L(P)$.

We now analyze the costs of **AreaMin**. To consider the worst case, we assume that F is constructed by using wheels only. Let n be the number of basic blocks and M be an upper bound on the dimensions of the realizations of the basic blocks. A realization of a sub-floorplan that contains m basic blocks can have a width at most mM. As a result, the number of nonredundant realizations of the sub-floorplan is at most mM since no two nonredundant realizations have the same width. From Theorem 6.3, there exists a constant C_1 such that the time cost of the algorithm for wheels is bounded by $C_1(\sum_{i \neq j} k_i k_j) \log k$. Now we show by induction that the time cost $T(n)$ of **AreaMin** is bounded by $C_1(nM)^2 \log(nM)$.

$$T(n) \quad \leq \quad \max_{n_1 + \cdots + n_5 = n} \left\{ \sum_{i=1}^{5} T(n_i) + C_1 \left(\sum_{i \neq j} (n_i M)(n_j M) \right) \log(nM) \right\}$$

$$
\begin{aligned}
&\leq \max_{n_1+\cdots+n_5=n} \left\{ \sum_{i=1}^{5} C_1 (n_i M)^2 \log(n_i M) + C_1 \left(\sum_{i \neq j} (n_i M)(n_j M) \right) \log(nM) \right\} \\
&\leq \max_{n_1+\cdots+n_5=n} \left\{ \left(\sum_{i=1}^{5} n_i^2 + \sum_{i \neq j} n_i n_j \right) C_1 M^2 \log(nM) \right\} \\
&= \max_{n_1+\cdots+n_5=n} \left\{ \left(\sum_{i=1}^{5} n_i \right)^2 C_1 M^2 \log(nM) \right\} \\
&= C_1 (nM)^2 \log(nM).
\end{aligned}
$$

Therefore, $T(n) = O((nM)^2 \log(nM))$, a polynomial in terms of n and M.

A direct application of the algorithm for wheels will result in **AreaMin** having space cost $O((nM)^2)$. Here we introduce a modification of the algorithm for wheels by retaining in R only realizations with different widths. R is organized as a balanced binary search tree according to the widths of the realizations. When a realization is generated (by one of the procedures), it is added to the tree only if either there is no realization in R with the same width, or the realization with the same width in R has a larger height. In the latter situation, the realization with a larger height is removed from R. It is easy to see that the modification does not change the time and space costs of the algorithm.

Consequently, in determining $L(P)$ in **AreaMin**, R contains at most nM realizations since no two realizations in R have the same width. Therefore, there exists a constant C_2 such that the space cost of this step is bounded by $C_2 nM$. Note that to determine a nonredundant realization of F in terms of the realizations of the basic blocks, the nonredundant realizations of all the intermediate sub-floorplans have to be kept. Now we show that by induction that the space cost $S(n)$ of **AreaMin** is bounded by $C_2 n^2 M$.

$$
\begin{aligned}
S(n) &\leq \max_{n_1+\cdots+n_5=n} \left\{ \sum_{i=1}^{5} S(n_i) + C_2 nM \right\} \\
&\leq \max_{n_1+\cdots+n_5=n} \left\{ \sum_{i=1}^{5} C_2 n_i^2 M + C_2 nM \right\} \\
&\leq C_2 n^2 M.
\end{aligned}
$$

Therefore $S(n) = O(n^2 M)$.

Theorem 6.4 *Given a hierarchical floorplan of order-5, let n be the number of basic blocks and M be an upper bound on the dimensions of the realizations of the blocks, the nonredundant realizations of the floorplan can be generated in $O((nM)^2 \log(nM))$ time and $O(n^2 M)$ space.*

Remarks: (1) Wang and Wong [83] have shown that the number of nonredundant realizations of a hierarchical floorplan of order-5 can be exponential in terms of the input size. This is also evident from Lemma 6.2. As a result, this algorithm is exponential in the worst case. (2) As was pointed out earlier, a sub-floorplan with m basic blocks has at most mM nonredundant realizations. This fact can be used to modify the algorithm for wheels so that the time cost of **AreaMin** is reduced to $O((nM)^2)$. However, the algorithm for wheels itself will become pseudo-polynomial also. (3) **AreaMin** can be applied to the case in which some realizations have dimensions with real values, without any modifications. The algorithm, however, is not pseudo-polynomial anymore, but it can be converted into an approximation scheme by the rounding and scaling techniques proposed in [82].

6.5 Experimental results

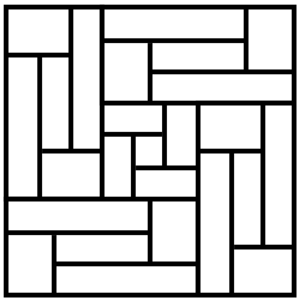


Figure 6.10: The floorplan used in EX1 to EX5.

AreaMin was implemented in the C language and executed on a SUN SPARCstation 10. We ran the program on a set of examples, most of them are from the literature: EX1 to EX5 are taken from [83]. They all use the floorplan shown in Figure 6.10. For EX1 each block has three realizations: $(4,1)$, $(2,2)$, and $(1,4)$. For EX2 each block has four realizations: $(6,1)$, $(3,2)$, $(2,3)$, and $(1,6)$. For EX3 each block has five realizations: $(16,1)$, $(8,2)$, $(4,4)$, $(2,8)$, and $(1,16)$. For EX4 each block has six realizations: $(12,1)$, $(6,2)$, $(4,3)$, $(3,4)$, $(2,6)$, and $(1,12)$. For EX5 each block has eight realizations: $(24,1)$, $(12,2)$, $(8,3)$, $(6,4)$, $(4,6)$, $(3,8)$,

$(2, 12)$, and $(1, 24)$. EX6 is the 24-block floorplan together with the sets of realizations in [85] (also see [83]). EX7 to EX12 are derived from EX1 to EX6, respectively. EX7 is constructed by substituting each of the five blocks of a wheel with a copy of EX1. EX8 to EX12 are constructed in the same way from EX2 to EX6.

test example	**# blocks**	**# realizations**			**time**
		total	**examined**	**final**	(sec)
EX1	25	3^{25}	168	5	0.02
EX2	25	4^{25}	365	8	0.03
EX3	25	5^{25}	776	17	0.13
EX4	25	6^{25}	994	17	0.18
EX5	25	8^{25}	2031	29	0.43
EX6	24	2.03×10^{16}	441	28	0.08
EX7	125	3^{125}	865	5	0.12
EX8	125	4^{125}	1930	8	0.27
EX9	125	5^{125}	4730	27	1.03
EX10	125	6^{125}	5723	23	1.18
EX11	125	8^{125}	12785	43	3.28
EX12	120	3.45×10^{81}	2382	53	1.23

Table 6.1: Experimental results.

Table 6.1 shows our experimental results. Column **total** lists the total number of possible realizations for each example. Column **examined** lists the total number of realizations of the sub-floorplans generated by **AreaMin**, as it reflects the number of realizations examined by **AreaMin**. Column **final** lists the number of nonredundant realizations for each example. From the table it is clear that **AreaMin** prunes a large number of redundant realizations. For example, although EX11 has 8^{125} possible realizations, our algorithm examined only 12785 realizations to determine the example has 43 nonredundant realizations, and it took only 3.28 seconds on a SUN SPARCstation 10. The running times for the other 11 examples are all under two seconds as shown in Table 6.1. The experimental results clearly show that the algorithm is very fast.

Chapter 7

Graph Constraint Reduction in Compaction

Compaction is the process of producing an area-efficient physical layout from a symbolic layout while enforcing the separation (design rules), electrical connectivity, and user-specified requirements. It is an active area of research in automatic VLSI layout design [4, 10, 28, 41, 44, 50, 54, 61, 70, 76, 92]. Comprehensive reviews of the subject can be found in [8, 18, 86].

The symbolic layout compaction problem is often formulated as a linear program (LP). Clearly, the size of the LP relates directly to the execution time and memory usage required to solve the corresponding compaction problem. As the complexity of layouts increases, the corresponding increase in the execution time and memory usage becomes a severe problem in many compaction systems [8]. As a result, it is desirable to reduce the size of the LP as was demonstrated in [4, 29, 44, 92]. Most of the constraints in the LP are derived directly or indirectly from physical separation and electrical connectivity requirements which can be expressed as "graph constraints". In this chapter we study the graph constraint reduction problem, i.e., the problem of producing, for a given system of graph constraints, an equivalent system with the fewest constraints.

Dutt and Lo gave a formulation of the graph constraint reduction problem and presented a heuristic algorithm for its solution [29]. We observe that their formulation not only renders the problem to be *NP-hard*, but also is over restrictive in the sense that the maximum possible reduction is not always attainable. In this chapter, We shall propose a new formulation that always achieves the maximum possible reduction. We further present a *polynomial-time* algorithm for our problem formulation.

It is obvious that constraint reduction usually does not help for graph-based leaf cell compaction. However, for the compaction of large hierarchical cells and compaction with general constraints involved, constraint reduction can be very beneficial. In the remainder of this section, we present three specific ways in which constraint reduction can be applied to symbolic layout compaction.

A conventional hierarchical pitchmatching constraint-graph compactor [70] carries out the compaction task in two phases. The first phase is called the *abstraction phase.* During this phase, in a bottom-to-top pass through a design's hierarchy, the *port abstraction* (or the more general *DONUT abstraction*) is generated for each subcell. The ports of a subcell are the elements of the subcell which interact with other cells and/or elements in the next (higher) level of the hierarchy; they will be stretched to pitchmatch with the environment. The port abstraction is a simplified definition of a subcell; it is a system of graph constraints among the ports of the subcell (in either the x-dimension or the y-dimension depending on which dimension is currently being compacted). A port abstraction will be passed to the next level of the hierarchy for the generation of the port abstraction for its parent cell. To reduce the execution time and memory usage of the compactor, a port abstraction can be reduced first, then passed to the next level.

In hierarchy-preserving compaction, constraints can be generated by substituting the equations that relate flat variables to subcell variables and pitch variables into the graph constraints on flat variables [44]. Thus, the number of constraints in the resulting LP is equal to the number of graph constraints on flat variables. We can reduce the number of graph constraints on flat variables before the substitution process. This will, of course, reduce the size of the final LP.

Compaction systems which admit user-specified constraints can generate LPs containing non-graph constraints, such as in hierarchy-preserving compaction [44, 54] and symmetry-constraint compaction [61]. For an LP with non-graph constraints, we can treat variables that appear in non-graph constraints as port variables. A new LP can be obtained by combining the constraints in the reduced "port abstraction" and all non-graph constraints. Since usually only a small fraction of the variables appear in non-graph constraints, the new LP can be considerably smaller comparing to the original one. Solving the new LP instead of the original one will lead to a reduction in execution time and memory usage.

7.1 Problem formulations

A *graph constraint* is an inequality of the form $x_i - x_j \geq c$, where x_i and x_j are variables and c is a constant. A *solution* of a system of graph constraints is a vector of values of the variables which satisfies all the graph constraints in the system. If a system has no solutions, it is *inconsistent*. Otherwise, it is *consistent*. The graph constraint reduction problem can be defined as follows: *Given a consistent system of graph constraints, find a system of graph constraints which has the minimum number of graph constraints and the same set of solutions.*

Let S be a system of graph constraints on variables $x_1, \cdots, x_n$. From S, we construct an edge weighted (directed) graph G as follows: Corresponding to each variable x_i, there is a vertex v_i in the vertex set, $V(G)$, of G. Corresponding to each constraint $x_i - x_j \geq c$ in S, there is an edge (v_j, v_i) with weight c in the edge set, $E(G)$, of G. G is called the *constraint graph*[1] of S. We shall use $w_G(e)$ to denote the weight of edge e in G. As an example, Figure 7.1 shows a system of graph constraints and its corresponding constraint graph.

$x_1 - x_4 \geq -1$; $\quad x_2 - x_1 \geq 2$;
$x_2 - x_4 \geq 1$; $\quad x_3 - x_1 \geq 2$;
$x_3 - x_2 \geq 0$; $\quad x_3 - x_4 \geq 1$;
$x_3 - x_5 \geq 0$; $\quad x_4 - x_1 \geq 1$;
$x_4 - x_3 \geq -1$; $\quad x_5 - x_1 \geq 2$;
$x_5 - x_4 \geq 1$.

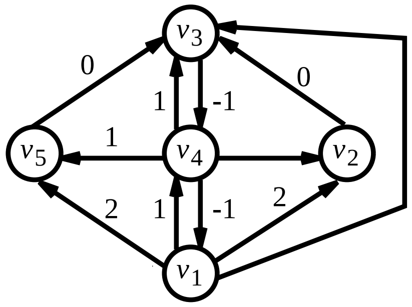


Figure 7.1: A system of graph constraints and its constraint graph.

[1] We assume that S contains at most one constraint of the form $x_i - x_j \geq c$ for any x_i and x_j ($i \neq j$); otherwise, we can retain the one with the largest value on the right-hand side and discard the others. Thus, according to this assumption, G has no multi-edges.

Let $w^*_G(u,v)$ denote the weight of a longest (directed) walk[2] (a walk with the largest weight) from u to v in constraint graph G. As a convention, we assume the degenerated walk from a vertex u to itself without passing any other vertex has weight zero, and $w^*_G(u,v) = -\infty$ if there is no path from u to v in G. The following result shows the importance of the concept of a constraint graph. Its proof can be found in [24, 50].

Theorem 7.1

(i) *If* $(a_1, \cdots, a_n)$ *is a solution of* S*, then,* $a_i - a_j$ *is larger than or equal to* $w^*_G(v_j, v_i)$*;*

(ii) S *is consistent iff* G *contains no (directed) positive closed walk.* □

Because of Theorem 7.1(ii), we shall assume, from now on, that G contains no positive closed walks. Under this assumption, it is easy to see that $w^*_G(u,v)$ is the same as the weight of a longest path from u to v, and $w^*_G(u,u) = 0$ for any u in $V(G)$.

In a constraint graph, an edge is said to be *redundant* if there is a longest walk from u to v that does not contain edge (u,v). From Theorem 7.1(i), it is easy to see that the constraint corresponding to a redundant edge can be removed without changing the set of solutions of a system. That is, a redundant edge can be removed from the constraint graph. However, it is not necessary that all redundant edges of a constraint graph can be removed simultaneously. The reason is that the removal of a redundant edge may cause some of the edges which were redundant originally to become nonredundant.

$G - R$ is called a *subgraph reduction* of G (where $G - R$ denotes the graph obtained from the removal of the edges in R from the edge set of G) if R is a maximum set of redundant edges that can be removed simultaneously. A natural formulation of the graph constraint reduction problem as was proposed by Dutt and Lo [29] is to find a subgraph reduction of a constraint graph (called the *subgraph reduction problem*). A heuristic algorithm for the subgraph reduction problem was also proposed in [29]. The algorithm starts with the given constraint graph, and removes one redundant edge at a time whenever it is possible to do so. The final output obviously depends on the order in which the edges are inspected. In general, the algorithm can

[2] *Walk, closed walk, path, cycle* in directed graphs are the same as those in undirected graphs except that the directions of the edges must be followed. We refer the reader to [7, pp. 11–14] for the definitions of these concepts in undirected graphs. The *weight* of a walk is defined to be the sum of the weights of its edges. A *positive walk* is a walk with positive weight. Negative walks and zero walks are similarly defined.

only remove a *maximal* set of redundant edges instead of a *maximum* set of redundant edges. The following result suggests that finding a subgraph reduction is probably costly.

Theorem 7.2 *The subgraph reduction problem is NP-hard.*

Proof: We reduce the following NP-complete problem [36] to the subgraph reduction problem:

Instance: A strongly-connected graph G.

Question: Is there a (directed) Hamiltonian cycle in G?

The key idea in the proof is the following observation: G has a Hamiltonian cycle iff there is a subgraph reduction of G with no more than $|V(G)|$ edges when each edge in G is assigned weight zero. We omit the details. □

There is another shortcoming with the subgraph reduction formulation. A subgraph reduction does not, in general, represent a maximum possible reduction that can be achieved. This can be demonstrated through the system in Figure 7.1. Since the constraint graph in Figure 7.1 has no directed Hamiltonian cycle, the system corresponding to a subgraph reduction must have at least six constraints. However, the system in Figure 7.2 which consists of only five constraints has the same set of solutions as the one in Figure 7.1 (a consequence of Theorem 7.3 in the following). Notice that the constraint $x_1 - x_5 \geq -2$ in Figure 7.2 is not in the original system. This suggests that adding constraints may be beneficial sometimes.

$$\begin{aligned} x_2 - x_1 &\geq 2; \\ x_3 - x_2 &\geq 0; \\ x_4 - x_3 &\geq -1; \\ x_5 - x_4 &\geq 1; \\ x_1 - x_5 &\geq -2. \end{aligned}$$

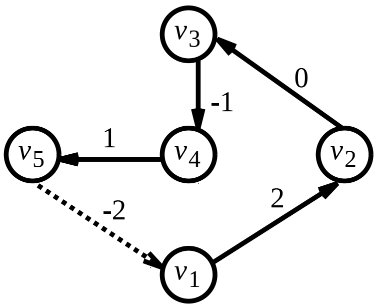


Figure 7.2: A smallest system equivalent to the one in Figure 7.1.

Two constraint graphs G_1 and G_2 on the same vertex set are said to be *equivalent* (denoted by $G_1 \equiv G_2$) if $w^*_{G_1}(u, v) = w^*_{G_2}(u, v)$ for any two vertices u and v. The following result is the basis of the new formulation.

Theorem 7.3 *Let S_1 and S_2 be two systems of graph constraints with G_1 and G_2 being the corresponding constraint graphs. S_1 and S_2 have the same set of solutions iff $G_1 \equiv G_2$.*

Proof: ($\Leftarrow$) Suppose $G_1 \equiv G_2$. Let $\vec{a} = (a_1, a_2, \ldots)$ be a solution of S_1. If $x_i - x_j \geq c$ is a constraint in S_2, that is, (v_j, v_i) is an edge in G_2 with weight c, then

$$a_i - a_j \geq w^*_{G_1}(v_j, v_i) = w^*_{G_2}(v_j, v_i) \geq c.$$

So, $\vec{a}$ is also a solution of S_2. Similarly, we can show a solution of S_2 is also a solution of S_1.

($\Rightarrow$) We observe that if G is the constraint graph of a system S, then for any $v_i, v_j \in V(G)$ there is a solution $\vec{a} = (a_1, a_2, \cdots)$ of S such that $a_j - a_i = w^*_G(v_i, v_j)$. This observation is a direct consequence of Theorem 7.1(ii) and the fact that if edge (v_j, v_i) is added to G with weight $-w^*_G(v_j, v_i)$, the resulting graph contains no positive closed walks.

Suppose that $w^*_{G_1}(u, v) \neq w^*_{G_2}(u, v)$ for some $v_i, v_j \in V$. Without loss of generality we assume $w^*_{G_1}(v_i, v_j) < w^*_{G_2}(v_i, v_j)$. As was stated above, there is a solution $\vec{a}$ of S_1 such that $a_j - a_i = w^*_{G_1}(v_i, v_j)$. It is obvious that $\vec{a}$ cannot be a solution of S_2 since $a_j - a_i = w^*_{G_1}(v_i, v_j) < w^*_{G_2}(v_i, v_j)$. □

G_1 is called an *optimal reduction* of G if $G_1 \equiv G$ and $|E(G_2)| \geq |E(G_1)|$ for any $G_2 \equiv G$. Our new problem formulation is to find an optimal reduction of a given constraint graph G (called the *optimal reduction problem*). Let G_1 be an optimal reduction of G, and G_2 be a subgraph reduction of G. By definition $G_2 \equiv G$, so $|E(G_2)| \geq |E(G_1)|$. Actually, according to Theorem 7.3, the number of graph constraints in the system corresponding to G_1 is minimum among all systems of graph constraints that have the same set of solutions as the one corresponding to G. The major difference between an optimal reduction and a subgraph reduction is that an optimal reduction is not required to be a subgraph of the original graph.

7.2 An algorithm for the optimal reduction problem

In this section we present a polynomial-time algorithm for computing an optimal reduction of a constraint graph. In view of Theorem 7.2, it is somewhat surprising that an efficient algorithm exists for a more general problem formulation. The proofs of the correctness of the algorithm and other results referred to in this section will be presented in Section 7.3.

Let R_G denote the set of redundant edges of constraint graph G. As will be proved in the next section (Theorem 7.4), if G contains no zero closed walks, $G - R_G$ is the unique optimal reduction of G. Furthermore, by Lemma 7.1 if G contains no zero closed walks, R_G can be determined as follows: The edges in G are inspected one by one. For edge (u, v), if there is a vertex $x \neq u, v$ such that $w^*_G(u, x) + w^*_G(x, v) \geq w_G(u, v)$, (u, v) is redundant; otherwise, (u, v) is not. It is obvious that, given w^*_G, this procedure can determine R_G in $O(nm)$ time. (Hereinafter, we let $n = |V(G)|$ and $m = |E(G)|$.)

We now turn our attention to constraint graphs with zero closed walks. For constraint graph G, two vertices u and v are said to be *dependent* if either $u = v$ or there is a zero closed walk in G that contains both u and v. It is not difficult to see that dependency is an equivalence relation. It, therefore, induces a partition, $\pi_G = \{V_1, \ldots, V_k\}$, on the vertex set of G. In other words, the vertex set of G can be divided into disjoint groups $V_1, \ldots, V_k$ such that two vertices are dependent iff they are in the same group. $V_1, \ldots$, and V_k are referred to as the *dependency classes* of G. Note that in the case that G contains no zero closed walks, each dependency class consists of only one vertex.

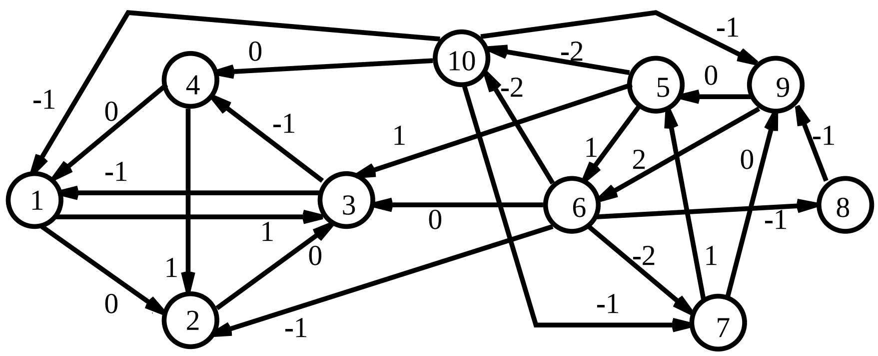
We introduce two weighted graphs G_Z and G_{NZ}. G_Z, where $V(G_Z) = V(G)$, is constructed by forming an arbitrary directed Hamiltonian cycle on the vertices in V_i if $|V_i| \geq 2$, for $1 \leq i \leq k$. G_{NZ} is constructed by arbitrarily choosing one vertex from each V_i ($1 \leq i \leq k$) and creating an edge from u to v where $u \in V_j$ and $v \in V_t$ ($j \neq t$) iff there is an edge in G from some vertex in V_j to some vertex in V_t. The weight assigned to an edge (u, v) in G_Z or G_{NZ} is equal to $w^*_G(u, v)$. The vertex in V_i that is chosen to be in G_{NZ} is called the *representative* of V_i. It can be shown that G_{NZ} contains no zero closed walks (Lemma 7.5). Therefore, G_{NZ} has a unique optimal reduction which will be denoted by G'_{NZ}. Let G_{opt} be the weighted graph obtained by adding all the edges of G'_{NZ} (with their weights) to G_Z. G_{opt} is an optimal reduction of G (Theorem 7.5). The following procedure **ORP** is a summary of the procedure.

ORP(G, w_G)

input: a graph G with weight function w_G.

output: an optimal reduction of G.

1. Compute w_G^* if it is not given;
2. Determine π_G;
3. Construct G_Z and G_{NZ};
4. Compute G'_{NZ};
5. Construct and output G_{opt};



(a) Constraint graph G

$$\begin{bmatrix}
0 & 1 & 1 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-1 & 0 & 0 & -1 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
-1 & 0 & 0 & -1 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
0 & 1 & 1 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
0 & 1 & 1 & 0 & 0 & 1 & -1 & 0 & -1 & -1 \\
-1 & 0 & 0 & -1 & -1 & 0 & -2 & -1 & -2 & -2 \\
1 & 2 & 2 & 1 & 1 & 2 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & -1 & 0 & -1 & -1 \\
1 & 2 & 2 & 1 & 1 & 2 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & -1 & 0 & -1 & 0
\end{bmatrix}$$

(b) w_G^* matrix

Figure 7.3: Constraint graph G and w_G^* matrix.

We use an example to illustrate the graphs constructed in various steps of **ORP**. Figure 7.3 shows a constraint graph G and its w_G^* matrix where the (i, j) entry is the weight of a longest walk from vertex i to vertex j, $1 \le i, j \le 10$. From the w_G^* matrix, it can be easily verified that G has three dependency classes: $\{1, 2, 3, 4\}$, $\{5, 6, 7, 8, 9\}$, and $\{10\}$. Figure 7.4(a) shows G_Z.

By choosing 3, 5, and 10 as the representatives for the dependency classes, we obtain G_{NZ} as shown in Figure 7.4(b). Figure 7.4(c) shows G'_{NZ}, the unique optimal reduction of G_{NZ}. The optimal reduction of G, G_{opt}, which is the union of G_Z and G'_{NZ} is shown in Figure 7.4(d). Note that G has 24 edges, but G_{opt} has only 12 edges.

We now determine the time complexity of **ORP**. First we note that w^*_G can be computed by an all-pair longest path algorithm in time $O(n^2 \log n + nm)$ [24]. Two vertices u and v in G are dependent iff $w^*_G(u,v) + w^*_G(v,u) = 0$ (Lemma 7.3(i)). Combining this result with the fact that dependency is an equivalence relation, π_G can be determined in $O(n^2)$ time when w^*_G is given. It is obvious that $|E(G)| \geq |E(G_{NZ})|$. Therefore, G'_{NZ} can be computed from G_{NZ} in $O(|V(G_{NZ})||E(G_{NZ})|) = O(nm)$ time. As for Step 3 and Step 5 of **ORP**, it is easy to see that they take $O(n^2)$ time when w^*_G is given. Thus, the time complexity of **ORP** is $O(n^2 \log n + nm)$. When w^*_G is given (this is the usual case in hierarchical compaction), the time complexity of **ORP** is $O(n^2 + nm)$.

We have now completely solved the graph constraint reduction problem. Namely, we have a polynomial-time algorithm which, for a given system of graph constraints, produces an equivalent system (of graph constraints) with the minimum number of constraints. Let us take a detailed look at the structure of an optimal reduction. Suppose that x_i and x_j are two variables in S such that v_i and v_j are dependent in G, the constraint graph of S. From Lemma 7.3(i) we have $w^*_G(v_j, v_i) + w^*_G(v_i, v_j) = 0$. We also know that $x_i - x_j \geq w^*_G(v_j, v_i)$ and $x_j - x_i \geq w^*_G(v_i, v_j)$. Therefore, $x_i - x_j = w^*_G(v_j, v_i)$ or $x_i = x_j + w^*_G(v_j, v_i)$. In other words, the relationship between x_i and x_j is actually a linear equality instead of an inequality. Namely, x_i and x_j are dependent on each other in that given the value of one of them, the value of the other is determined. (This is why v_i and v_j are said to be dependent.) As a result, we can remove variable x_i from S (or from any other constraints system that contains S) by simply replacing each appearance of x_i by $x_j + w^*_G(v_j, v_i)$. (Notice that such a replacement transforms a graph constraint into another graph constraint.) We found that a similar idea was also proposed by other researchers [46, 86]. However, there is a major difference between our approach and that in [46, 86]. In both [46] and [86] only explicit equations were considered. Our approach can handle both explicit and implicit (or implied) equations in a uniform way. (Note that an equation $x_i - x_j = c$ can be expressed as two graph constraints $x_i - x_j \geq c$ and $x_j - x_i \geq -c$ before reduction.) Therefore, only our approach can lead to the removal of the maximum number of variables.

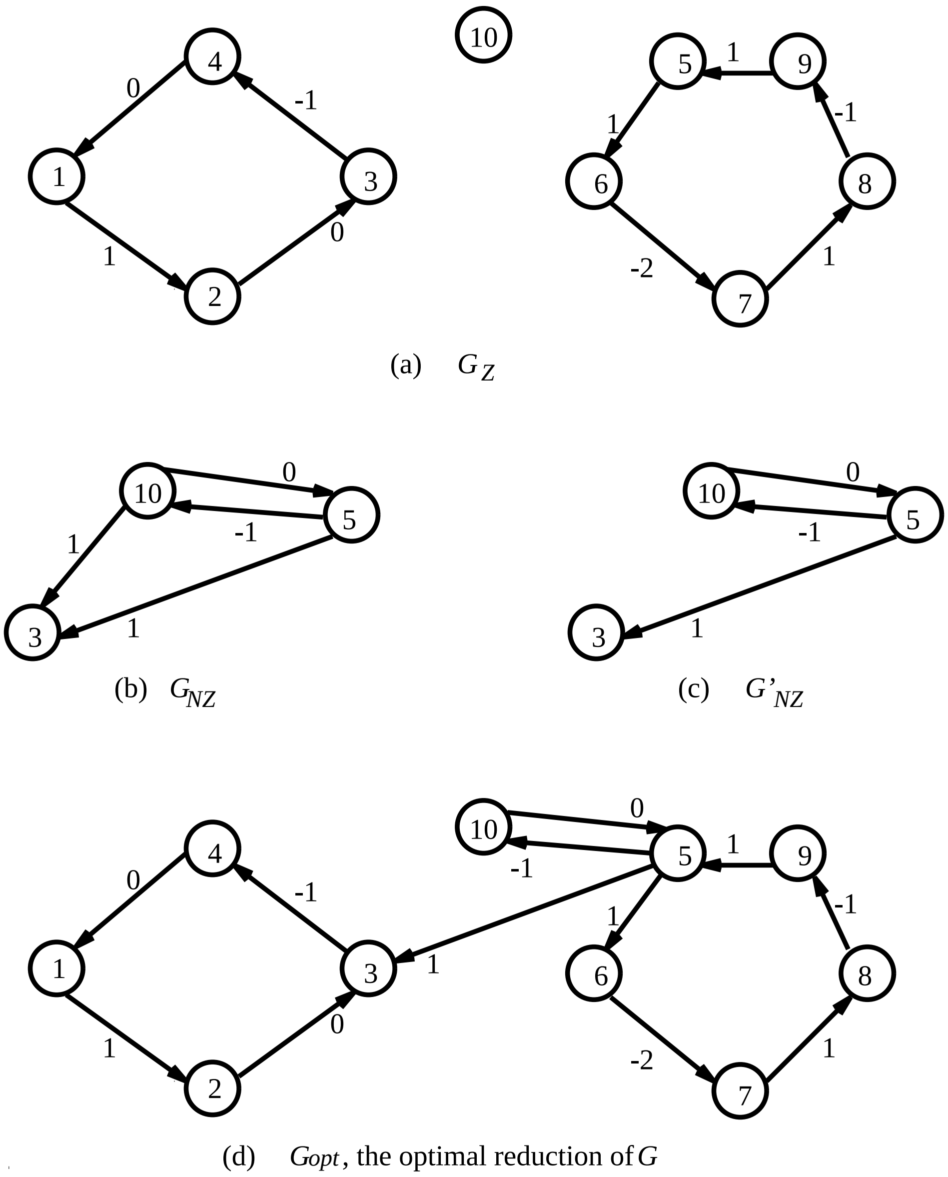


Figure 7.4: An example illustrating the steps of **ORP**.

For the application to hierarchical compaction, instead of passing the system of graph constraints corresponding to the optimal reduction G_{opt} to the next level, we can actually pass the system of graph constraints corresponding to G'_{NZ}. We also retain all the equations that relate the variables not in G'_{NZ} to its representative in G'_{NZ}. (Note that each dependency class has exactly one representative in G'_{NZ}.) These equations will be used to eliminate those variables not represented in G'_{NZ} from the constraints of the parent cell in the abstraction phase and to recover their own values in the realization phase. The advantage of variable reduction is obvious — The execution time and memory usage of a compaction system may be further reduced.

Finally, we would like to point out that the optimal reduction problem can be viewed as a generalization of the classical *transitive reduction problem* studied by Aho, Garey, and Ullman [2]. For an unweighted graph G, if weight zero is assigned to each of its edges, vertex v is reachable from vertex u iff $w^*_G(u,v) = 0$. Otherwise, $w^*_G(u,v) = -\infty$. Therefore, $G_1 \equiv G$ iff G_1 and G have the same reachability relation among their common vertices. (Note that every edge in G_1 must also have zero weight.) Hence, an optimal reduction of the zero-weighted G is a transitive reduction of the unweighted G. In this sense, the optimal reduction problem be viewed as a weighted version of the transitive reduction problem.

7.3 Correctness of the algorithm

In this section we present and prove all the results referred to in Section 7.2. These results validate the correctness as well as the complexity analysis of our algorithm.

Lemma 7.1 *If G contains no zero closed walks, then the set of redundant edges in G,*

$$R_G = \{(u,v) \in E(G) \mid \exists x \neq u, v \text{ such that } w^*_G(u,x) + w^*_G(x,v) \geq w_G(u,v)\}.$$

Proof: That R_G is a subset of the set defined on the right side of the equality is obvious. Now we prove the converse. Let (u,v) be an edge in G such that $w^*_G(u,x) + w^*_G(x,v) \geq w_G(u,v)$ for some $x \neq u, v$. We want to show that (u,v) is redundant, i.e., (u,v) is in R_G. If $w_G(u,v) < w^*_G(u,v)$, (u,v) itself cannot be a longest path from u to v, so (u,v) is redundant. In what follows we assume that $w_G(u,v) = w^*_G(u,v)$. Let P_1 be a longest path from u to x, P_2 be a longest path from x to v in G. We claim that $V(P_1) \cap V(P_2) = \{x\}$. If claim is true, then $P_1 + P_2$ must be a longest path from u to v that does not contain edge (u, v). Namely, (u, v) is redundant. Now we show that the claim is indeed true. Suppose otherwise, let y be the vertex nearest to u in $V(P_1) \cap V(P_2)$ along P_1 (note $y \neq x$). y partitions P_1 into two parts: $P_{1,1}$, the part from u to y, and $P_{1,2}$, the part from y to x. y also partitions P_2 into two parts: $P_{2,1}$, the part from x to y, and $P_{2,2}$, the part from y to v. Clearly, $P_{1,1} + P_{2,2}$ is a path from u to v in G, and $P_{1,2} + P_{2,1}$ is a closed walk in G with at least one edge. Of course, $w(P_{1,2} + P_{2,1}) < 0$. Therefore,

$$
\begin{aligned}
w(P_{1,1} + P_{2,2}) &> w(P_{1,1} + P_{2,2}) + w(P_{1,2} + P_{2,1}) \\
&= w(P_1) + w(P_2) \\
&= w_G^*(u, x) + w_G^*(x, v) = w_G(u, v) = w_G^*(u, v)
\end{aligned}
$$

This is a contradiction. □

Lemma 7.2 *Let $G_1 \equiv G_2$ and neither G_1 nor G_2 contains zero closed walks. Let e be an edge in $E(G_1)$. If either (i) $e \notin E(G_2)$, or (ii) $e \in E(G_2)$ and $w_{G_1}(e) \neq w_{G_2}(e)$, then $G_1 - \{e\} \equiv G_2$.*

Proof: Let $e = (u, v)$. We prove the lemma by examining three possible cases:

Case 1: $e \notin E(G_2)$.

In this case, a longest walk from u to v in G_2 must pass some vertex $x \neq u, v$. In other words, $w_{G_2}^*(u, v) = w_{G_2}^*(u, x) + w_{G_2}^*(x, v)$. Since $G_1 \equiv G_2$, we have

$$w_{G_1}(u, v) \leq w_{G_1}^*(u, v) = w_{G_1}^*(u, x) + w_{G_1}^*(x, v)$$

By Lemma 7.1, (u, v) is redundant in G_1, thus $G_1 - \{e\} \equiv G_1 \equiv G_2$.

Case 2: $e \in E(G_2)$ and $w_{G_1}(e) < w_{G_2}(e)$.

In this case, $w_{G_1}^*(u, v) = w_{G_2}^*(u, v) \geq w_{G_2}(e) > w_{G_1}(e)$. Obviously, (u, v) is redundant in G_1, so, $G_1 - \{e\} \equiv G_1 \equiv G_2$.

Case 3: $e \in E(G_2)$ and $w_{G_1}(e) > w_{G_2}(e)$.

Applying Case 2, we know that $G_2' = G_2 - \{e\} \equiv G_2$. Then applying Case 1 for G_1 and G_2', we have $G_1 - \{e\} \equiv G_2' \equiv G_2$. □

Suppose G contains no zero closed walks. Since $G - \{e\} \equiv G$ for any $e \in R_G$, we have $G - R_G \equiv G$ by Lemma 7.2. We also know that $G - R_G$ is minimal in the sense that removing any edge or changing the weight of any edge of $G - R_G$ will result in a graph not equivalent to G. Once again by Lemma 7.2, any graph that is equivalent to G must contain $G - R_G$ together with its edge weights. Thus, $G - R_G$ is the unique optimal reduction of G. This result is stated as the following theorem.

Theorem 7.4 *If G is a constraint graph without zero closed walks, then $G - R_G$ is the unique optimal reduction of G.* □

The following lemma lists two important properties of the dependency relation.

Lemma 7.3

(i) *Two vertices u and v in G are dependent iff $w_G^*(u,v) + w_G^*(v,u) = 0$.*

(ii) *If u_1 and u_2 are in the same dependency class and v_1 and v_2 are in the same dependency class, then $w_G^*(u_1,v_1) = w_G^*(u_1,u_2) + w_G^*(u_2,v_2) + w_G^*(v_2,v_1)$.*

Proof: (i) ($\Leftarrow$) If $w_G^*(u,v) + w_G^*(v,u) = 0$, then a longest path from u to v together with a longest path from v to u form a zero closed walk that contains both u and v. Hence, u and v are dependent.

($\Rightarrow$) Let $P_1 + P_2$ be a zero closed walk that contains u and v, where P_1 is a walk from u to v and P_2 is a walk from v to u. Let P be a longest walk from u to v. Suppose $w(P) = w_G^*(u,v) > w(P_1)$, then $P + P_2$ is a positive closed walk — impossible! Thus $w(P_1) = w_G^*(u,v)$. Similarly, we can show $w(P_2) = w_G^*(v,u)$. So,

$$w_G^*(u,v) + w_G^*(v,u) = w(P_1) + w(P_2) = 0.$$

By the construction of G_N, (i) is obvious.

(ii) By the "triangle inequality", we have

$$w_G^*(u_1,v_1) \geq w_G^*(u_1,u_2) + w_G^*(u_2,v_2) + w_G^*(v_2,v_1) \tag{7.1}$$

$$w_G^*(u_2,v_2) \geq w_G^*(u_2,u_1) + w_G^*(u_1,v_1) + w_G^*(v_1,v_2) \tag{7.2}$$

On the other hand, by (i), we have

$$w_G^*(u_1,u_2) + w_G^*(u_2,u_1) = 0 \quad \text{and} \quad w_G^*(v_1,v_2) + w_G^*(v_2,v_1) = 0 \tag{7.3}$$

Combining (1), (2) and (3), the conclusion follows. □

If u_1, u_2, and u_3 are in the same dependency class, by (ii),

$$w^*_G(u_1, u_3) = w^*_G(u_1, u_2) + w^*_G(u_2, u_3) + w^*_G(u_3, u_3) = w^*_G(u_1, u_2) + w^*_G(u_2, u_3).$$

Thus, we have the following corollary of (ii).

Corollary 7.1 *If u_1, u_2, and u_3 are in the same dependency class, then*
$w^*_G(u_1, u_3) = w^*_G(u_1, u_2) + w^*_G(u_2, u_3)$. □

We have the following result concerning G_Z and G_{NZ}.

Lemma 7.4

(i) $w^*_{G_Z}(u, v) = w^*_G(u, v)$ *if u and v are in the same dependency class.*

(ii) $w^*_{G_{NZ}}(u, v) = w^*_G(u, v)$ *for $u, v \in V(G_{NZ})$.*

Proof: (i) By repeatedly applying Corollary 7.1, we can easily show the following more general result: If $v_1, \ldots, v_t$ $(t \geq 3)$ are in the same dependency Class, then

$$w^*_G(v_1, v_t) = w^*_G(v_1, v_2) + \cdots + w^*_G(v_i, v_{i+1}) + \cdots + w^*_G(v_{t-1}, v_t).$$

(ii) $w^*_{G_{NZ}}(u, v) \leq w^*_G(u, v)$ because any edge in G_{NZ} corresponds to a walk in G with the same weight and starting and ending vertices. What remains to be shown is that $w^*_{G_{NZ}}(u, v) \geq w^*_G(u, v)$. Let P be a longest walk from u to v in G. P can be partitioned in the following way:

$$P = P_1 e_1 P_2 e_2 \cdots e_t P_{t+1} \qquad (t \geq 1)$$

where P_i (for $1 \leq i \leq t+1$) is a walk from u_{l_i} to u_{r_i} such that u_{l_i} and u_{r_i} are in the same dependency class, and $e_i = (u_{r_i}, u_{l_{i+1}})$ (for $1 \leq i \leq t$) is an edge such that u_{r_i} and $u_{l_{i+1}}$ are in different dependency classes. (Note that $u = u_{l_1}$ and $v = u_{r_{t+1}}$.)

Let u_{c_i} denote the representative of the dependency class that contains u_{l_i} and u_{r_i}. (Note that $u_{c_1} = u = u_{l_1}$ and $u_{c_{t+1}} = v = u_{r_{t+1}}$.) By Corollary 7.1, for $2 \leq i \leq t$,

$$w_G(P_i) \leq w^*_G(u_{l_i}, u_{r_i}) = w^*_G(u_{l_i}, u_{c_i}) + w^*_G(u_{c_i}, u_{r_i}).$$

By the construction of G_{NZ}, we know that $(u_{c_i}, u_{c_{i+1}})$ is an edge in G_{NZ} with weight $w^*_G(u_{c_i}, u_{c_{i+1}})$ for $1 \le i \le t$. Therefore,

$$\begin{aligned}
w^*_G(u,v) &= w_G(P) \\
&= w_G(P_1) + \sum_{2\le i\le t} w_G(P_i) + \sum_{2\le i\le t} w_G(e_i) + w_G(P_{t+1}) \\
&\le w^*_G(u_{l_1}, u_{r_1}) + \sum_{2\le i\le t} [w^*_G(u_{l_i}, u_{c_i}) + w^*_G(u_{c_i}, u_{r_i})] \\
&\quad + \sum_{2\le i\le t} w^*_G(u_{r_i}, u_{l_{i+1}}) + w^*_G(u_{l_{t+1}}, u_{r_{t+1}}) \\
&= \sum_{1\le i\le t} [w^*_G(u_{c_i}, u_{r_i}) + w^*_G(u_{r_i}, u_{l_{i+1}}) + w^*_G(u_{l_{i+1}}, u_{c_{i+1}})] \\
&\le \sum_{1\le i\le t} w^*_G(u_{c_i}, u_{c_{i+1}}) \\
&= \sum_{1\le i\le t} w_{G_{NZ}}(u_{c_i}, u_{c_{i+1}}) \\
&\le w^*_{G_{NZ}}(u_{c_i}, u_{c_{i+1}}) = w^*_{G_{NZ}}(u,v).
\end{aligned}$$

□

Suppose G_{NZ} contains a zero closed walk and both u and v are on the closed walk. By Lemma 7.3(i) $w^*_{G_{NZ}}(u,v) + w^*_{G_{NZ}}(v,u) = 0$. This implies that $w^*_G(u,v) + w^*_G(v,u) = 0$ by Lemma 7.4(ii). Namely, u and v are dependent in G, which obviously contradicts the fact that each dependency class has one and only one representative in G_{NZ}. Thus, we have the following result:

Lemma 7.5 *G_{NZ} contains no zero closed walks.* □

Finally, we show the correctness of algorithm **ORP**.

Theorem 7.5 *G_{opt} is an optimal reduction of G.*

Proof: First we show that $G_{opt} \equiv G$, i.e., for any u and v, $w^*_{G_{opt}}(u,v) = w^*_G(u,v)$. Since any edge in G_{opt} corresponds to a walk in G with the same weight and starting and ending vertices, for any walk from u to v in G_{opt}, there is a walk from u to v in G with the same weight. Consequently, $w^*_{G_{opt}}(u,v) \le w^*_G(u,v)$. Therefore, it suffices to show that $w^*_{G_{opt}}(u,v) \ge w^*_G(u,v)$. There are two cases:

Case 1: u and v are in the same dependency class.

$$\begin{aligned} w^*_G(u,v) &= w^*_{G_Z}(u,v) \qquad \text{(Lemma 7.4(i))} \\ &\leq w^*_{G_{opt}}(u,v) \end{aligned}$$

Case 2: u and v are in different dependency classes.

Let u_1 (resp. v_1) be the representative of the dependency class that contains u (resp. v). Then,

$$\begin{aligned} w^*_G(u,v) &= w^*_G(u,u_1) + w^*_G(u_1,v_1) + w^*_G(v_1,v) && \text{(Lemma 7.3(ii))} \\ &= w^*_{G_Z}(u,u_1) + w^*_{G_{NZ}}(u_1,v_1) + w^*_{G_Z}(v_1,v) && \text{(Lemma 7.4(i)(ii))} \\ &\leq w^*_{G_{opt}}(u,u_1) + w^*_{G_{opt}}(u_1,v_1) + w^*_{G_{opt}}(v_1,v) \;\leq\; && w^*_{G_{opt}}(u,v). \end{aligned}$$

We now show that G_{opt} has the minimum number of edges. According to the construction procedure G_Z depends only on w^*_G and is independent of the structure of G. The construction of G_{NZ} does depend on the structure of G. However, $w^*_{G_{NZ}}$ is a submatrix of w^*_G; we also know that G_{NZ} contains no zero closed walks. By Theroem 7.1, G'_{NZ}, the optimal reduction of G_{NZ}, is unique. Therefore, G_{opt} only depends on w^*_G. In other words, for any G' such that $G' \equiv G$, if we run **ORP** on G' and use the same Hamiltonian cycle and choose the same representative for each dependency class as when running **ORP** on G, we will end up with the same output graph: G_{opt}. Therefore, it suffices to show $|E(G)| \geq |E(G_{opt})|$. To see why $|E(G)| \geq |E(G_{opt})|$, we partition G into two graphs G_1 and G_2. G_1 contains the edges in G with their two end-vertices in the same dependency class, and G_2 contains the edges in G with their two end-vertices in different dependency classes. Note that each dependency class induces a strongly-connected component in G_1 because all vertices in a zero closed walk obviously belong to the same dependency class. Therefore, $|E(G_1)| \geq \sum_{|V_i|\geq 2} |V_i|$. By the construction of G_{NZ}, we also have $|E(G_2)| \geq |E(G_{NZ})| \geq |E(G'_{NZ})|$. Thus,

$$|E(G)| = |E(G_1)| + |E(G_2)| \geq \sum_{|V_i|\geq 2} |V_i| + |E(G'_{NZ})| = |E(G_{opt})|$$

This concludes our proof. □

7.4 Experimental results

Our algorithm **ORP** was implemented. For comparison, we also implemented the heuristic algorithm in [29] which will be denoted by **SRP**. We present in the following our experimental results.

First, we compared **ORP** with **SRP** using test examples which are randomly chosen subgraphs of the MCNC compaction benchmarks nand3 and afa. nand3 has 57 vertices and 236 constraints; afa has 210 vertices and 987 constraints. The results are listed in Table 7.1. The results in the **SRP** column were obtained by running **SRP** ten times according to different edge-orderings for redundancy checking and choosing the best result. Under the column *diff*, we list the difference between the number of constraints produced by **SRP** and that produced by **ORP**. From Table 7.1, we observed that **SRP** usually does not yield the maximum possible reduction which is, however, guaranteed by **ORP**.

example	# of vertices	# of constraints			diff
		initial	**SRP**	**ORP**	
nand3.1	5	6	5	5	0
nand3.2	10	5	5	4	1
nand3.3	20	27	25	21	4
nand3.4	30	62	45	41	4
nand3.5	57	236	105	89	16
afa.1	40	61	50	45	5
afa.2	50	110	77	66	11
afa.3	60	97	80	76	4
afa.4	70	177	113	98	15
afa.5	210	987	417	400	17

Table 7.1: Comparison between **ORP** and **SRP**.

Next, we carried out an experiment to test the effectiveness of the concept of graph constraint reduction in hierarchical pitchmatching compaction. We also used compaction benchmarks nand3 and afa. In each test example, we randomly selected a set of variables to be the (fictitious) ports of the layout. We then computed the initial port abstraction and the reduced port abstractions generated by **ORP** and **SRP**. As in the first experiment, the results under column **SRP** were generated by running **SRP** ten times according to different edge-orderings and cho85 the best result. The results are summarized in Table 7.2. Under the column *re-*

duction percentage, we list the percentage of reduction produced by **ORP**. From this table it is clear that the amount of reduction is substantial. The results in Table 7.2 also show that for port abstractions, **SRP** has a better chance to produce good results. The reason is that the constraint graphs for the initial port abstractions are transitive. For a transitive graph, a subgraph reduction happens to be an optimal reduction. Therefore, there exists at least one edge-ordering that will lead **SRP** to an optimal reduction. This is in contrast to the general case in which there may not exist an edge-ordering that can lead **SRP** to an optimal reduction.

example	# of vertices	# of constraints			reduction percentage
		initial	**SRP**	**ORP**	
nand3.1'	5	7	5	5	28.6
nand3.2'	10	56	14	14	75.0
nand3.3'	18	152	40*	37	75.7
nand3.4'	20	189	34	34	82.0
nand3.5'	30	494	44	41	91.7
afa.1'	27	302	67*	65	78.5
afa.2'	40	699	80	79	88.7
afa.3'	50	1148	94	91	92.1
afa.4'	60	1560	120	118	92.4
afa.5'	70	2237	130	127	94.3

*These examples use the actual ports of the cells and the results of **SRP** were generated by Dutt and Lo.

Table 7.2: The effectiveness of constraint reduction.

Finally, we applied **ORP** to hierarchy-preserving compaction. For the MCNC compaction benchmark mul2, the compactor in [44] generates 4008 graph constraints on flat variables. Using algorithm **ORP**, we produced an equivalent system with 2623 graph constraints. This is about a 35% reduction. Consequently, the number of general constraints in the final LP for the hierarchy-preserving compaction of mul2 is reduced by about 35%.

In summary, our experimental results further testify that our algorithm is quite effective and the amount of reduction is substantial.

7.5 Summary

Constraint reduction is an effective way to reduce the execution time and memory usage of compaction systems. Since most of the constraints are derived from separation and connectivity requirements which can be represented as graph constraints, we study the graph constraint reduction problem in this chapter. We propose a new formulation of the problem under which a maximum possible reduction is guaranteed. Furthermore, we present a polynomial-time algorithm for determining a maximum possible reduction. The results presented here can be used to improve the efficiency of various compaction systems.

Chapter 8

Conclusions and Future Work

We have studied several problems arising from the design of VLSI circuits in this thesis. Specifically, we studied the following problems: partial scan with retiming, retiming and resynthesis for cycle time reduction, technology mapping for LUT-based FPGAs, area minimization in floorplanning, and graph constraint reduction in compaction. We proposed algorithms for solving these problems. These algorithms have been shown to be efficient and effective both theoretically and experimentally.

As is always the case in any research, many problems remain to be answered. In addition to the open problems mentioned in each chapter, one main issue to be investigated in the future is how to integrate retiming into the solution of various synthesis and optimization problems in sequential circuits. Currently many optimization problems for sequential circuits are still posed under the assumption that the FF positions are fixed and in many cases the optimization is carried out only on the combinational logic between the FFs. The assumption that the FFs have fixed positions greatly restricts the solution space that can be explored for an optimization problem on sequential circuits. Therefore, to find the best solution to an optimization problem, retiming need to be integrated into the solution process of the problem. Our results in Chapters 2, 3 and 4 are some encouraging initial steps toward this goal.

Bibliography

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, New York, 1990.

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1:131–137, 1972.

[3] P. Ashar and S. Malik. Implicit computation of minimum-cost feedback-vertex sets for partial scan and other applications. In *Proc. ACM/IEEE Design Automation Conf.*, pages 77–80, 1994.

[4] C. S. Bamji and R. Varadarajan. Hierarchical pitchmatching compaction using minimum design. In *Proc. ACM/IEEE Design Automation Conf.*, pages 311–317, 1992.

[5] K. Bartlett, G. Borriello, and S. Raju. Timing optimization of multiphase sequential logic. *IEEE Trans. on Computer-Aided Design*, 10:51–62, 1991.

[6] N. Bhat and D. Hill. Routable technology mapping for FPGAs. In *ACM/SIGDA Workshop on FPGAs*, pages 143–148, 1992.

[7] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North-Holland, New York, 1976.

[8] D. Boyer. Symbolic layout compaction review. In *Proc. ACM/IEEE Design Automation Conf.*, pages 383–389, 1988.

[9] R. L. Brooks, C. A. B. Smith, A. H. Stone, and W.T. Tutte. The dissection of rectangles into squares. *Duke Math. J.*, 7:312–340, 1940.

[10] J. Burns and R. Newton. Sparcs: A new constraint-based IC symbolic layout spacer. In *Proc. IEEE Custom Integrated Circuits Conf.*, pages 534–539, 1986.

[11] S. T. Chakradhar, A. Balakrishnan, and V. D. Agrawal. An exact algorithm for determining partial scan flip-flops. In *Proc. ACM/IEEE Design Automation Conf.*, pages 81–86, 1994.

[12] S. T. Chakradhar and S. Dey. Resynthesis and retiming for optimum partial scan. In *Proc. ACM/IEEE Design Automation Conf.*, pages 87–93, 1994.

[13] S. T. Chakradhar, S. Dey, M. Potkonjak, and S. G. Rothweiler. Sequential circuit delay optimization using global path delays. In *Proc. ACM/IEEE Design Automation Conf.*, pages 483–489, 1993.

[14] C.-H. Chen and I. G. Tollis. Area optimization of spiral floorplans. *Journal of Circuits, Systems, and Computers*, 3:833–857, 1993.

[15] K.C. Chen, J. Cong, Y. Ding, A.B. Kahng, and P. Trajmar. DAG-Map: Graph-based FPGA technology mapping for delay optimization. *IEEE Design and Test of Computer*, 10:7–20, 1992.

[16] K. T. Cheng and V. D. Agrawal. A partial scan method for sequential circuits with feedback. *IEEE Trans. on Computers*, 39:544–548, 1990.

[17] V. Chickermane and J. H. Patel. A fault oriented partial scan design approach. In *Digest Intl. Conf. on Computer-Aided Design*, pages 400–403, 1991.

[18] Y. E. Cho. A subjective review of compaction. In *Proc. ACM/IEEE Design Automation Conf.*, pages 396–404, 1985.

[19] K. Chong and S. Sahni. Optimal realizations of floorplans. *IEEE Trans. on Computer-Aided Design*, 12:793–801, 1993.

[20] J. Cong and Y. Ding. Beyond the combinational limit in depth minimization for LUT-based FPGA designs. In *Digest Intl. Conf. on Computer-Aided Design*, pages 110–114, 1993.

[21] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. on Computer-Aided Design*, 13:1–11, 1994.

[22] J. Cong and Y. Ding. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Trans. on VLSI Systems*, 2:137–148, 1994.

[23] J. Cong and Y. Ding. On nominal delay minimization in LUT-based FPGA technology mapping. *Integration, the VLSI Journal*, 18:73–94, 1994.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, New York, 1990.

[25] W.-M. Dai and E. S. Kuh. Simultaneous floor planning and global routing for hierarchical building block layout. *IEEE Trans. on Computer-Aided Design*, 6:828–837, 1987.

[26] S. Dey, F. Brglez, and G. Kedem. Partitioning sequential circuits for logic optimization. In *Proc. Intl. Conf. on Computer Design*, pages 70–76, 1991.

[27] S. Dey, M. Potkonjak, and S. G. Rothweiler. Performance optimization of sequential circuits by eliminating retiming bottlenecks. In *Digest Intl. Conf. on Computer-Aided Design*, pages 504–509, 1992.

[28] J. Doenhardt and T. Lengauer. Algorithmic aspects of one dimensional layout compaction. *IEEE Trans. on Computer-Aided Design*, 6:863–878, 1987.

[29] D. Dutt and C. Y. Lo. On minimal closure constraint generation for symbolic cell assembly. In *Proc. ACM/IEEE Design Automation Conf.*, 1991.

[30] H. Edelsbrunner and E. P. Mucke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics*, 9:66–104, 1990.

[31] A. H. Farrahi and M. Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Trans. on Computer-Aided Design*, 13:1319–1332, 1994.

[32] R. J. Francis, J. Rose, and K. Chung. Chortle: A technology mapping for lookup table-based field programmable gate arrays. In *Proc. ACM/IEEE Design Automation Conf.*, pages 613–619, 1990.

[33] R. J. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *Proc. ACM/IEEE Design Automation Conf.*, pages 227–233, 1991.

[34] R. J. Francis, J. Rose, and Z. Vranesic. Technology mapping for lookup table-based FPGAs for performance. In *Digest Intl. Conf. on Computer-Aided Design*, pages 568–571, 1991.

[35] M. L. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

[36] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of* **NP**-*completeness*. Freeman, San Francisco, 1979.

[37] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. Assoc. Comput. Mach.*, 22:463–468, 1975.

[38] D. Kagaris and S. Tragoudas. Partial scan with retiming. In *Proc. ACM/IEEE Design Automation Conf.*, pages 249–254, 1993.

[39] K. Karplus. Xmap: A technology mapper for table-lookup FPGAs. In *Proc. ACM/IEEE Design Automation Conf.*, pages 240–243, 1991.

[40] K. S. Kim and C. R. Kime. Partial scan by use of empirical testability. In *Digest Intl. Conf. on Computer-Aided Design*, pages 314–317, 1990.

[41] C. Kingsley. A hierarchical error-tolerant compactor. In *Proc. ACM/IEEE Design Automation Conf.*, pages 126–132, 1984.

[42] E.L. Lawler, K.N. Levitt, and J. Turner. Module clustering to minimize delay in digital networks. *IEEE Trans. on Computers*, 18:47–57, 1969.

[43] D. H. Lee and S. M. Reddy. On determining scan flip-flops in partial-scan designs. In *Digest Intl. Conf. on Computer-Aided Design*, pages 322–325, 1990.

[44] J. F. Lee and D. T. Tang. Himalayas-a hierarchical compaction system with a minimized constraint set. In *Digest Intl. Conf. on Computer-Aided Design*, pages 150–157, 1992.

[45] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. 3rd Caltech Conf. on VLSI*, pages 87–116, 1983.

[46] T. Lengauer. On the solution of inequality systems relevant to ic-layout. *J. of Algorithms*, 2:408–421, 1984.

[47] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, New York, 1990.

[48] T. Lengauer and R. Muller. The complexity of floorplanning based on binary circuit partitions. Technical Report 46, Department of Mathematics and Computer Science, University of Paderborn, Paderborn, Germany, 1988.

[49] T. Lengauer and R. Muller. Robust and accurate hierarchical floorplanning with integrated global wiring. *IEEE Trans. on Computer-Aided Design*, 12:802–809, 1993.

[50] Y. Liao and C. K. Wong. An algorithm to compact VLSI symbolic layouts with mixed constraints. *IEEE Trans. on Computer-Aided Design*, 2:62–69, 1983.

[51] C. L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill Book Company, New York, 1968.

[52] S. Malik, E. M. Sentovich, and R. Brayton. Retiming and resynthesis: optimizing sequential networks with combinational techniques. *IEEE Trans. on Computer-Aided Design*, 10:74–84, 1991.

[53] S. Malik, K. J. Singh, R. Brayton, and A. L. Sangiovanni-Vincentelli. Performance optimization of pipelined logic circuits using peripheral retiming and resynthesis. *IEEE Trans. on Computer-Aided Design*, 12:568–578, 1993.

[54] D. Marple. A hierarchy preserving hierarchical compactor. In *Proc. ACM/IEEE Design Automation Conf.*, pages 375–381, 1990.

[55] A. Mathur and C. L. Liu. Performance driven technology mapping for lookup-table based FPGAs using the general delay model. In *ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1994.

[56] G. De Micheli. Synchronous logic synthesis: algorithms for cycle-time minimization. *IEEE Trans. on Computer-Aided Design*, 10:63–73, 1991.

[57] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential synthesis for table look up programmable gate arrays. In *Proc. ACM/IEEE Design Automation Conf.*, pages 224–229, 1993.

[58] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis algorithms for table look up programmable gate arrays. In *Proc. ACM/IEEE Design Automation Conf.*, pages 620–625, 1990.

[59] R. Murgai, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Improved logic synthesis algorithms for table look up architectures. In *Digest Intl. Conf. on Computer-Aided Design*, pages 564–567, 1991.

[60] R. Murgai, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Performance directed synthesis for table look up programmable gate arrays. In *Digest Intl. Conf. on Computer-Aided Design*, pages 572–575, 1991.

[61] R. Okuda, T. Sato, H. Onodera, and K. Tamaru. An efficient algorithm for layout compaction problem with symmetry constraints. In *Digest Intl. Conf. on Computer-Aided Design*, pages 148–153, 1989.

[62] R. H. J. M. Otten. Automatic floorplan design. In *Proc. ACM/IEEE Design Automation Conf.*, pages 261–267, 1982.

[63] R. H. J. M. Otten. Graphs in floorplan design. *Intl Journal of Circuit Theory and Appl.*, 16:391–410, 1988.

[64] P. Pan, S.K. Dong, and C. L. Liu. Optimal graph constraint reduction for symbolic layout compaction. In *Proc. ACM/IEEE Design Automation Conf.*, pages 401–406, 1993.

[65] P. Pan and C. L. Liu. Area minimization for general floorplans. In *Digest Intl. Conf. on Computer-Aided Design*, pages 606–609, 1992.

[66] P. Pan and C. L. Liu. Partial scan with pre-selected scan signals. In *Proc. ACM/IEEE Design Automation Conf.*, pages 189–194, 1995.

[67] P. Pan and C.L. Liu. Area minimization for floorplans. *IEEE Trans. on Computer-Aided Design*, 14:123–132, 1995.

[68] P. Pan, W. Shi, and C. L. Liu. Area minimization for hierarchical floorplans. In *Digest Intl. Conf. on Computer-Aided Design*, pages 436–440, 1994.

[69] P. S. Parikh and M. Abramovici. A cost-based approach to partial scan. In *Proc. ACM/IEEE Design Automation Conf.*, pages 255–259, 1993.

[70] M. Reichelt and W. Wolf. An improved cell model for hierarchical constraint-graph compaction. In *Digest Intl. Conf. on Computer-Aided Design*, pages 482–485, 1986.

[71] P. Sawkar and D. Thomas. Area and delay mapping for table-look-up based field programmable gate arrays. In *Proc. ACM/IEEE Design Automation Conf.*, pages 368–373, 1992.

[72] P. Sawkar and D. Thomas. Performance directed technology mapping for look-up table based FPGAs. In *Proc. ACM/IEEE Design Automation Conf.*, pages 208–212, 1993.

[73] M. Schlag, J. Kong, and P.K. Chan. Routability-driven technology mapping for lookup table-based FPGA's. *IEEE Trans. on Computer-Aided Design*, 13:13–26, 1994.

[74] E. M. Sentovich et al. Sequential circuit design using synthesis and optimization. In *Proc. Intl. Conf. on Computer Design*, pages 328–333, 1992.

[75] N. V. Shenoy, R. Brayton, and A. L. Sangiovanni-Vincentelli. Resynthesis of multi-phase pipelines. In *Proc. ACM/IEEE Design Automation Conf.*, pages 490–496, 1993.

[76] H. Shin, A. L. Sangiovanni-Vincentelli, and C. H. Sequin. Two-dimensional module compactor based on zone-refining. In *Proc. Intl. Conf. on Computer Design*, pages 201–203, 1987.

[77] K. J. Singh, A. R. Wang, R. Brayton, and A. L. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Digest Intl. Conf. on Computer-Aided Design*, pages 282–285, 1988.

[78] L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Info. and Control*, 59:91–101, 1983.

[79] K. J. Supowit and E. A. Slutz. Placement algorithms for custom VLSI. *Computer Aided Design*, 16:45–50, 1984.

[80] H. J. Touati, H. Savoj, and R. K. Brayton. Delay optimization of combinational logic circuits by clustering and partial collapsing. In *Digest Intl. Conf. on Computer-Aided Design*, pages 188–191, 1991.

[81] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11:298–313, 1982.

[82] T.-C. Wang and D. F. Wong. A note on the complexity of stockmeyer's floorplan optimization technique. In M. Sarrafzadeh and D. T. Lee, editors, *Algorithmic Aspects of VLSI Layout*, pages 309–320. World Scientific Publishing, 1992.

[83] T.-C. Wang and D. F. Wong. Optimal floorplan area optimization. *IEEE Trans. on Computer-Aided Design*, 11:992–1002, 1992.

[84] U. Weinmann and W. Rosenstiel. Technology mapping for sequential circuits based on retiming techniques. In *Proc. European Design Automation Conf.*, pages 318–323, 1993.

[85] S. Wimer, I. Koren, and I. Cederbaum. Optimal aspect ratios of building blocks in VLSI. *IEEE Trans. on Computer-Aided Design*, 8:139–145, 1989.

[86] W. H. Wolf and A. E. Dunlop. Symbolic layout and compaction. In B. Preas and M. Lorenzetti, editors, *Physical Design Automation of VLSI Systems*. The Benjamin/Cummings Publ. Co., Inc., Menlo Park, CA, 1988.

[87] D. F. Wong and P. S. Sakhamuri. Efficient floorplan area optimization. In *Proc. ACM/IEEE Design Automation Conf.*, pages 586–589, 1989.

[88] N.-S. Woo. A heuristic method for FPGA technology mapping based on the edge visibility. In *Proc. ACM/IEEE Design Automation Conf.*, pages 248–251, 1991.

[89] Xilinx. *The Programmable Gate Arrays Data Book.* Xilinx, San Jose, CA, 1993.

[90] H. Yang and D. F. Wong. Edge-Map: Optimal performance driven technology mapping for iterative LUT based FPGA designs. In *Digest Intl. Conf. on Computer-Aided Design*, pages 150–155, 1994.

[91] S. Yang. Logic synthesis and optimization benchmarks, user guide version 3.0. Technical report, MCNC, 1991.

[92] S.-Z. Yao, C.-K. Cheng, D. Dutt, S. Nahar, and C.-Y. Lo. Cell-based hierarchical pitch-matching compaction using minimal LP. In *Proc. ACM/IEEE Design Automation Conf.*, pages 395–400, 1993.

[93] K. H. Yeap and M. Sarrafzadeh. An integrated algorithm for optimal floorplan sizing and enumeration. In *Proc. European Design Automation Conf.*, pages 29–33, 1993.

[94] G. Zimmermann. A new area and shape function estimation technique for VLSI layouts. In *Proc. ACM/IEEE Design Automation Conf.*, pages 60–65, 1988.