Optimal Generation of Design-Specific Cell Libraries

(設計固有セルライブラリの最適生成手法)

A dissertation submitted in partial satisfaction of
the requirements for the degree of
Doctor of Philosophy in Electronic Engineering

in the

DEPARTMENT OF ELECTRONIC ENGINEERING
of the
UNIVERSITY OF TOKYO

by

Hiroaki Yoshida

Supervisor: Professor Kunihiro Asada

December 2006

Abstract

This dissertation focuses on optimal generation of design-specific cell libraries. In cell-based integrated circuit design, a cell library defines the final quality of a design. Hence, use of a general-purpose cell library may lead to a poor quality. We address various issues regarding optimal generation of design-specific cell libraries, targeting high-performance digital circuit design.

The goal of the first part of the dissertation is to provide the key components required to successfully realize the automatic generation of design-specific cell libraries, which consists of cell logic type selection and drive strength type selection.

Chapter 2 addresses feasibility issues on transistor-level optimization. During transistor-level optimization, cell layout synthesis and characterization steps are the major bottlenecks with respect to runtime. To resolve this drawback, we present a fast and accurate prelayout estimation technique of cell characteristics. Our estimation technique is based on quick transistor placement. Given a transistor-level circuit of a cell, layout parasitics are estimated using quick transistor placement. Then, the cell is characterized by simulating an estimated circuit which is built according to the estimated layout parasitics. Experimental results on a 0.13μm industrial standard cell library demonstrate that the proposed technique estimates the cell characteristics with a reasonable accuracy in a negligibly small amount of time.

Chapter 3 addresses a cell logic type selection problem for design-specific cell libraries. Our methodology consists of two steps: logic-rich cell library generation and cell logic type count minimization. We propose a cell logic type count minimization method which minimizes the logic type count iteratively under performance constraints. Experimental results on the ISCAS 85 benchmark suite in an industrial 90nm technology demonstrate that it is feasible to find the minimal set of cell logic types under performance constraints.

Chapter 4 addresses a performance-constrained cell count minimization problem for continuously-sized circuits. After providing a formal formulation of the problem, we propose an effective heuristic for the problem. The proposed hill-climbing heuristic iteratively minimizes the number of cells under performance constraints such as area, delay and power. Experimental results on the ISCAS 85 benchmark suite in an industrial 90nm technology

demonstrate its effectiveness. We also discuss several implementation issues towards a practical application of the proposed method to large-scale circuits.

The second part of the dissertation focuses on transistor-level topology synthesis, which is an important component in the manual generation phase where portions of a circuit are manually identified and cells for the portions are synthesized at the transistor level. We present three transistor-level topology synthesis methods. Although their objectives are to minimize the transistor count, they have different solution spaces. Combining these methods, the minimum solution in larger solution space can be obtained.

Chapter 5 presents a method for synthesis of minimal static CMOS circuits where the solution space is restricted to the circuit structures which can be obtained by performing algebraic transformations on an arbitrary prime-and-irredundant two-level circuit. The circuit structures are implicitly enumerated via structural transformations on a single graph structure, then a dynamic-programming based algorithm efficiently finds the minimum solution among them. Experimental results on a benchmark suite targeting standard cell implementations demonstrate the feasibility of the proposed procedure. We also demonstrate the efficiency of the proposed algorithm by a numerical analysis on randomly-generated problems. It is also shown that the proposed procedure sometimes generates significantly smaller circuits compared to conventional approach.

Chapter 6 presents an exact method for minimum logic factoring which can be viewed as the synthesis of a static CMOS compound gate. We first introduce a novel graph structure, called an X-B (eXchanger Binary) tree, which implicitly enumerates binary trees. Using this X-B tree, the factoring problem is compactly transformed into a quantified Boolean formula (QBF) and is solved by general-purpose QBF solver. Experimental results on artificially-created benchmark functions show that the proposed method successfully finds the exact minimum solutions to the problems with up to 12 literals.

Chapter 7 studies the synthesis of read-once switch networks in which every variable appears only once. The proposed procedure is based on the notions of prime implicants and unateness, which establish a basis for Boolean expression synthesis. We also propose a pruning technique for an efficient search. Experimental results on randomly-generated problems with up to 20 switches demonstrate that the proposed procedure successfully solves about 90% of the problems in 10 minutes each and the resulting read-once switch networks are up to 78% smaller compared to series-parallel switch networks.

Chapter 8 conducts an experimental study using a circuit consisting of C432 and C499 from the ISCAS 85 benchmark suite as a design example. We compare the circuits synthesized with a typical cell library and optimal design-specific libraries in an industrial 90nm technology, and demonstrate that using the design-specific cell libraries, the area-delay tradeoff curve is shifted to the left-bottom from that using the typical library. Comparing between the area-optimal circuits, the area is improved by 27.3%. And, comparing between the delay-optimal circuits, the maximum delay is improved by 22.4%. These results clearly prove the effectiveness of the flow and the key components for optimal generation of design-specific cell libraries.

Acknowledgments

I would never have been able to write this dissertation without the support by many of my family members, friends and colleagues.

I would like to express my greatest gratitude to Prof. Kunihiro Asada, my supervisor for eight years since I was an undergraduate student, for his guidance, encouragement, support and constant belief in me. His deep insights into research problems and his incredibly broad vision have been invaluable in my development as a researcher. I also learned from him how to conduct a research independently; everything from the ability to identify a potential research area to technical writing and presentation skills. I feel truly fortunate to have studied under him.

I am deeply grateful to Prof. Makoto Ikeda for his guidance and support over the years. His in-depth knowledge on integrated circuit design helped me develop my research work further, and his devoted and continued effort to provide a comfortable environment was essential to make my research activities successful.

I would like to thank Prof. Masahiro Fujita for generously sharing invaluable ideas and providing me advices and feedbacks throughout my graduate life. He offered me many precious opportunities to work in the United States. Also, as my dissertation committee member, he gave me constructive feedbacks on my research.

I would like to give my thanks to Dr. Yusuke Oike, who was my research colleague for many years in my graduate student life. His determined attitude and enthusiasm for research has been always motivating me. The discussions with him have been enjoyable and intellectually stimulating. I have been fortunate to work closely with him.

I would like to thank Mr. Tetsuya Iizuka, who has been my recent research colleague after returning from two-year absence from school, for meaningful discussions and feedbacks on my research. I also owe tremendous gratitude to him for doing the thankless job at Asada & Ikeda Laboratory.

I would also like to acknowledge the members of my dissertation committee, Prof. Tadashi Shibata, Prof. Shuichi Sakai and Prof. Makoto Takamiya, for their constructive suggestions and feedbacks on my research.

The interactions with the current and past members of Asada & Ikeda Laboratory have been always stimulating. I would like to thank Ms. Noriko Yokochi, Ms. Naomi Yoshida, Prof. Tohru Ishihara, Mr. Ruotong Zheng, Dr. Satoshi Komatsu, Dr. Masahiro Sasaki, Dr. Toru Nakura, Dr. Tomohiro Nezuka, Dr. Hiroaki Yamaoka and other members at the laboratory. Special thanks to Mr. Yusuke Yachide for helping me prepare this dissertation.

I would like to thank the current and past members of Zenasis Technologies, Inc. I am deeply grateful to Dr. Vamsi Boppana, my supervisor at Zenasis, for having me play a key role at Zenasis and understanding my research activities. I also learned from him how to tackle practical problems in a systematic way. The work presented in Chapter 2 in this dissertation is based on a research collaboration with him; to Dr. Rajeev Murgai, who was my supervisor when I was working at Fujitsu Laboratories of America (FLA) for a summer internship and was also my colleague at Zenasis, for his kind support and valuable advices; Special thanks to Dr. Robert Carragher, who helped me adjust to the new environment when I was working at FLA and Zenasis. He has also been my best teacher of dancing as well as English language.

I am grateful to Dr. Yuji Kukimoto for his extremely valuable advices and comments from his professional perspective.

I would like to thank Prof. Tsutomu Sasao, Prof. Yusuke Matsunaga and Prof. Shigeru Yamashita for their precious suggestions and interests on my research.

I am thankful to Fujitsu Laboratories of America for hiring me for a summer internship. I also owe gratitude to all the members of VLSI Design and Education Center (VDEC), the University of Tokyo, for supporting industrial EDA tools. The work presented in this dissertation is supported by VDEC in collaboration with Synopsys, Inc. and Cadence Design Systems, Inc.

Finally, I would like to express my gratitude to my parents, Naliaki and Reiko, and my sister Naoko for their continual support and understanding throughout my life. Also thanks to my friends, Ryo Osada and Hiroaki Kubo, who always encourage and support me in my hour of need.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In spite of the significant advances of computer-aided design tools for large-scale integrated circuit (LSI) over the decades, there has been a large performance gap between application-specific integrated circuit (ASIC) and custom LSI. Figure 1.1 compares the clock frequencies of high-performance custom and ASIC processors. For instance, Intel Pentium 4 processor was designed in custom design methodology and operates at 2GHz in 0.18μm technology. In contrast, Sony/Toshiba Emotion Engine which was designed in ASIC methodology operates at 300MHz in 0.18μm technology. Although the clock frequency does not necessarily represent the performance of a chip, examining the factors contributing this frequency gap may provide a clue to improving the ASIC performance further and closing the gap between ASIC and custom LSIs. Recently, a number of studies on the examination of the factors [CK00, CNK01, RPS01, CK02] have been presented. In summary, they claimed that there are various factors contributing to the gap as shown in Table 1.1. Based on this background, many researchers in various fields of computer-aided design have been tackling the problems. In particular, design optimization at the transistor level has been gaining an attention because it can improve most of the factors at a time.

Design optimization at the transistor level is well-known and has been successfully used to achieve significant performance benefits above and beyond gate-level design optimization. The approaches range from transformations such as sizing [FD85, CEWWM+99], all the way to macro-cell based design methodologies [BF98]. More recently, transistor-level optimization techniques targeting standard cell based design flow have also been proposed [PDE+98, BB02]. The techniques aim to gain an equivalent performance improvement to transistor-level optimization by enhancing the cell library. These optimization techniques take advantage of the recent progress in automated cell-layout solutions [GH97, GMD+97, SS99, RS03, abr03].

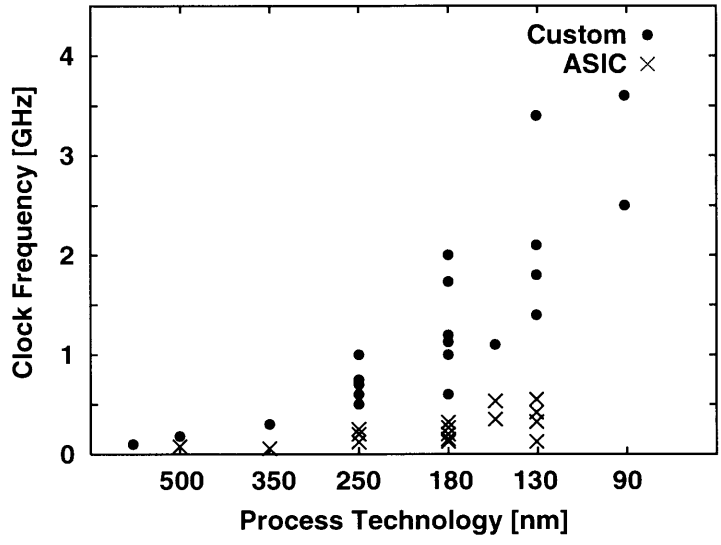


Figure 1.1 Clock frequencies of high-performance ASIC and custom processors [CK02].

Table 1.1 Factors contributing to the gap between ASICs and custom [CK02].

Factor	Difference
Microarchitecture	x1.80
Timing overhead	x1.45
High speed logic styles	x1.40
Logic design	x1.30
Cell design	x1.45
Layout	x1.40
Process variation	x2.00

This approach can also be viewed as an automation of the creation of design-specific macro-cells which are typically manually crafted. However, due to several major drawbacks and limitations, the practical use of this incremental optimization approach is restricted. The first drawback is the limitation on the cell creation cost. The creation of cells consists of several computationally-intensive processes including cell layout synthesis and characterization. In an ideal circuit, every gate is optimized at the transistor level and hence the cell creation cost can be considerable for large-scale circuits. The second drawback is that the approach requires a complete set of a cell library as an initial cell library. If we could somehow construct a design-specific cell library from scratch, it would not be necessary anymore. The last drawback is the lack of optimal transistor-level synthesis methods. Although optimal sizing algorithms have been extensively studied, optimal transistor-level synthesis is still a challenging problem. To gain the maximum benefits from transistor-level optimization, the development of optimal transistor-level synthesis methods is essential.

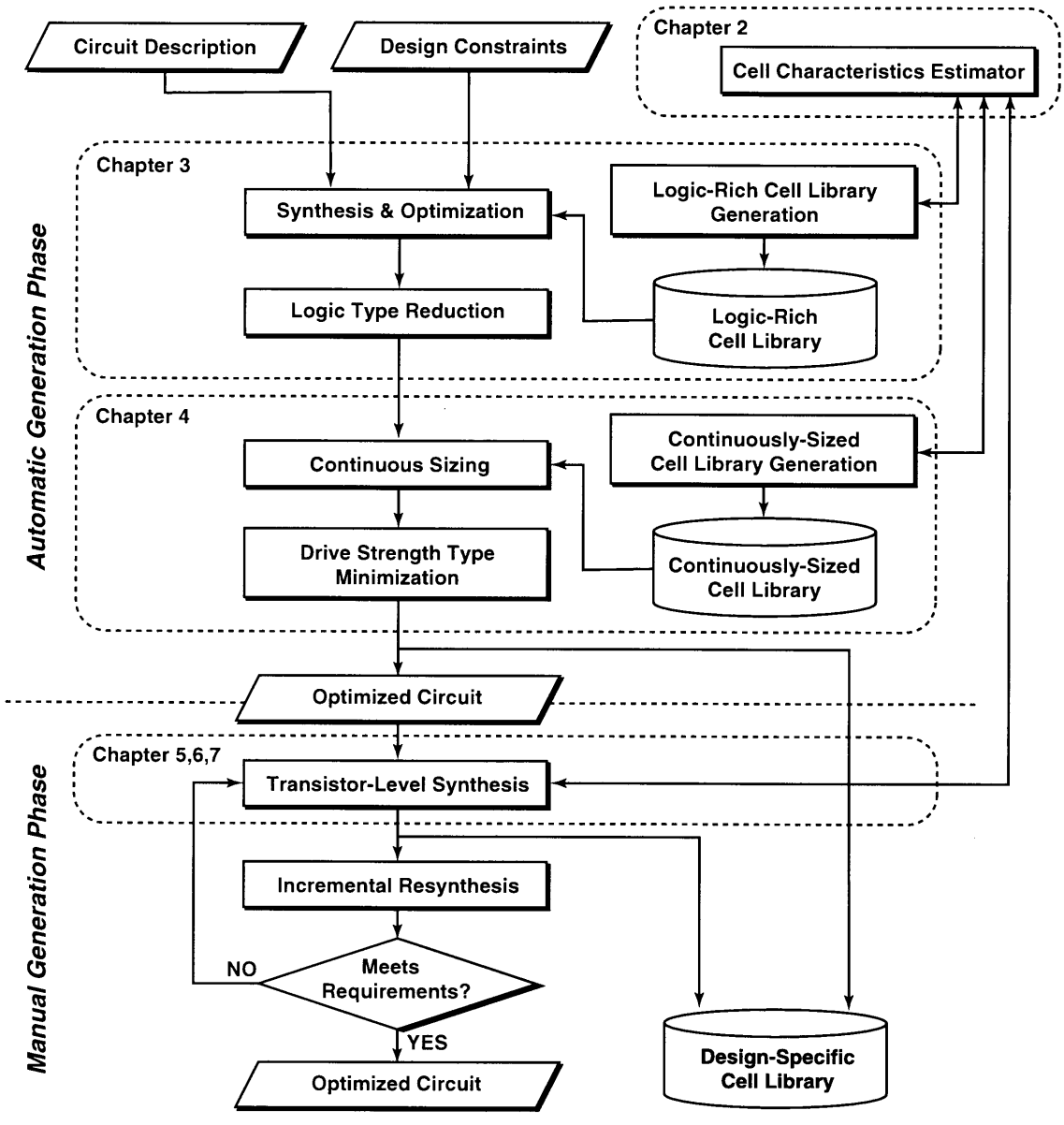


Figure 1.2 Overall flow for optimal generation of design-specific cell libraries.

1.2 Objectives and Organization of This Thesis

The focus of this dissertation is optimal generation of design-specific cell libraries. Figure 1.2 illustrates the overall flow for generating optimal design-specific libraries. The flow is divided into two phases: *automatic generation phase* and *manual generation phase*. The first phase is the automatic generation phase. Given an initial circuit description and a set of design constraints, an initial circuit is synthesized with a logic-rich cell library generated by the method described in Section 3.2. The logic types are reduced under the design constraints (Section 3.3), and the minimal set of logic types for the design-specific library is obtained. Next, the circuit is continuously sized by an optimal continuous sizer (Section 4.2.2) and then the total number of cells is

minimized under the design constraints by minimizing the drive strength count for each logic type. Thus, the optimal design-specific library and the optimized circuit using the library are automatically obtained. If the design requirements are not met at this point, the manual generation phase is performed as follows. A portion of the circuit is identified manually, and then the optimal cell for the portion is synthesized by the transistor-level synthesis methods proposed in Chapter 5, Chapter 6 and Chapter 7. This step is repeated until the requirements are met. Every optimization step in this flow includes the cell characteristics evaluation upon which the overall runtime and the final quality depend. The fast and accurate evaluation of cell characteristics is accomplished by the prelayout cell characteristics estimation method in Chapter 2. Thus, all components proposed in this dissertation are combined together in this flow.

The goal of the first part of the dissertation is to provide the key components required to successfully realize the automatic generation phase (the upper half part of the overall flow in Figure 1.2).

Chapter 2 will present a fast and accurate prelayout estimation technique of cell characteristics. During transistor-level optimization, the cell layout synthesis and characterization steps are the major bottlenecks with respect to runtime. Besides, the convergence of the optimization depends upon the accuracy of cell characteristics. Our estimation technique is based on quick transistor placement. Given a transistor-level circuit of a cell, the layout parasitics are estimated using quick transistor placement. Then, the cell is characterized by simulating an estimated circuit which is built according to the estimated layout parasitics. The proposed technique establishes a basis for the transistor-level optimization methods presented in the following chapters.

Chapter 3 will present a methodology for optimal selection of logic types for design-specific cell libraries. The proposed methodology can be used as the logic type selection in the automatic generation phase. The methodology is divided into two major steps. The first step is to prepare a cell library with a rich variation of logic types. The second step is to find a minimal subset of the logic types subject to performance constraints such as area, delay and power. Thus, the minimal set of logic types specific to a design is obtained. In this chapter, we will provide an effective method for each step.

Chapter 4 will address a performance-constrained cell count minimization problem for continuously-sized circuits. A continuously-sized circuit resulting from transistor sizing consists of gates with large variety of sizes. This leads to an increase of the library creation cost,

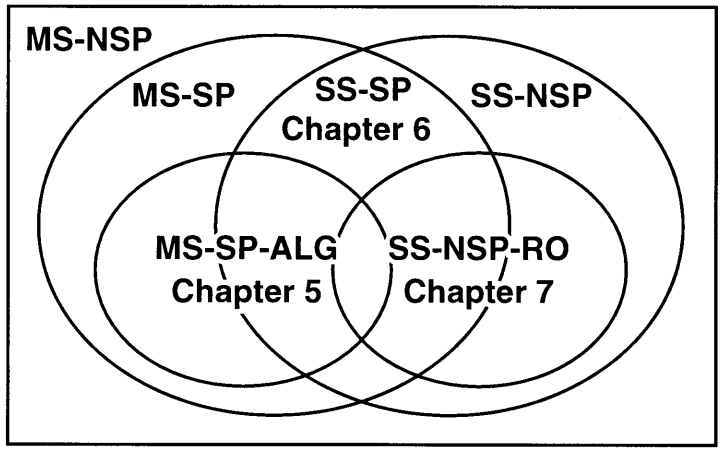


Figure 1.3 A Venn diagram which informally illustrates the relationships between the solution spaces of the proposed transistor-level synthesis methods where MS stands for multiple-stage, SS for single-stage, SP for series-parallel, NSP for non-series-parallel, ALG for algebraic and RO for read-once.

which renders the approach to be impractical particularly for large-scale designs. After providing a formal formulation of the problem, we will propose an effective heuristic for the problem. The proposed hill-climbing heuristic iteratively minimizes the number of cells under performance constraints such as area, delay and power. The method can be used as the drive strength type selection in the automatic generation phase.

The second part of the dissertation focuses on transistor-level topology synthesis, which is an important component in the manual generation phase (the lower half portion of the flow in Figure 1.2). In spite of the fact that exact transistor and gate sizing algorithms have been well studied, few studies have been carried out on exact transistor-level synthesis. We will provide exact transistor-level topology synthesis methods with different solution spaces. Figure 1.3 informally illustrates the relationships between the solution spaces of the proposed transistor-level synthesis methods. In the figure, MS stands for multiple-stage, SS for single-stage, SP for series-parallel, NSP for non-series-parallel. SS-SP is the set of single-stage series-parallel circuits. MS-SP-ALG is the set of multiple-stage series-parallel circuits obtained by performing algebraic transformations on an arbitrary prime-and-irredundant two-level circuit. SS-NSP-RO is the set of single-stage non-series-parallel read-once circuits in which every variable appears only once. More detailed explanations on the solution spaces will be given in the following chapters.

Chapter 5 will present a method for synthesis of minimal static CMOS circuits. To make the problem tractable, the solution space is restricted to the circuit structures which can be ob-

tained by performing algebraic transformations on an arbitrary prime-and-irredundant two-level circuit. The circuit structures are implicitly enumerated via structural transformations on a single graph structure, then a dynamic-programming based algorithm efficiently finds the minimum solution among them.

Chapter 6 will present an exact method for minimum logic factoring. Logic factoring can be viewed as the synthesis of a static CMOS compound gate. We will first introduce a novel graph structure, called an X-B (eXchanger Binary) tree, which implicitly enumerates binary tree structures. Using this X-B tree, the factoring problem is compactly transformed into a quantified Boolean formula (QBF) and is solved by general-purpose QBF solver.

Chapter 7 will study the synthesis of read-once switch networks. The ultimate goal is to synthesize a non-series-parallel switch network representing a given Boolean function with the minimum number of switches. As a first step towards the goal, we will focus on the synthesis of a read-once switch network in which every variable appears only once. The proposed procedure is based on the notions of prime implicants and unateness, which establish a basis for Boolean expression synthesis. We will also propose a pruning technique for an efficient search.

Chapter 8 will conduct an experimental study using a benchmark circuit. The objective of the case study is to demonstrate the effectiveness of the overall flow for generating optimal design-specific cell libraries. We will compare the circuits synthesized with a typical cell library and optimal design-specific libraries, and will demonstrate that using the design-specific cell libraries, the area-delay tradeoff curve is shifted to the left-bottom from that using a typical library. The results indicate that the intrinsic improvement can be achieved by the proposed flow.

Finally, the dissertation will be concluded in Chapter 9.

Chapter 2

Cell Characteristics Estimation Using Quick Transistor Placement

2.1 Introduction

Design optimization at the transistor level is well-known and has been successfully used to achieve significant performance benefits above and beyond gate-level design optimization. The approaches range from transformations such as sizing [FD85, CEWWM+99], all the way to macro-cell based design methodologies [BF98]. More recently, transistor-level optimization techniques targeting standard cell based design flow have also been proposed [PDE+98, BB02]. These optimization techniques take advantage of the recent progress in automated cell-layout solutions [GH97, GMD+97, SS99, RS03, abr03]. Figure 2.1 illustrates the basic flow of such a transistor-level optimization technique.

In this optimization flow, since a large number of transistor-level circuits are explored to find an optimum solution, the overall runtime heavily depends on the cell characterization process. Conventional transistor-level optimization techniques, such as [FD85, SRVK93], have not attempted to account for the impact of layout parasitics, which has become increasingly significant at the deep-submicron technologies [Sem04]. Due to the inaccuracy imposed by disregarding the layout parasitics, the optimization may take a long time to converge, or just doesn't converge. One naive approach to this problem is to incorporate the actual layout synthesis and subsequent extraction processes into the flow. This approach is, in fact, infeasible because the layout synthesis and extraction processes consume a significant amount of time. Consequently, cell characteristics must be estimated without actually performing the detailed layout and subsequent extraction steps.

In this chapter, we propose a novel approach based on a cell characteristics estimation. The cell characteristics estimator based on this approach can be used at every decision step in the flow for generating design-specific cell libraries illustrated in Figure 1.2. Given a transistor-

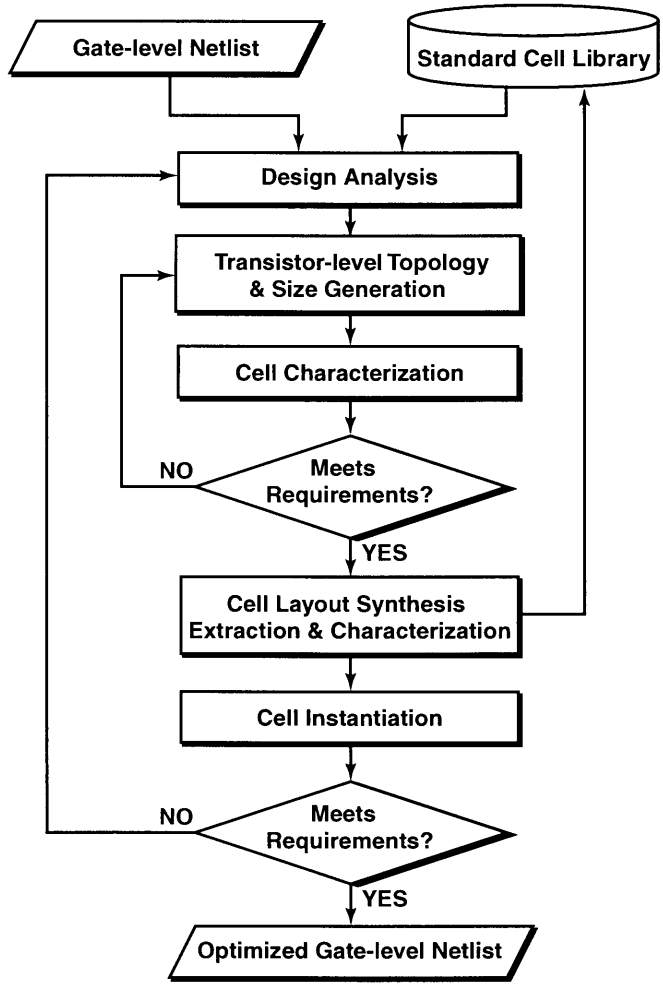


Figure 2.1 Transistor-level optimization flow targeting standard cell based design flow.

level circuit of a cell, the layout parasitics are estimated using quick transistor placement. Then, the cell is characterized by simulating an estimated netlist which is built according to the estimated layout parasitics. Our approach can be interpreted as an analogy to the route estimation technique at the inter-cell level which is commonly used in physical synthesis [SID+99, CMC+01]. In this technique, a physical design space is divided into multiple bins and each cell is assigned to one of the bins. The routing of each net is estimated by a steiner tree connecting the bins which include the terminals of the net. The objective of this work is to show that such an approach works even at the intra-cell level.

The rest of the chapter is organized as follows. Section 2.2 and Section 2.3 describe the proposed quick transistor placement and the cell characteristics estimation techniques, respectively. Section 2.4 presents experimental results on an industrial standard cell library in 0.13μm technology. In Section 2.5, the impact of intra-cell layout parasitics and its future trends are discussed. Conclusions are drawn in Section 2.6.

2.2 Quick Transistor Placement

The quick transistor placement algorithm is required to satisfy the following properties: (a) *simple and flexible* and (b) *the runtime is substantially small.* Although the transistor placement problem has been well studied up until now [GH97, GMD+97, SS99, RS03], we do not need a complete placement but an estimate of placement. (a) implies that the algorithm must be as independent as possible from real design constraints such as design rules and layout styles. As a rule of thumb, a simple algorithm applies well to a wide range of technologies. As for (b), more specifically, the runtime is required to be substantially small compared to that of the simulation.

To achieve the requirements, the algorithm is designed utilizing the two techniques: *bi-partitioning based placement technique* and *hierarchical placement technique.* The bi-partitioning based placement technique [Bre77] is very simple and runs very fast if an efficient partitioning algorithm is chosen. The hierarchical placement technique first divides a transistor-level circuit into stages and builds a hierarchical network consisting of two levels: *stage level* and *transistor level.* Then, the placement at the stage level is performed followed by the placement at the intra-stage level. This hierarchical approach has three major advantages as follows. First, it can lead to a reasonable placement. Typically, transistors in a stage are placed closely each other to maximize the connections by diffusions. Second, the computation complexity can be reduced, particularly, when a circuit has a large number of transistors. At each level of hierarchy, the size of the placement problem is smaller than when the problem is solved flatly. The third advantage is an incremental update of the placement. A typical transistor-level optimizer changes very small portion of a circuit at a time. As long as the topology at the stage level remains the same, it's sufficient to update the placement of transistors inside the touched stages.

2.2.1 Hierarchical Network Construction

The construction of a hierarchical network starts with transistor folding. Since the height of a cell is fixed, wide transistors in a pre-layout netlist are divided into smaller transistors to meet the cell height. The folded transistors are connected in parallel to preserve the original functionality. Our model allows two transistor folding styles, a *fixed P/N ratio* style and an *adaptive P/N ratio* style. In the fixed P/N ratio style, the maximum widths of P-type and N-type transistors are fixed for all cells. In the adaptive P/N ratio style, the maximum widths for each cell are determined such that the width of the cell is minimized. In either style, each

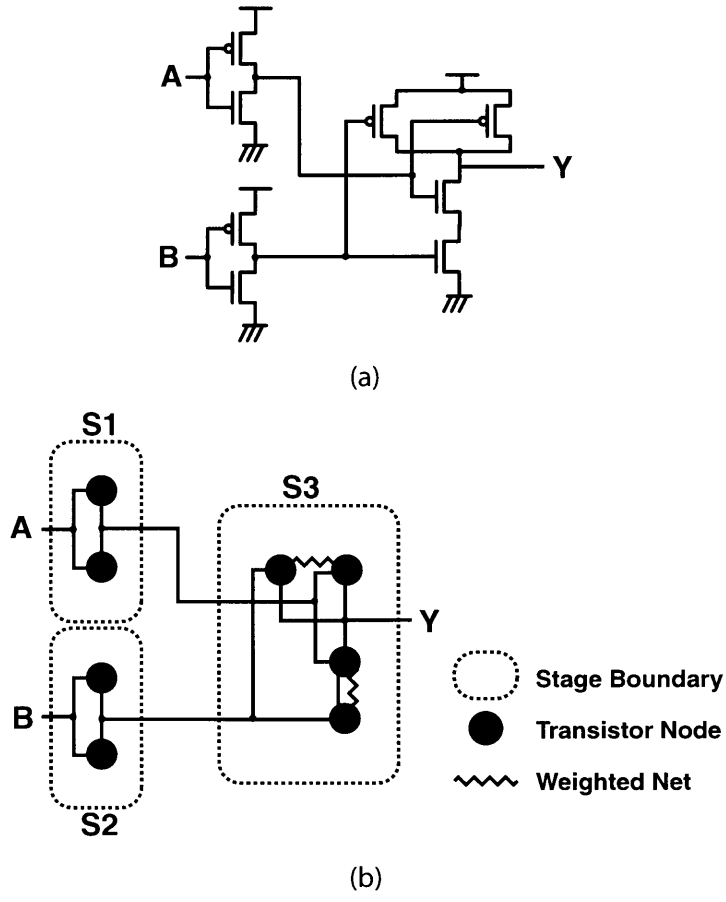


Figure 2.2 (a) Transistor-level circuit of 2-input OR gate and (b) its hierarchical network.

transistor is divided equally to meet the maximum width.

Next, a hierarchical network is constructed by dividing the netlist into stages. A number of techniques for the stage recognition have been proposed as a part of gate-level extraction techniques [Boe88, Bry91]. In this chapter, we define a stage as a Channel Connected Component (CCC). A CCC is the maximum set of transistors connected at the sources and drains each other. As an exception, a transmission gate is recognized as a distinct stage. If any unknown structure is found in the netlist during the stage recognition, the hierarchization is canceled, *i.e.*, the netlist remains flat. In such a case, the stage placement phase will be skipped.

Finally, weighted nets are added in the hierarchical network as follows. Nets connecting two or more drains/sources of the same transistor type are identified. The nets are weighted by adding new nets connecting the corresponding drains and sources. This net-weighting is used to pull transistors which share diffusions together, duplicating what a human designer does to save area. Figure 2.2 (a) shows a transistor-level circuit of 2-input OR gate and Figure 2.2 (b) shows the corresponding hierarchical network. In the figure, the weighted nets are drawn as wiggle lines.

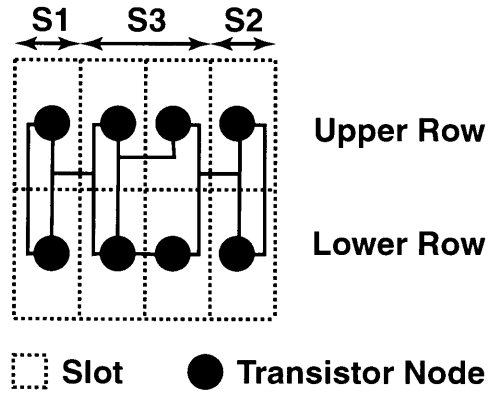


Figure 2.3 Placement model.

2.2.2 Stage Placement

In a stage-level network, each node corresponds to a stage. The stage placement problem can be viewed as a one-dimensional placement problem of a stage-level network. As already mentioned, we use the bi-partitioning placement technique to solve this problem. Assuming that the stages are placed horizontally, we are interested only in the horizontal order of nodes, *i.e.*, whether a node is on the left or right of another node. Figure 2.3 illustrates our placement model. In the figure, the stage *S1* is placed on the left, *S2* on the middle, and *S3* on the right. At each partitioning step, the partitioned sets of nodes are simply assigned to either the left set L or the right set R of nodes. As a bi-partitioning algorithm, we chose the well-known F-M algorithm [FM82] due to its linear runtime complexity. A net is said to be a cut if it is connected to at least one node in each set. The size of a cut is defined as the number of cuts. The gain of a node is defined as the change in the cut size when the node is moved into the other set. To balance the two sets of nodes, the following conditions must be satisfied:

$$-1 \leq \frac{|L| - |R|}{2} \leq 1 \tag{2.1}$$

The F-M algorithm finds a partition of nodes such that the cut size between the two sets is minimized. Figure 2.4 shows a basic procedure of the F-M algorithm.

2.2.3 Intra-Stage Placement

This phase determines the placement of transistors in each stage. In a transistor-level network, each node corresponds to either a transistor or an external stage. An external stage is defined as a stage other than the enclosing stage. A transistor node is assigned to either the upper or lower row depending on its transistor type. Every external stage node is placed and fixed at either the left or right of the placement space for the intra-stage transistors.

F-M Algorithm

Input: Hypergraph $H(V, E)$

Output: Vertex ordering: $v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}$

```
1: Generate an initial partition (L, R)
2: repeat
3:    repeat
4:       Pick up the best nodes from L and R
5:       Throw away the nodes that don't satisfy the constraint 2.1 after moving
6:       Choose a node that has the highest gain
7:       Move the node to the other side and lock it
8:    until all nodes are locked
9:    Unlock all the nodes
10: until no further improvement can be made
```

Figure 2.4 F-M algorithm.

In our placement model shown in Figure 2.3, a placement space is divided vertically into two rows, upper row and lower row, and horizontally into a number of columns. Each divided space is referred to as a slot. At most one transistor can be assigned to a single slot. P-type transistors are assigned to the upper row, and N-type transistors to the lower row. The leftmost and rightmost slots are reserved for external stages, and can accept two or more external stages.

Since the placement problem of a transistor-level network is not simply a one-dimensional placement problem, the F-M algorithm cannot be used in a straightforward manner. Therefore, we extend the algorithm for the double row placement style. A basic procedure is similar to the original except the differences as follows. First, the partition is represented by the four sets of nodes, L_U, L_L, R_U and R_L. Here, the subscript U denotes the upper row and L denotes the lower row. Second, the balance between two sets of nodes is re-defined as follows:

$$-1 \leq \frac{|L_D| - |R_D|}{2} \leq 1 \text{ and } L_D \geq L_N \text{ and } R_D \geq R_N \tag{2.2}$$

where L_D, L_N, R_D, R_N are:

$$(L_D, L_N, R_D, R_N) = (L_U, L_L, R_U, R_L) \qquad \text{if } |L_U| + |R_U| \geq |L_L| + |R_L| \tag{2.3}$$

$$(L_D, L_N, R_D, R_N) = (L_L, L_U, R_L, R_U) \qquad \text{if } |L_U| + |R_U| < |L_L| + |R_L| \tag{2.4}$$

The subscript D indicates that the row is *dominant*, and N indicates that the row is *non-dominant*. The procedure of this algorithm is shown in Figure 2.5.

Double-Row F-M Algorithm

Input: Hypergraph $H(V, E)$
Upper-row vertices: $V^U \subset V$
Lower-row vertices: $V^L \subset V$
$V^U \cup V^L = V$

Output: Upper-row vertex ordering: $v^U_{\pi(1)}, v^U_{\pi(2)}, \ldots, v^U_{\pi(n^U)}$
Lower-row vertex ordering: $v^L_{\pi(1)}, v^L_{\pi(2)}, \ldots, v^L_{\pi(n^L)}$

```
1: Generate an initial partition (L_U, L_L, R_U, R_L)
2: repeat
3:    repeat
4:       Pick up the best nodes from L_U, L_L, R_U and R_L
5:       Throw away the nodes that don't satisfy the constraint 2.2 after moving
6:       Choose a node that has the highest gain
7:       Move the node to the other side and lock it
8:    until all nodes are locked
9:    Unlock all the nodes
10: until no further improvement can be made
```

Figure 2.5 Proposed double-row F-M algorithm.

2.2.4 Detailed Placement

Up to this point, the horizontal order of transistors is known. In this phase, the X and Y coordinates of the transistors are determined. The X coordinate of the leftmost transistors is assumed to be zero. The distance between two adjacent transistors is categorized into three types as illustrated in Figure 2.6. The distance value of each type is given as a technology parameter specific to the cell layout architecture. Given two adjacent transistors, the type is determined according to their connectivity. If the two transistors are not connected by drain(s) and/or source(s), the type is of Figure 2.6 (a). If the transistors are connected by drain(s) and/or source(s) and the corresponding net is not connected to any other transistor, the type is of Figure 2.6 (b). Otherwise, the type is of Figure 2.6 (c). Thus, the X coordinates of the transistors are determined in a left-to-right manner. The Y coordinate of a transistor is Y_{trans} if the transistor is P-type and $-Y_{trans}$ otherwise, where Y_{trans} is a constant value specific to the cell layout architecture.

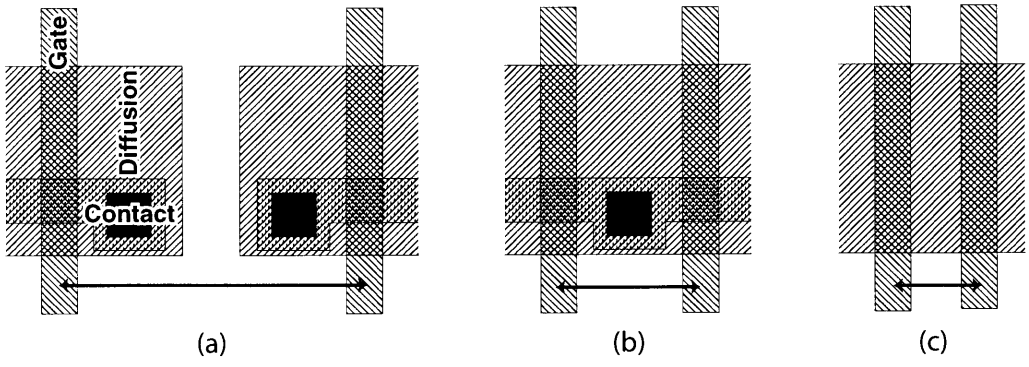


Figure 2.6 Three types of distance between two adjacent transistors: (a) two transistors with two contacts and a diffusion gap in between, (b) two transistors sharing diffusion with a contact in between, and (c) two transistors sharing diffusion.

2.2.5 Runtime Complexity

Let N_N be the number of nodes and N_T be the number of terminals in a network. According to the original paper [FM82], the runtime complexity of F-M algorithm is $O(N_T)$. Likewise, the complexity of the modified F-M algorithm is approximated by $O(N_T)$. Hence, the runtime complexity of the bi-partitioning placement is $O(N_T \log N_N)$. The detailed placement has a runtime complexity of $O(N_N)$. Since the number of transistors is at least twice of the number of stages, the runtime complexity of the quick placement algorithm is approximated by $O(N_{TA} \log N_{TS})$ where N_{TA} is the number of transistors and N_{TS} is the typical number of transistors in a stage.

2.3 Cell Characteristics Estimation

For the purpose of static timing analysis and design optimization, a cell is typically characterized with respect to the following parameters: *timing*, *power*, *area* and *input capacitances*. This section describes estimation techniques of these cell characteristics based on the placement information obtained by the quick placement engine.

2.3.1 Timing/Power

The cell delay model most widely used now is the *Non-Linear Delay Model* (NLDM) [Lib03]. In NLDM, a cell timing is modeled by lookup tables with output load and input transition time as indices. Likewise, a cell power is typically modeled in the same way. Each entry in a lookup table is obtained by simulating a transistor-level netlist under a specific output load and input slope. Note that the proposed technique can be applied for any other delay/power models as

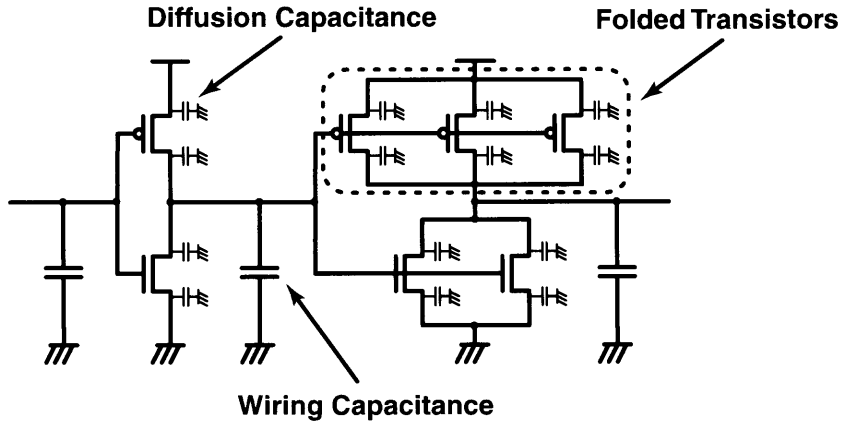


Figure 2.7 An example of an estimated netlist.

long as the model can be built based on simulated values.

A timing is estimated by simulating an estimated netlist, instead of an extracted netlist which is not available at the pre-layout level. An estimated netlist consists of the following elements: *folded transistors*, *diffusion capacitances* and *wiring capacitances*. For example, Figure 2.7 illustrates an example of an estimated netlist.

Diffusion Capacitances

Typical SPICE MOSFET models [L+98] calculate a diffusion capacitance according to the area and perimeter of a diffusion region. Let us assume that a diffusion is in a rectangular shape. The height is estimated as the width of the transistor associated with the diffusion. The estimated width of the diffusion is available during the detailed placement phase described in Section 2.2.4.

Wiring Capacitances

Given a wire connecting a set of transistors, a steiner tree is first constructed to estimate the total length of the metal portion of the wire. The metal wire connects the transistors at drains and/or sources and the contact points. Supposing that a wire is connected to a transistor t at its gate, a contact point is defined as the intersecting point of the X axis (the center line of the upper and lower rows) and the vertical line passing through the transistor t. Figure 2.8 shows an example of steiner tree for the metal wiring estimation. A large number of steiner tree construction algorithms have been proposed. In particular, we used [KR92] which is known

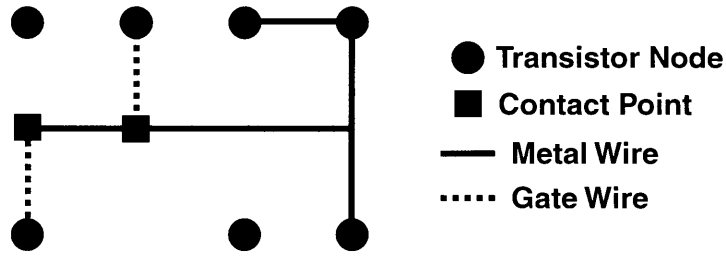


Figure 2.8 An example of steiner tree for metal wiring estimation.

as one of the state-of-the-art algorithms. Finally, the wiring capacitance is estimated as follows:

$$C_{wire} = k_{wire}(C_{metal}L_{tree} + C_{poly}N_{gate}) \tag{2.5}$$

where k_{wire} is a technology-specific constant value, C_{metal} is a metal-to-ground capacitance per unit length, L_{tree} is the total length of the steiner tree, C_{poly} is the poly capacitance per an individual gate, and N_{gate} is the number of gates connected to the wire. k_{wire} is introduced as a scaling factor to take account of the effect of wire-to-wire capacitances and wiring resistance. This needs to be calibrated according to the technology and cell layout architecture. C_{poly} is the poly capacitance of the wire except the portion of transistor gate.

2.3.2 Area

The area of a cell is estimated by the following formula:

$$A = H_{cell}(W_{LR} + 2W_{edge}) \tag{2.6}$$

where H_{cell} is the cell height, W_{LR} is the distance from the leftmost transistor to the rightmost transistor, W_{edge} is the distance from the leftmost(rightmost) transistor to the left(right) edge of the cell boundary.

2.3.3 Input Capacitances

An input capacitance of a cell can be modeled as the sum of gate capacitance and wiring capacitance. Thus, the input capacitance at an input i of a cell is estimated as follows:

$$C_{input}(i) = \Big(\sum_{t \in T(i)} C_{gate} W_{trans}(t) \Big) + C_{wire}(i) \tag{2.7}$$

where $T(i)$ is the set of transistors connected to i, C_{gate} is a unit gate capacitance, and $W_{trans}(t)$ is the width of the transistor t, and $C_{wire}(i)$ is the capacitance of the wiring associated with i.

Table 2.1 Summary of 0.13μm industrial standard cell library. All cells are single-output cells.

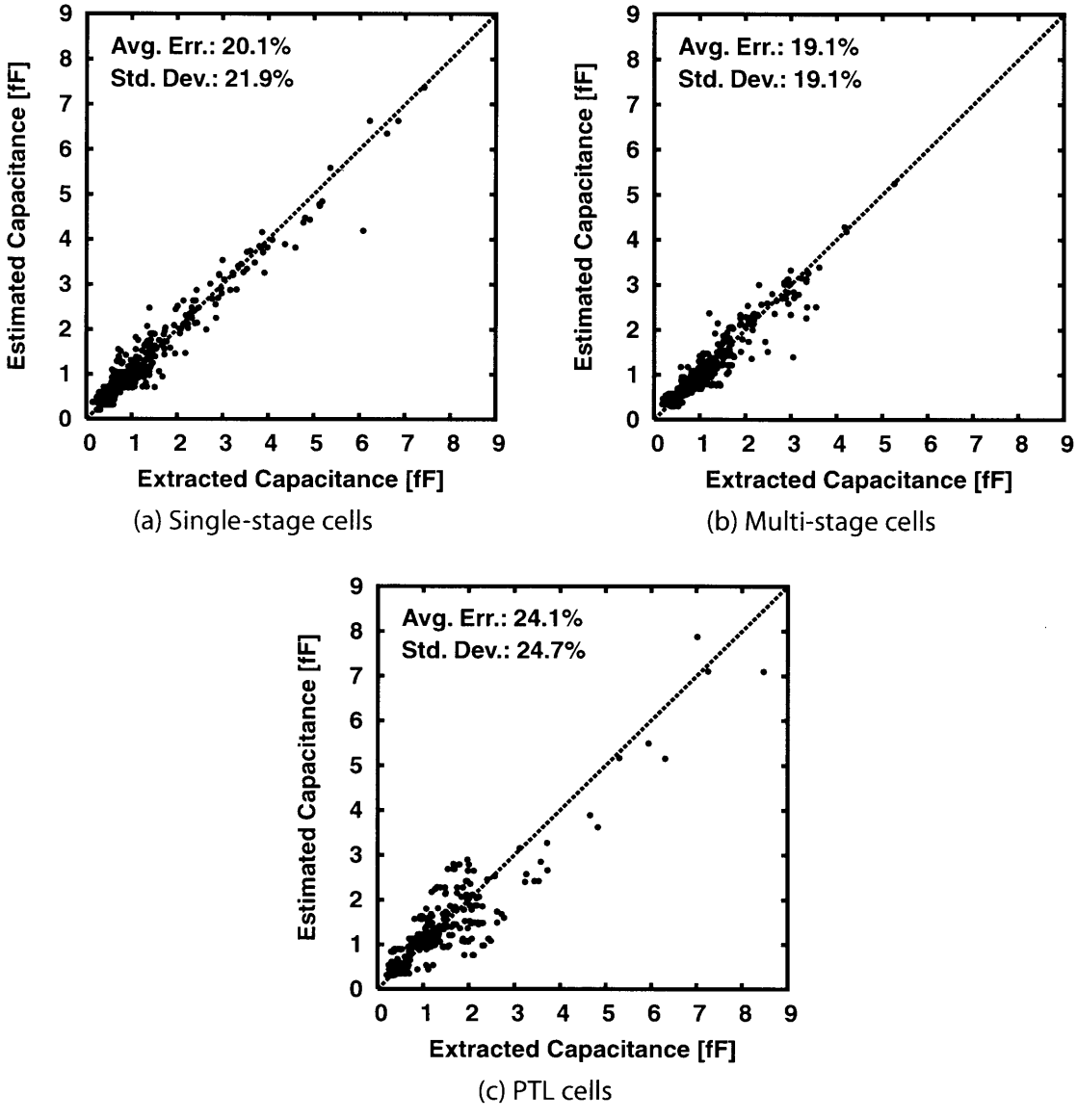
Cell Type	#cells	#inputs		#wires		#transistors (unfolded)		Max. #stages	Max. #logic levels
		Min.	Max.	Min.	Max.	Min.	Max.		
Single-stage Cells	119	1	6	2	9	2	36	1	1
Multi-stage Cells	122	1	6	3	7	4	24	3	3
PTL Cells	54	2	4	5	18	10	66	12	4
Overall	295	1	6	2	18	2	66	12	4

2.4 Experimental Results

The proposed quick transistor placement and cell characteristics estimation technique have been implemented within the framework of a standard cell characterization flow. The framework is built using HSPICE [HSP03] as a transistor-level simulator and Calibre xRC [Cal04] as a parasitic extractor. Using this framework, we conducted experiments on a state-of-the-art standard cell library implemented in 0.13μm technology. The summary of the library is shown in Table 2.1. In the table, the number of logic levels is defined as the maximum number of stages on any input-to-output path. The cells are categorized into three types: *single-stage cells*, *multi-stage cells* and *pass transistor logic (PTL) cells*. A PTL cell is defined as a cell including pass-transistor logic, *e.g.*, multiplexer and XOR gates in this library are PTL cells. As can be noticed, some single-stage cells have a large number of transistors. This is because such cells have multiple gates connected in parallel to gain the drive strength. From the definition of the CCC, such cells are recognized as single-stage cells.

First, we compared the estimated and extracted values of wiring capacitances inside the cells. The extracted capacitances are obtained as follows. First, lumped C netlists are extracted using Calibre xRC. The capacitance of each wire is obtained by summing up all capacitances associated with the wire, *i.e.*, the sum of a wire-to-ground capacitance and wire-to-wire capacitances. Figure 2.9 shows the comparison of the extracted and estimated capacitances in (a) the single-stage cells, (b) the multi-stage cells, and (c) the PTL cells. The average errors and the standard deviations are shown in the figures. The table shows that the average error is up to 24.1%. Since the impact of the wiring capacitance on the timing is comparably small ($\sim$ 10%), the estimation is accurate enough.

Next, we estimated the timing of the cells using the framework and compared them against the timing obtained by simulating extracted netlists. The extracted netlists consist of folded



(a) Single-stage cells

(b) Multi-stage cells

(c) PTL cells

Figure 2.9 Comparisons of extracted and estimated capacitances.

transistors, diffusion areas and perimeters, wiring resistances and capacitances. Originally, the framework characterizes the cell timing for the non-linear delay model. We employed, however, a linear model which is more suitable to understand the contributions of input transition time and output load to the total delay. The linear delay model calculates a cell delay as follows:

$$t_{delay} = t_{int} + k_{trans}t_{trans} + k_{load}C_L \tag{2.8}$$

where t_{trans} is an input transition time and C_L is an output load. This translation from the non-linear model to the linear model was done by the multiple regression analysis within the typical ranges of output load and input transition time. A timing error is calculated as

Table 2.2 Timing error without layout parasitics.

Cell Type	Intrinsic Delay t_{int}		Input Transition Dependence k_{trans}		Output Load Dependence k_{load}	
	Avg. Err. [%]	Std. Dev. [%]	Avg. Err. [%]	Std. Dev. [%]	Avg. Err. [%]	Std. Dev. [%]
Single-stage Cells	6.7	8.7	2.8	3.8	3.1	5.5
Multi-stage Cells	12.1	15.4	2.9	3.9	4.2	5.7
PTL Cells	17.1	20.2	2.3	3.6	5.1	6.8
Overall	10.7	14.1	2.7	3.8	3.9	5.8

Table 2.3 Timing error with estimated layout parasitics.

Cell Type	Intrinsic Delay t_{int}		Input Transition Dependence k_{trans}		Output Load Dependence k_{load}	
	Avg. Err. [%]	Std. Dev. [%]	Avg. Err. [%]	Std. Dev. [%]	Avg. Err. [%]	Std. Dev. [%]
Single-stage Cells	2.9	3.9	1.7	2.4	1.5	2.9
Multi-stage Cells	2.3	3.2	1.4	2.1	1.4	2.4
PTL Cells	2.9	4.0	1.7	3.2	1.6	2.6
Overall	2.7	3.7	1.6	2.5	1.5	2.7

the relative error against the extracted timing. Table 2.2 and Table 2.3 show the timing error without and with layout parasitics estimated by the proposed technique, respectively. Without considering layout parasitics, the intrinsic delay error of the PTL cells is 17.1% on average. The error improved to 2.9% by the proposed estimation technique. Overall, the average error of intrinsic delay improved from 10.7% to 2.7%, and the average errors of k_{trans} and k_{load} improved to 1.6% and 1.5%, respectively. Again, note that the linear delay model is used only *for the comparison* and any timing model can be used as long as the model can be built by simulating transistor-level circuits with parasitics.

Figure 2.10 plots the actual and estimated cell areas and Figure 2.11 plots the actual and estimated input capacitances. The average errors and the standard deviations are shown in the figures. The CPU time required for the estimation of the entire library, excluding the simulation, was less than one second, while the simulation time was on the order of hours. The results show that the proposed technique is feasible to estimate cell characteristics with a reasonable accuracy.

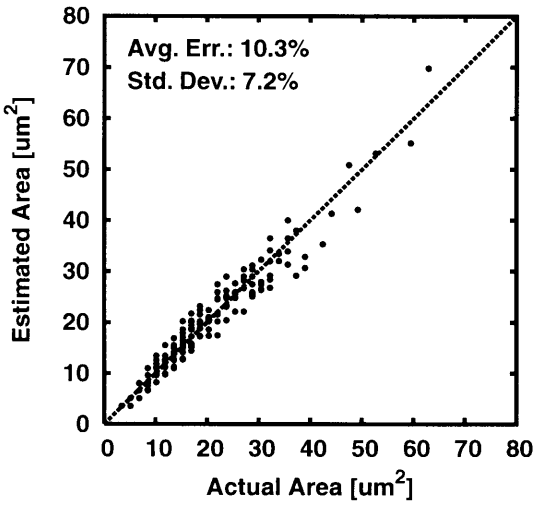


Figure 2.10 Comparison of actual and estimated cell areas.

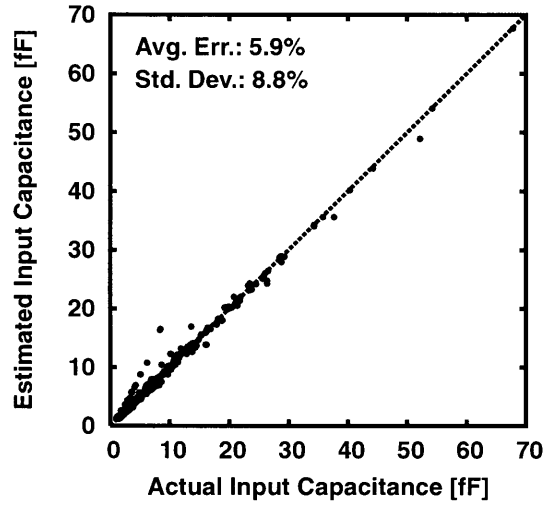


Figure 2.11 Comparison of actual and estimated input capacitances.

2.5 Future Trends

In the previous section, it has been shown that the proposed technique successfully estimates the cell characteristics of a 0.13μm standard cell library. The impact of the layout parasitics are expected to be more dominant in the future technologies. In this section, the future trends are analyzed based on the latest ITRS prediction [Sem04].

As a starting point, we show the impact of each layout parasitic type using the same 0.13μm standard cell library used in the previous section. 5 different types of netlists of the cells are prepared:

(1) Folded transistors, wiring resistances and capacitances, and diffusion capacitances

(2) Folded transistors, wiring capacitances, and diffusion capacitances

(3) Folded transistors and diffusion capacitances

(4) Folded transistors

(5) No layout parasitics

Then, we calculated the timing error of each type of netlists against the timing of (1), as shown in Figure 2.12.

Now we begin analyzing the future trends of the impacts. As a circuit for the analysis, we use a circuit consisting of two consecutive inverters illustrated in Figure 2.13 (a). To analyze the impacts of intra-cell layout parasitics, the circuit is modeled as an RC network shown in

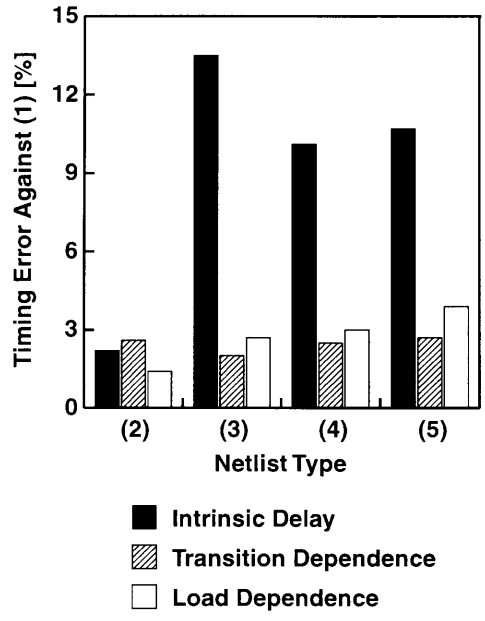


Figure 2.12 Impacts of intra-cell parasitics on a 0.13μm industrial standard cell library.

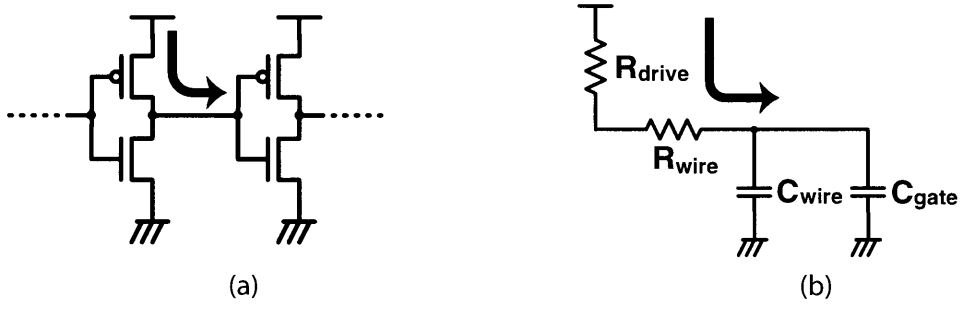


Figure 2.13 A circuit model for the analysis of the impact of layout parasitics. (a) two consecutive inverters and (b) a corresponding RC-level circuit model.

Figure 2.13 (b). In the figure, R_{drive} is the resistance of the driving inverter, R_{wire} and C_{wire} are the resistance and capacitance of the wire connecting the two inverters, and C_{gate} is the gate capacitance of the driven inverter. The delay of the inverter at the input side is given as follows:

$$t_{inv} = (R_{drive} + R_{wire})(C_{wire} + C_{gate}) \tag{2.9}$$

The impacts are calculated as follows:

$$I_{R_{wire}} = \frac{R_{wire}}{R_{drive} + R_{wire}} \tag{2.10}$$

$$I_{C_{wire}} = \frac{C_{wire}}{C_{wire} + C_{gate}} \tag{2.11}$$

$$I_{R_{wire}C_{wire}} = 1 - \frac{R_{drive}C_{gate}}{(R_{drive} + R_{wire})(C_{wire} + C_{gate})} \tag{2.12}$$

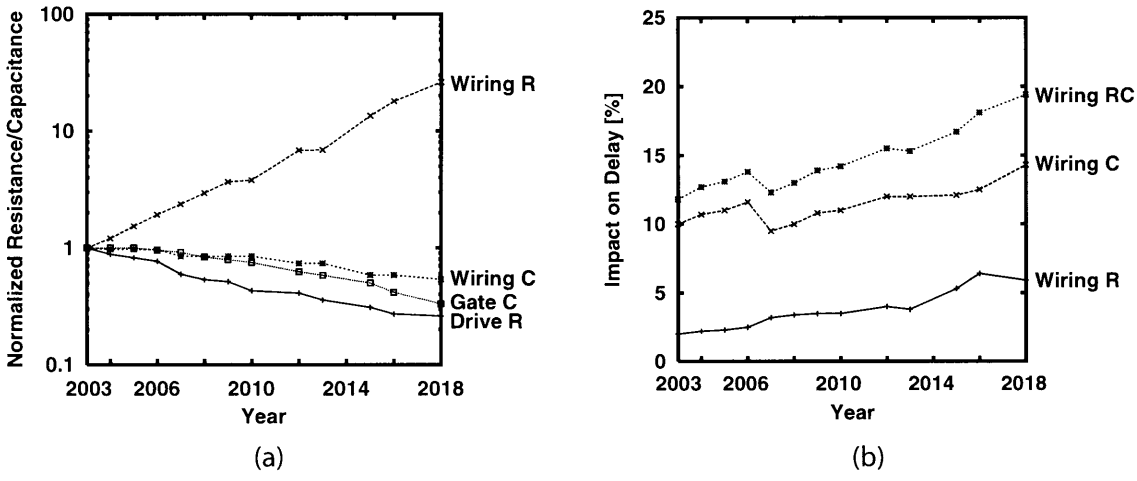


Figure 2.14 (a) Technology parameters based on ITRS and (b) impacts of intra-cell parasitics on cell delay.

where $I_{R_{wire}}$ is the impact of wiring resistance, $I_{C_{wire}}$ is the impact of wiring capacitance, and $I_{R_{wire}C_{wire}}$ is the impact of both wiring resistance and capacitance. Figure 2.14(a) shows the normalized technology parameters based on ITRS prediction [Sem04] and Figure 2.14(b) plots the predicted impacts calculated by the above formulas. The wiring resistance and capacitance, *i.e.* R_W and C_W, for the year 2003 are determined such that their impacts are 10% and 2% respectively. These values, 10% and 2%, are approximated from the impacts in 0.13μm technology (Figure 2.12). The graph shows that the impacts will keep increasing as the technology proceeds.

2.6 Conclusions

This chapter addressed an important issue of transistor-level optimization in the deep-submicron technologies where the impact of intra-cell layout parasitics cannot be neglected anymore. As a solution to this issue, we proposed a novel approach based on a cell characteristics estimation. To realize this approach in a feasible way, we also proposed a technique that estimates cell characteristics accurately in a short time using quick transistor placement. The experimental results on a 0.13μm industrial standard cell library demonstrated that the proposed technique estimated the cell characteristics with a reasonable accuracy in a negligibly small amount of time. We also showed that the impact of intra-cell layout parasitics will become more dominant in the future technologies. We hope that the proposed technique successfully solves the issue which is expected to be more crucial in the future.

Chapter 3

Logic Type Selection for Design-Specific Cell Libraries

3.1 Introduction

Cell library design at the transistor level is divided into two phases: *logic type selection* and *drive strength selection*. In the logic type selection phase, Boolean functions and transistor-level topologies are selected. The drive strength selection phase determines how many drive strengths are provided for each logic type and also optimizes the transistor sizes of each drive strength. In an ideal circuit, each gate is optimized at the transistor level with respect to its topology and transistor sizes. However, due to the limitation on the number of cells in a library, a sufficient number of logic types and drive strengths cannot be included. Only in a custom design methodology, this ideal can be realized. Therefore, it is desirable to find the minimal set of logic types and drive strengths specific to each design. This chapter focuses on the logic type selection for design-specific cell libraries. The design-specific drive strength selection problem will be studied in Chapter 4.

There are several studies available on the logic type selection for standard cell libraries. In [KKL87], they demonstrated that using a library with a rich set of logic types can reduce area considerably. [SK94] provided two principles (guidelines) for selecting cell sizes and logic types. One is to provide multiple drive strength for each logic type and the other is to provide both polarities for each logic type. They demonstrated that simple modifications following these principles in a cell library could lead to 20-30% improvement in final circuit speed. [GS96] performed an experiment-based study on how circuit area and performance depends upon the number of stack height where stack height is defined as the number of series-connected transistors in pullup/pulldown network in static CMOS gate. From the study, they demonstrated that using cells with bigger stack heights of up to 7 could improve circuit area by 30% while it does not significantly affect circuit performance. These studies are primarily in-

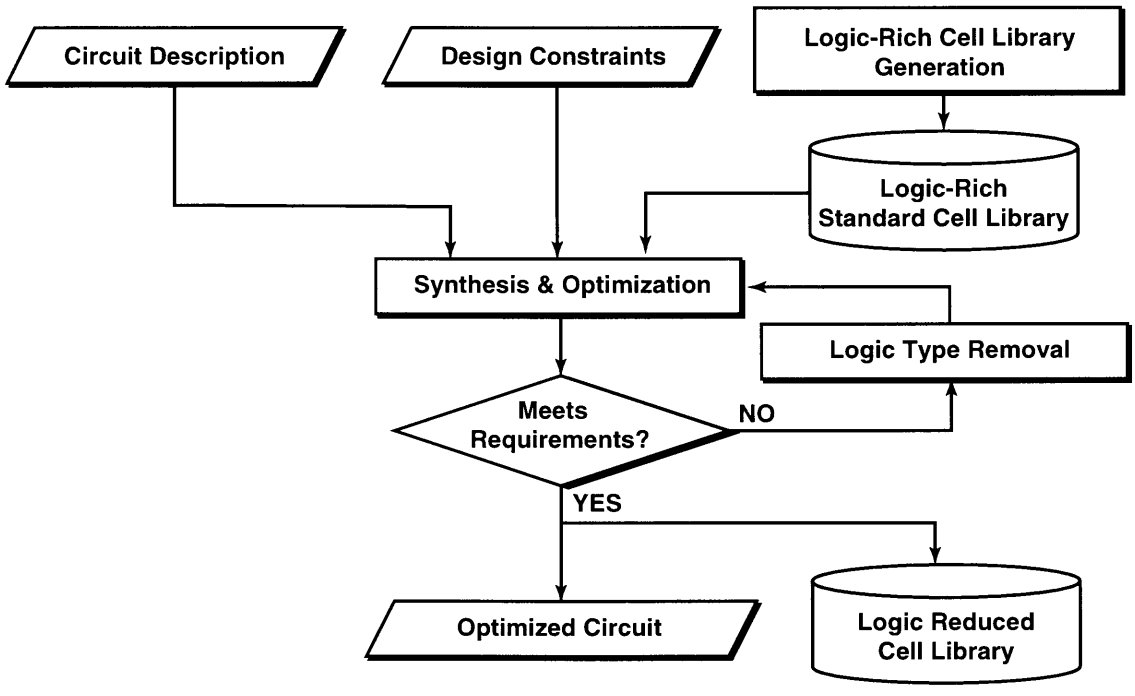
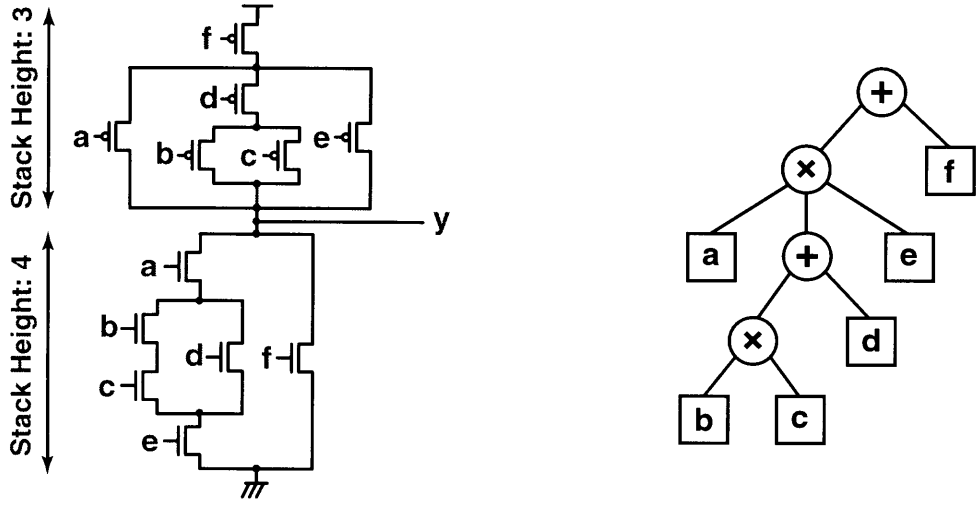


Figure 3.1 Overall flow for cell logic type selection.

tended to provide guidelines for designing general-purpose cell libraries, but not to construct the design-specific cell libraries.

This chapter presents a methodology for logic type selection for design-specific cell libraries. The proposed methodology can be used as the first step in the flow for generating design-specific cell libraries illustrated in Figure 1.2. Unlike the existing studies described above, the objective of the proposed methodology is to find the minimal set of logic types specific to a design. The overall flow is illustrated in Figure 3.1. The flow is divided into two steps: (a) preparing a cell library with a rich variation of logic types and (b) finding a minimal subset of the logic types subject to circuit performance constraints such as area, delay and power. Using the logic-rich library, an initial circuit is synthesized. Since the circuit may include a large number of logic types, the logic types are reduced by iteratively removing logic types of less importance and resynthesizing the circuit using the reduced cell library.

The rest of the chapter is organized as follows. Section 3.2 describes how to construct a logic-rich cell library which is used in our methodology. Section 3.3 presents the proposed logic type count minimization procedure. The method minimizes the logic type count iteratively under performance constraints. Section 3.4 presents the experimental results on a benchmark suite to demonstrate its effectiveness. Conclusions are drawn in Section 3.5.



(a) (b)

Figure 3.2 (a) A static CMOS compound gate where the number of inputs I is 6, the stack height of P-type transistors S_P is 3 and the stack height of N-type transistors S_N is 4 and (b) its corresponding AND/OR tree. The logic expression for the gate is $\overline{a \cdot (b \cdot c + d) \cdot e + f}$.

3.2 Constructing a Logic-Rich Cell Library

As explained in the previous section, transistor-level design of a cell library includes two important decision steps: the selection of transistor-level topologies and transistor sizes. This section explains how to construct a cell library with a rich variation of logic types.

In our cell library, every cell consists of a single-stage static CMOS compound gate under the assumption that multiple-stage static CMOS gates can be implemented as a combination of single-stage gates. A *static CMOS compound gate* is a channel connected component (CCC), which is a set of transistors connected at the sources and drains. It consists of two parts, a part of P-type transistors and a part of N-type transistors, which are structurally complementary. By regarding transistors as switches, it can be viewed as a series-parallel (*or* parallel-series) nested switch network which implements a Boolean function. An example of a static CMOS compound gate is shown in Figure 3.2 (a). Note that a static CMOS compound gate always implements a negative unate (*i.e.* monotonically decreasing) Boolean function.

3.2.1 Enumerating Transistor-level Topologies

Static CMOS compound gates can be categorized by the following characteristics: the number of inputs I, the stack height of P-type transistors S_P and the stack height of N-type transistors S_N where the stack height is defined as follows.

Definition 3.1. *The stack height of P(N)-type transistors is defined as the maximum number of P(N)-type series-connected transistors in a static CMOS compound gate.*

Thus, the set of transistor-level topologies for a logic-rich cell library can be specified by the following structural constraints:

- Maximum number of inputs: I_{max}
- Maximum stack height of P-type transistors: S_{Pmax}
- Maximum stack height of N-type transistors: S_{Nmax}.

Under these structural constraints, all possible transistor-level topologies are enumerated. First, an AND/OR tree representation is introduced as the representation of the transistor-level topology of a static CMOS compound gate. Then, we propose a procedure for enumerating AND/OR trees under these structural constraints.

AND/OR Tree Representation

To enumerate transistor-level topologies for cells, we use an AND/OR tree as the structural representation of a static CMOS compound gate. An AND/OR tree is defined as follows.

Definition 3.2. *An AND/OR tree is a rooted ordered tree where every leaf node corresponds to an input and every internal node represents either a Boolean AND operator or OR operator.*

An AND/OR tree can be mapped into a static CMOS compound gate in a unique way by transforming the tree into the series-parallel nested network, and *vice versa*. For a P-type transistor network, an OR node is transformed into the series connection of the subnetworks and an AND node is transformed into the parallel connection of the subnetworks. Similarly, for an N-type transistor network, an AND node is transformed into the series connection of the subnetworks and an OR node is transformed into the parallel connection of the subnetworks. Also, an AND/OR tree can be transformed into a logic expression by traversing the tree in depth-first order. For instance, the logic expression for the AND/OR tree in Figure 3.2 is $\overline{a \cdot (b \cdot c + d) \cdot e + f}$. The complement of the logic expression is the function represented by the corresponding static CMOS compound gate. The performance of a gate depends on how far a transistor is placed from the output, *i.e.* the order of series-connected transistors. Hence we distinguish AND/OR trees with different orders of children as different AND/OR trees. Figure 3.3 shows a static CMOS compound gate and its corresponding AND/OR tree such

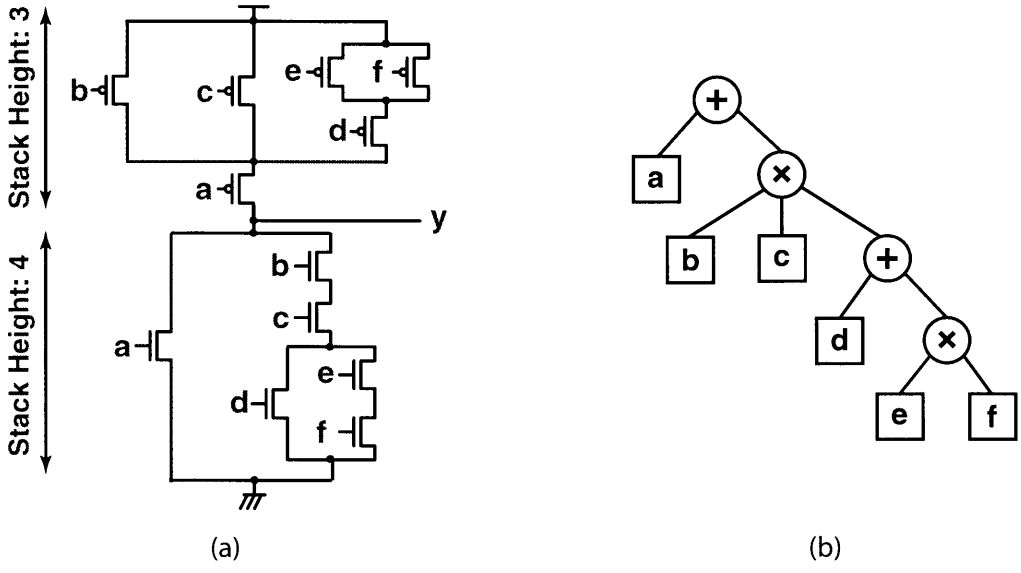


Figure 3.3 (a) A static CMOS compound gate and (b) its corresponding AND/OR tree such that its structure is equivalent to that of Figure 3.2 but the order of children is different. The logic expression for the gate is $\overline{a + b \cdot c \cdot (d + e \cdot f)}$.

that its structure is equivalent to that of Figure 3.2 but the order of children is different. Note that leaf nodes are not distinguished since the inputs of a cell are indistinctive. The AND or OR width of a tree is defined recursively as follows.

Definition 3.3. *The AND width of a node n in an AND/OR tree is defined as*

$$W_{AND}(n) = \begin{cases} 1 & \text{if } n \text{ is a leaf node} \\ \sum_{c \in Children(n)} W_{AND}(c) & \text{if } n \text{ is an AND node} \\ max\{W_{AND}(c) \mid c \in Children(n)\} & \text{if } n \text{ is an OR node} \end{cases} \tag{3.1}$$

where Children(n) is the set of the children of n. Similarly, the OR width of a node n in an AND/OR tree is defined as

$$W_{OR}(n) = \begin{cases} 1 & \text{if } n \text{ is a leaf node} \\ \sum_{c \in Children(n)} W_{OR}(c) & \text{if } n \text{ is an OR node} \\ max\{W_{OR}(c) \mid c \in Children(n)\} & \text{if } n \text{ is an AND node.} \end{cases} \tag{3.2}$$

Definition 3.4. *The AND(OR) width W(t) of an AND/OR tree t is defined as the AND(OR) width of the root node.*

The AND(OR) width of an AND/OR tree t is equivalent to the stack height of N(P)-type transistors, *i.e.*, $S_P \equiv W_{OR}(t)$ and $S_N \equiv W_{AND}(t)$.

Table 3.1 Number of AND/OR trees.

Maximum #leaf nodes	*Maximum AND width*	*Maximum OR width*	*#trees*
1	1	1	1
2	2	2	3
3	3	3	9
4	4	4	31
5	5	5	121
6	4	4	461
6	5	5	513
7	4	4	1631
7	5	5	2245
7	7	7	2322

Enumerating AND/OR Trees

Given structural constraints I_{max}, S_{Pmax} and S_{Nmax}, all possible AND/OR trees satisfying the structural constraints are enumerated. For AND/OR trees, I_{max} is the maximum number of leafs, S_{Pmax} is the maximum OR width and S_{Nmax} is the maximum AND width. The procedure for enumerating AND/OR trees is given in Figure 3.4. Starting from the set of AND/OR trees, each of which consists only of a leaf node, it creates a new AND/OR tree by AND-ing or OR-ing the subset of the AND/OR trees. If the new AND/OR tree satisfies the structural constraints, the tree is added to the set. The procedure is repeated until no new tree can be added. Table 3.1 shows the numbers of AND/OR trees under different constraints. Finally, the set of transistor-level topologies is obtained by transforming each AND/OR tree in the final set into the static CMOS compound gate.

3.2.2 Transistor Size Selection

As will be demonstrated in Chapter 4, transistor sizes in a cell have a big impact on the circuit performance. Since we use discrete-sized cells in our library, the transistor sizes should be selected carefully. In our cell library, every transistor in either P-type or N-type transistor network in a cell has the same size. Given a transistor-level topology of a cell, the transistor sizes of P-type and N-type networks are selected under the following constraints.

AND/OR Tree Enumeration

Input: $n_{leaf} \equiv$ Maximum number of leaf nodes ($\equiv I_{max}$)
$n_{AND} \equiv$ Maximum AND width ($\equiv S_{Nmax}$)
$n_{OR} \equiv$ Maximum OR width ($\equiv S_{Pmax}$)

Output: $T \equiv$ Set of AND/OR trees

Variables: $T_{AND} \equiv$ Set of AND/OR trees whose root is an AND node
$T_{OR} \equiv$ Set of AND/OR trees whose root is an OR node

1: $T_{AND} \leftarrow \{\}$
2: $T_{OR} \leftarrow \{\}$
3: **for** $i = 1$ to $i \leq n_{leaf}$ **do**
4: $t \leftarrow$ an AND/OR tree consisting only of the i-th leaf node
5: $T_{AND} \leftarrow T_{AND} \cup \{t\}$
6: $T_{OR} \leftarrow T_{OR} \cup \{t\}$
7: **end for**
8: **repeat**
9: **for all** subset $\{t_1, \ldots, t_n\} \subseteq T_{OR}$ **do**
10: $t \leftarrow$ an AND/OR tree such that the root node is an AND node and its children are $t_1, \ldots, t_n$.
11: **if** t satisfies the constraints of n_{leaf}, n_{AND} and n_{OR} **then**
12: $T_{AND} \leftarrow T_{AND} \cup \{t\}$
13: **end if**
14: **end for**
15: **for all** subset $\{t_1, \ldots, t_n\} \subseteq T_{AND}$ **do**
16: $t \leftarrow$ an AND/OR tree such that the root node is an OR node and its children are $t_1, \ldots, t_n$.
17: **if** t satisfies the constraints of n_{leaf}, n_{AND} and n_{OR} **then**
18: $T_{OR} \leftarrow T_{OR} \cup \{t\}$
19: **end if**
20: **end for**
21: **until** no new tree has been added in the last iteration
22: **return** $T_{AND} \cup T_{OR}$

Figure 3.4 AND/OR tree enumeration procedure.

- Set of drive strengths for each logic type: $D = \{d_1, \ldots, d_n\}$
- P-type and N-type transistor widths in the smallest inverter: w_{PINV} and w_{NINV}
- Beta ratio factor: k_β

The beta ratio for a cell is calculated as follows.

$$\beta = (w_{PINV}/w_{NINV}) + k_\beta(S_P - S_N) \tag{3.3}$$

Then, the widths of P-type and N-type transistors, w_P and w_N, are calculated as follows.

$$\begin{aligned} w_P = w_N \cdot \beta,\ w_N = w_{NINV} \quad &\text{if } \beta \geq w_{PINV}/w_{NINV} \\ w_P = w_{PINV},\ w_N = w_P/\beta \quad &\text{if } \beta < w_{PINV}/w_{NINV} \end{aligned} \tag{3.4}$$

Finally, for each drive strength $d_i \in D$, the widths of P-type and N-type transistors, w_{Pi} and w_{Ni}, are calculated as

$$w_{Pi} = d_i \cdot w_P \tag{3.5}$$

$$w_{Ni} = d_i \cdot w_N. \tag{3.6}$$

Note that the set of drive strengths may be different from a logic type to another.

3.3 Logic Type Count Minimization

The objective of the problem addressed in this section is to minimize the number of logic types required to implement a circuit under performance constraints such as area, delay and power. The proposed minimization procedure is based on hill-climbing heuristic [KV02]. Using the logic-rich library, an initial circuit is synthesized. Since the circuit may include a large number of logic types, the logic types are reduced by iteratively removing logic types of less importance and resynthesizing the circuit using the reduced cell library. The procedure for minimizing the logic type count is presented in Figure 3.5.

The quality of this heuristic depends heavily upon the selection of a logic type to be removed. In our procedure, the logic type with the largest slack is selected as a candidate for removal where the slack of the logic type is defined as follows.

Definition 3.5. *The **slack of a pin** is defined as the difference between the required time and the arrival time at the pin. The **slack of a gate** is the worst (smallest) slack at the pins on the gate. The **slack of a logic type** is the worst slack of the gates of the logic type. The **total slack of a logic type** is the sum of the slacks of the gates of the logic type.*

Logic Type Count Minimization

Input: $D \equiv$ Circuit description
$L_{rich} \equiv$ Set of cells in a logic-rich cell library
$C \equiv$ Set of performance constraints

Output: $L_{min} \equiv$ Minimal set of cells
$S_{min} \equiv$ Circuit synthesized using the minimal set of cells

Variable: $l \equiv$ Logic type to be removed

1: $L \leftarrow L_{rich}$
2: $S \leftarrow$ the circuit synthesized from D using L under C
3: **repeat**
4: $L_{min} \leftarrow L$
5: $S_{min} \leftarrow S$
6: Sort L in descending order of slack as the first priority and total slack as the second
7: $l \leftarrow$ the logic type at the head of L excluding the inverter and 2-input NAND gate
8: $L \leftarrow L - \{l\}$
9: $S \leftarrow$ the circuit synthesized from D using L under C
10: **until** S does not meet C
11: **return** L_{min} and S_{min}

Figure 3.5 Logic type count minimization procedure.

If there are two or more logic types with the largest slack, the logic type with the largest total slack is selected. Thus, the logic type having the least impact on the circuit performance is removed from the cell library. Note that the inverter and 2-input NAND gate are essential for most synthesis tools and hence they are excluded from the candidates for removal.

3.4 Experimental Results

First, we constructed a logic-rich cell library under the maximum number of inputs of 6, the maximum number of stack height of 4 for each transistor type. Each logic type has 7 drive strengths: 1x, 2x, 4x, 6x, 8x, 12x and 16x, and the beta ratio factor is 0.1. The number of logic types is 461 and the total number of cells is 1844 cells. The cell characteristics in an industrial 90nm technology were obtained using the prelayout cell characteristic estimator proposed in Chapter 2 with HSPICE [HSP03].

Table 3.2 ISCAS 85 benchmark circuits.

Name	Description
C432	27-channel interrupt controller
C499/C1355	32-bit SEC circuit
C880	8-bit ALU
C1908	16-bit SEC/DED circuit
C2670	12-bit ALU and controller
C3540	8-bit ALU
C5315	9-bit ALU
C6288	16x16 multiplier
C7552	32-bit adder/comparator

Next, we applied the proposed flow to 10 circuits from the ISCAS 85 benchmark circuits [BF85]. Table 3.2 provides the descriptions of the circuits [HYH99]. Cadence PKS (Physically-Knowledgeable Synthesis) [PKS04] was used to synthesize circuits from the register-transfer-level descriptions of the benchmark circuits. We performed cell logic type selection on the benchmark circuits with three types of performance constraints. In the first experiment, we performed cell logic type selection under the constraint of the maximum area of 1% within the optimal area, *i.e.*, $(A_{area} * 1.01)$ where A_{area} is the area of the area-optimal circuit. In the second experiment, the maximum area is limited to $(A_{delay} * 1.1)$ and the maximum path delay is limited to $(D_{delay} * 1.1)$ where A_{delay} and D_{delay} are the area and the maximum path delay of the delay-optimal circuit. In the last experiment, the maximum path delay is limited to 1% within the optimal delay, *i.e.*, $(D_{delay} * 1.01)$. Table 3.3 shows the results of these three sets of cell logic type selection. Overall, the area-optimal circuits require more logic types than the delay-optimal circuits. This reconfirms the fact that complex cells are are beneficial for area reduction.

Table 3.4 (a) and (b) show the statistics of the cell logic types in the design-specific cell libraries for C1908. Table 3.4 (a) corresponds to the result under the constraint of the maximum area of 1% within the optimal area, and Table 3.4 (b) corresponds to the result under the constraint of the maximum delay of 1% within the optimal delay. The design-specific library for the area-optimal circuit includes complex cells with up to 5 inputs. In contrast, the design-specific library for the delay-optimal circuit includes simple cells. Besides, two different topologies, $\overline{(A + B) \cdot C}$ and $\overline{A \cdot (B + C)}$, of OAI21 are used. Since these two types have different input-to-output delays, they are used according to the timing criticality of gate inputs. In Figure 3.6, Figure 3.7 and Figure 3.8, we also present the logic type count vs. performance

Table 3.3 Cell logic type selection results on the ISCAS 85 benchmark circuits in an industrial 90nm technology.

Circuit	1% Within Optimal Area			10% Area Degradation From 10% Within Optimal Delay			1% Within Optimal Delay		
	Area [μm^2]	Delay [ns]	#Logic Types	Area [μm^2]	Delay [ns]	#Logic Types	Area [μm^2]	Delay [ns]	#Logic Types
C432	346.5	3.3944	17	824.7	2.0820	8	2333.2	1.2314	15
C499	1055.7	1.9823	9	1512.9	1.5102	4	4180.7	1.0629	10
C880	806.9	2.7445	17	1114.3	1.8459	13	2215.3	1.0635	10
C1355	1057.9	1.9857	9	1598.2	1.5317	6	4325.0	1.0955	3
C1908	1065.2	3.1340	15	1438.7	2.2943	13	4374.7	1.4069	7
C2670	1472.1	2.7224	19	1883.7	1.7954	8	4539.1	0.9453	6
C3540	2376.8	4.0307	40	3298.6	3.0546	10	8694.1	1.7066	22
C5315	3344.5	3.3346	29	4313.3	2.3756	11	9708.2	1.3639	14
C6288	6090.7	11.1553	11	14398.6	8.0644	8	36195.9	4.8214	9
C7552	4589.8	5.0769	31	5562.6	2.7164	10	17515.6	1.2932	7

Table 3.4 Statistics of logic types in the design-specific cell libraries for C1908.

(a) Cell logic types for the area-optimal circuit.

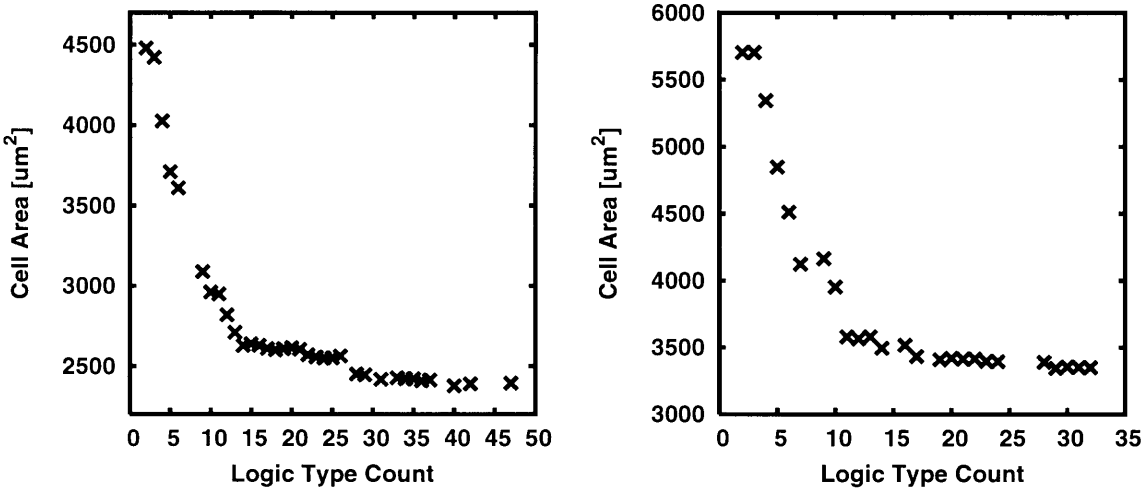
The number of cells is 15.

Cell Function	#Instances
$\overline{A \cdot B}$	55
$\overline{A \cdot (B + C)}$	45
$\overline{A + B}$	37
$\overline{A}$	34
$\overline{A \cdot B + C}$	24
$\overline{A \cdot B \cdot C}$	17
$\overline{A + B + C}$	12
$\overline{(A + B \cdot (C + D)) \cdot E}$	7
$\overline{A \cdot B \cdot C \cdot D}$	6
$\overline{A \cdot (B + C) + D}$	6
$\overline{A \cdot B + C + D}$	5
$\overline{A + B + C + D}$	5
$\overline{A \cdot B \cdot C + D}$	4
$\overline{(A + B + C) \cdot D}$	4
$\overline{A \cdot (B + C \cdot D) + E}$	2

(b) Cell logic types for the delay-optimal circuit.

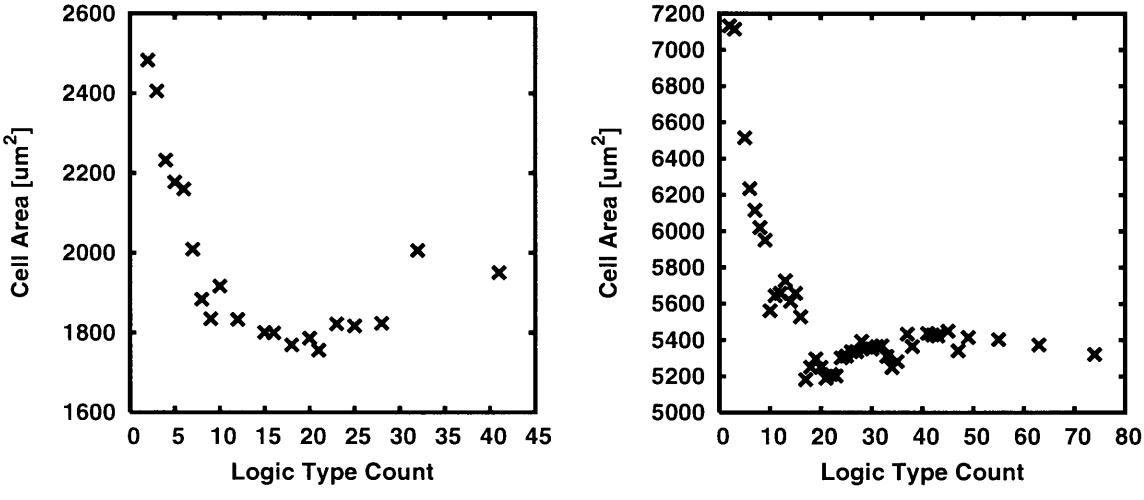
The number of cells is 7.

Cell Function	#Instances
$\overline{A}$	251
$\overline{A \cdot B}$	164
$\overline{A + B}$	70
$\overline{A \cdot (B + C)}$	41
$\overline{A + B \cdot C}$	31
$\overline{(A + B) \cdot C}$	26
$\overline{A \cdot B \cdot C}$	23



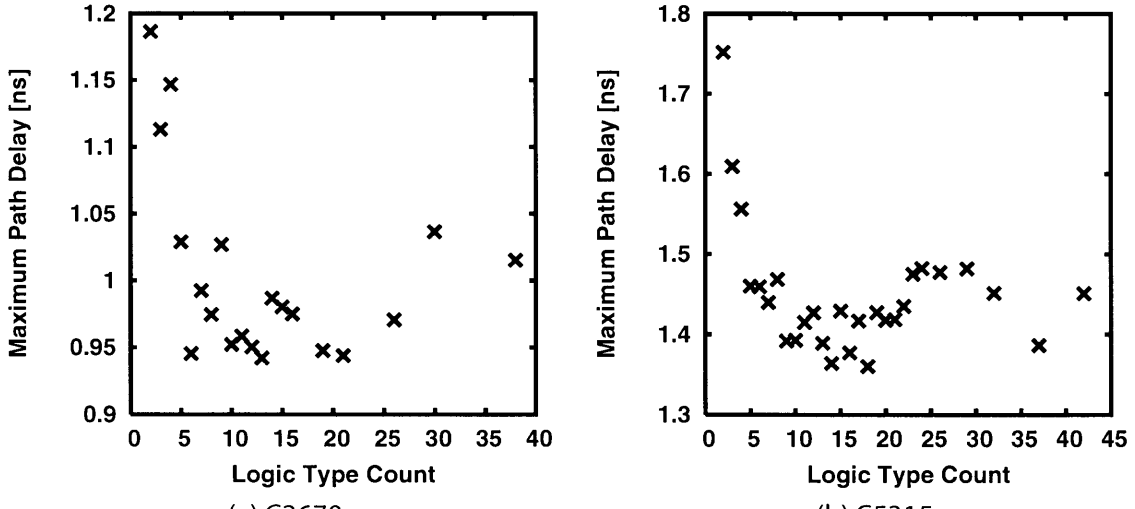
(a) C3540 (b) C5315

Figure 3.6 Logic type count vs. area tradeoff curves on area-optimal circuits.



(a) C1908 (b) C7552

Figure 3.7 Logic type count vs. area tradeoff curves on area-optimal circuits under the delay constraint of 10% within optimal delay.



(a) C2670 (b) C5315

Figure 3.8 Logic type count vs. delay tradeoff curves on delay-optimal circuits.

tradeoff curves obtained in this experiment. As can be seen from Figure 3.7 and Figure 3.8, the tradeoff curves are not smooth when the delay constraints exist. This is mainly due to the sub-optimality of delay optimization of the logic synthesis tool and to the discreteness of cell drive strengths.

3.5 Conclusions

This chapter addressed the cell logic type selection problem for design-specific cell libraries. Our methodology consists of two steps: the construction of a logic-rich library and the cell logic type count minimization. The proposed cell logic type count minimization method minimizes the logic type count iteratively under performance constraints. The experimental results on the ISCAS 85 benchmark suite in an industrial 90nm technology demonstrated that it is feasible to find the minimal set of logic types under performance constraints.

Chapter 4

Performance-Constrained Cell Count Minimization for Continuously-Sized Circuits

4.1 Introduction

As explained in Chapter 1, design optimization at the transistor level has been successfully used to achieve significant performance benefits above and beyond gate-level design optimization. In particular, continuous transistor sizing is known to have a significant impact on circuit performance and hence has been extensively studied. Although early work does not guarantee the optimality [FD85], Sapatnekar *et al.* first provided an exact sizing method based on an interior-point algorithm [SRVK93]. More recently, Chen *et al.* showed an elegant formulation of the sizing problem [CCW99] which can be optimally and efficiently solved by Lagrangian relaxation method [Ber00].

A continuously-sized circuit resulting from transistor sizing consists of gates with large variety of sizes. Figure 4.1 shows a cell size distribution of 2-input NOR gates after delay-optimal sizing in an ISCAS 85 benchmark circuit C499 implemented in an industrial 90nm technology. The cells are parameterized with two parameters: the drive strength and the beta ratio which is the ratio of N-type transistor width to P-type transistor width. In the figure, a circle indicates the number of instances of the cell is 1, a triangle indicates between 2 and 10, and a square indicates more than 10. In the standard cell based design flow where every gate is implemented by a cell, a large number of cells need to be prepared to implement a whole circuit. As the technology advances, the number of effects which need to be taken into account, *e.g.* performance variability and manufacturability, is increasing. Reflecting this situation, the design and characterization of cells are also becoming increasingly complex [BHSA03]. This drawback renders any continuous sizing solution in the standard cell based design flow to be impractical. Also, minimizing the cell count can directly improves the production throughput in the character projection based electron beam direct writing (CP-EBDW) method [Pfe79, IMK+00] in which

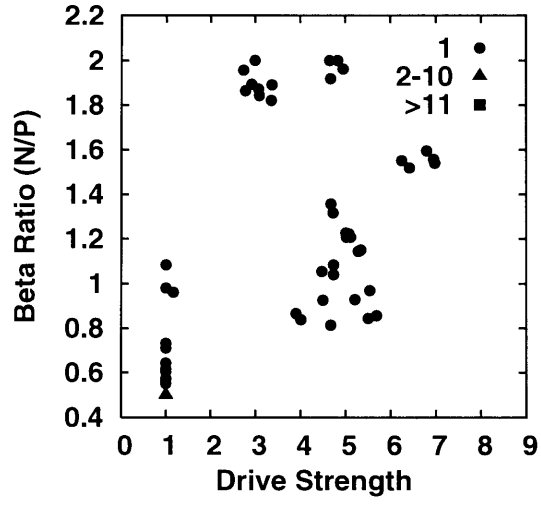


Figure 4.1 Cell size distribution of 2-input NOR gates after delay-optimal sizing in an ISCAS 85 benchmark circuit C499 implemented in an industrial 90nm technology. A circle indicates the number of instances of the cell is 1, a triangle indicates between 2 and 10, and a square indicates more than 10.

each gate is masklessly projected on a wafer at a time.

This chapter addresses a performance-constrained cell count minimization problem. Unlike the gate size selection problem [BKKS98, KP99] whose objective is to build a general-purpose cell library, the proposed method minimizes the number of cells of a circuit under performance constraints such as area, delay and power. In the flow for optimal generation of design-specific cell libraries illustrated in Figure 1.2, we first select a design-specific set of cell logic types by the methodology for cell logic type selection proposed in the previous chapter. Given the design-specific set of cell logic types, an optimal continuous sizing is performed. Then, a design-specific set of cell sizes for each logic type is obtained by the method proposed in this chapter. Thus, a design-specific cell library is obtained.

The rest of the chapter is organized as follows. Section 4.2 describes a posynomial cell model which we use to model cell characteristics, and provides a quick overview of a geometric programming based transistor sizing algorithm [CCW99]. In Section 4.3, we first formulate the performance-constrained cell count minimization problem formally, and then propose a hill-climbing heuristic for the problem. We also address several implementation issues towards a practical application of the proposed method to large-scale circuits. Section 4.4 presents the experimental results on a benchmark suite to demonstrate its effectiveness. Conclusions are drawn in Section 4.5.

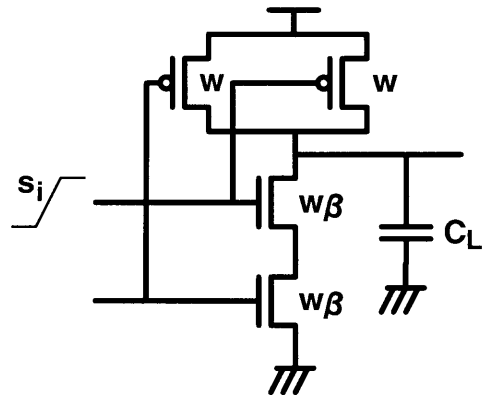


Figure 4.2 Our continuous cell model where s_i is the input slew and C_L is the output load capacitance. A cell consists of 2 parameters: P-type transistor width w and beta ratio β which is the ratio of N-type transistor width to P-type transistor width.

4.2 Preliminaries

4.2.1 Posynomial Cell Model

Our cell model is the posynomial cell model [FD85] which is most-commonly used by convex optimization based transistor sizing. Each cell is a *parameterized cell* where the sizes of the transistors in the cell are specified by a set of parameters $(p_1, \ldots, p_m)$, *e.g.* beta ratio and taper factor. Each parameter p_i has its lower bound p_i^L and upper bound p_i^U.

$$p_i^L \leq p_i \leq p_i^U \tag{4.1}$$

Figure 4.2 illustrates our continuous cell model. The model consists of 2 parameters: P-type transistor width w and beta ratio β which is the ratio of N-type transistor width to P-type transistor width. Note that other parameters such as taper factor can be incorporated to increase the degree of freedom and/or to improve the accuracy.

A cell is characterized with respect to the following characteristics: *timing*, *power*, *area* and *input capacitances*. A *timing* of a cell can be defined as a delay d or slew s of an arc of the cell for a given input slew s_i and an output load C_L.

$$d = f_d(p_1, \ldots, p_m, s_i, C_L) \tag{4.2}$$

$$s = f_s(p_1, \ldots, p_m, s_i, C_L) \tag{4.3}$$

Likewise, a cell power is typically modeled in the same way. Also, an area A and an input capacitance C_i of a cell are given as the functions of the parameters:

$$A = f_A(p_1, \ldots, p_m) \tag{4.4}$$

$$C_i = f_{C_i}(p_1, \ldots, p_m) \tag{4.5}$$

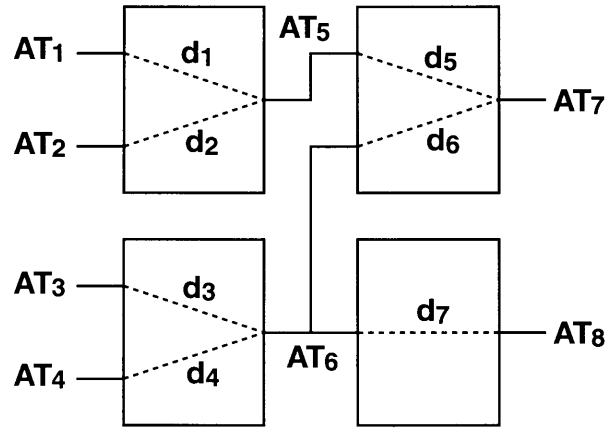


Figure 4.3 An example circuit model for continuous transistor sizing.

where C_i is the capacitance of i-th input.

A *posynomial* [Eck80] is a function g of a positive vector variable $t \in R^m$ having the form:

$$g(t) = \sum_{i=1}^{N} c_i t_1^{a_{i1}} t_2^{a_{i2}} \cdots t_m^{a_{im}} \tag{4.6}$$

where the exponents a_{ij} are arbitrary real numbers and the coefficients c_i are positive. An important property of a posynomial is that a posynomial is convex under a variable transformation [Eck80]:

$$t_j = e^{z_j} \quad j = 1, 2, ..., m. \tag{4.7}$$

For a convex function, any local minimum is also a global minimum.

The characteristics of a cell given by the equations 4.2–4.5 are modeled by posynomials. A posynomial for a cell characteristic can be obtained by fitting a number of data points which are obtained by a circuit simulation. There are several fitting techniques proposed for posynomials [RC05, RC06]. In addition, more accurate cell models based on posynomials have been proposed. [KKS00] proposed a generalized posynomial, and [TS05] presented piecewise convex cell model by dividing the function domain into small regions and fitting a function per region. In Section 4.4, we will demonstrate the accuracy of the posynomial cell model in an industrial 90nm technology.

4.2.2 Optimal Continuous Transistor Sizing

This section overviews an optimal continuous transistor sizing algorithm [CCW99] which we use in the proposed method. Figure 4.3 shows an example circuit model for continuous

transistor sizing. *For ease of explanation*, the following formulation does not take account of slews and load capacitances, nor does it distinguish rise and fall delays.

Definition 4.1. *A* ***gate*** g_i *is an instance of a cell* $c_{g_i} \in \{c_1, c_2, \ldots\}$ *with an associated set of parameters* $(p_{i1}, \ldots, p_{im})$

$$g_i = (c_{g_i}, p_{i1}, \ldots, p_{im}). \tag{4.8}$$

A circuit consists of a set of gates $G = \{g_1, \ldots, g_n\}$ and a set of wires $W = \{w_1, \ldots, w_o\}$. Each wire w_i has its associated arrival time AT_i and each input-to-output arc in a gate has its associated delay d. Then, area minimization problem under delay constraints can be formulated as follows:

$$\begin{aligned}
&\textbf{minimize } A(p) = \textstyle\sum_{i=1}^{n} A_i(p) \\
&\textbf{subject to} \\
&\quad p_j^L \le p_{ij} \le p_j^U \ (i = 1, \ldots, n, j = 1, \ldots, m) \\
&\quad AT_i \le AT_{max} \ (i = 1, \ldots, o) \\
&\quad AT_1 + d_1(p) \le AT_5, \ AT_2 + d_2(p) \le AT_5 \\
&\quad AT_3 + d_3(p) \le AT_6, \ AT_4 + d_4(p) \le AT_6 \\
&\quad AT_5 + d_5(p) \le AT_7, \ AT_6 + d_6(p) \le AT_7 \\
&\quad AT_6 + d_7(p) \le AT_8
\end{aligned} \tag{4.9}$$

where $p = (p_{i1}, p_{i2}, \ldots, p_{nm})$ is the set of all parameters, $A_i(p)$ is the area of g_i and AT_{max} is the maximum arrival time at any output. Similarly, delay minimization problem under an area constraint can be formulated as follows (throughout the remainder of this chapter , constraints for parameters and arrival times at internal wires are omitted for ease of explanation):

$$\begin{aligned}
&\textbf{minimize } AT_{worst} \\
&\textbf{subject to } A(p) \le A_{max}, \ AT_i \le AT_{worst} \ (i = 1, \ldots, o)
\end{aligned} \tag{4.10}$$

where A_{max} is the maximum area. Since the convexity is preserved under sums and maxima, a local optimum of these problems is the global optimum. Therefore, any nonlinear solver which finds a local minimum can find the global optimum solution. In [CCW99], Chen *et al.* showed that these constrained problems are efficiently and optimally solved by Lagrangian relaxation method.

4.3 Cell Count Minimization

4.3.1 Problem Formulation

Informally speaking, the objective of the problem addressed in this chapter is to minimize the number of cells required to implement a circuit under performance constraints such as area, delay and power. Note that only the cell parameters are subject to this optimization problem, *i.e.*, neither the topology of a circuit nor any cell logic type is changed. Before formulating the problem formally, we start with the following series of definitions.

Definition 4.2. *Two gates* $g_i = (c_{g_i}, p_{i1}, \ldots, p_{im})$ *and* $g_j = (c_{g_j}, p_{j1}, \ldots, p_{jm})$ *are said to be* ***equivalent*** *if and only if* $c_i = c_j$ *and* $p_{ik} = p_{jk}$ *for all* k*, and are denoted by* $g_i \sim g_j$.

For example, consider a circuit consisting of the following gates.

$$\begin{aligned} g_1 &= (c_1, 1, 1) \\ g_2 &= (c_1, 1, 2) \\ g_3 &= (c_2, 2, 3) \\ g_4 &= (c_2, 2, 3) \end{aligned}$$

g_1 and g_2 are not equivalent because the second parameters are different. Also, g_2 and g_3 are not equivalent because the cells are different. Since all parameters are same, g_3 and g_4 are equivalent.

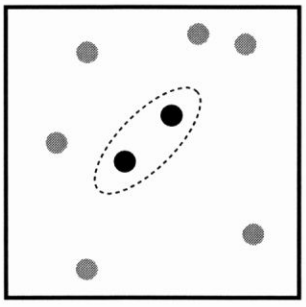
Definition 4.3. *A* ***gate group*** Γ *is defined as an equivalence class on the set of gates* G*, i.e.,* $g_i, g_j \in \Gamma \Longleftrightarrow g_i \sim g_j$.

Definition 4.4. *A* ***cell count*** $N(p)$ *is defined as* $|G/\sim|$*, the size of the quotient set of* G *(the number of all equivalence classes on* G*).*

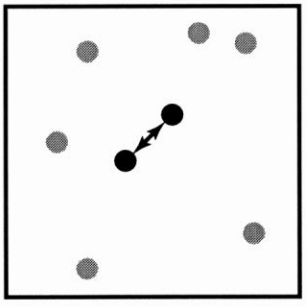
Definition 4.5. *Two gate groups* $\Gamma_i = (c_{\Gamma_i}, p_{i1}, \ldots, p_{im})$ *and* $\Gamma_j = (c_{\Gamma_j}, p_{j1}, \ldots, p_{jm})$ *are said to be* ***compatible*** *if and only if* $c_{\Gamma_i} = c_{\Gamma_j}$.

$N(p)$ can also be viewed as the number of cells which are required to implement the circuit. In the previous example, there are three gate groups:

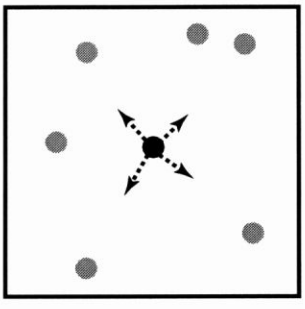
$$\begin{aligned} \Gamma_1 &= \{g_1\} \\ \Gamma_2 &= \{g_2\} \\ \Gamma_3 &= \{g_3, g_4\}. \end{aligned}$$



(a) Finding two compatible gate groups



(b) Merging them into a gate group



(c) Finding the optimal set of parameters of the gate group

Figure 4.4 An illustration of hill-climbing heuristic. Three steps (a)(b)(c) are repeatedly performed until no further change can be made.

Therefore, $N(p) = 3$ and hence three cells are required to implement the circuit. Also, Γ_1 and Γ_2 are compatible, and Γ_1 and Γ_3 are not.

Using these definitions, the problem addressed in this chapter is formulated as follows:

$$\begin{aligned} &\textbf{minimize } N(p) \\ &\textbf{subject to } A(p) \leq A_{max},\ AT_i \leq AT_{max}\ (i = 1, \ldots, o). \end{aligned} \tag{4.11}$$

Other performance constraints such as maximum power can be incorporated in a straightforward manner. Obviously, $N(p)$ is a non-smooth and non-convex function. Since the problem is intrinsically the combination of nonlinear and discrete problems, it is unlikely that the conventional nonlinear programming techniques solve this problem robustly. Also, the linear programming approach is inapplicable due to the nonlinearity of the constraint functions. Therefore, we propose an effective heuristic to solve this problem.

4.3.2 Hill-Climbing Heuristic

The proposed heuristic is based on hill-climbing method [KV02]. Starting from an optimally-sized circuit which satisfies the constraints, it reduces $N(p)$ by one at a time while satisfying the constraints, and it is repeated until no further change can be made. The basic

idea of reducing $N(p)$ by one is (a) finding two compatible gate groups Γ_i and Γ_j, (b) merging them into a gate group Γ_k, and (c) finding the optimal set of parameters of Γ_k, as shown in Figure 4.4.

The quality of this heuristic depends heavily upon the choice of two gate groups to be merged. Basically, merging two gate groups reduces the degrees of freedom of sizing and hence the resulting performance can also degrade. Therefore, two gate groups need to be chosen such that merging them has the least impact on the circuit performance. The proposed method uses the notions of slack and distance which are defined as follows.

Definition 4.6. *The **slack of a wire** is defined as the difference between the required time and the arrival time at the wire. The **slack of a gate** is the worst (smallest) slack of the wires connected to the gate. The **slack of a gate group** is the worst slack of the gates in the gate group.*

Definition 4.7. *The **distance** between two compatible gate groups $\Gamma_i = (c_{\Gamma_i}, p_{i1}, \ldots, p_{im})$ and $\Gamma_j = (c_{\Gamma_j}, p_{j1}, \ldots, p_{jm})$ is the Euclidean distance between two vectors of parameters:*

$$D(\Gamma_i, \Gamma_j) = \sqrt{\sum_{k=1}^{m} (K_i(p_{ik} - p_{jk}))^2} \tag{4.12}$$

where K_i is the weight factor for i-th parameter.

The weight factor K_i is determined based on the impact of the i-th parameter to the circuit performance. Thus, the distance between two gate groups can be viewed as an estimate of the impact on the circuit performance when the gate groups are merged. The basic criteria are to choose two compatible groups such that (a) the distance between the gate groups is small and (b) at least one of the slacks of the gate groups is large. To accurately analyze the slacks of gate groups, the proposed method performs total slack maximization under the given performance constraints:

$$\begin{aligned} &\textbf{maximize } S(p) \\ &\textbf{subject to} \\ &\quad A(p) \leq A_{max} \\ &\quad AT_i \leq AT_{max} \ (i = 1, \ldots, o) \\ &\quad p_{ik} = p_{jk} \ (k = 1, \ldots, m, \ g_i \in \Gamma_l, \ g_j \in \Gamma_l, \ \Gamma_l \in G/\sim) \end{aligned} \tag{4.13}$$

where $S(p)$ is the sum of the slacks of the wires. In the last constraint, the gates in each gate group are forced to have the same set of parameters. Thus, the cell count remains the same during the total slack maximization.

Cell Count Minimization

1: **repeat**
2: Maximize total slack under performance constraints
3: **for all** gate group Γ_i in descending order of slack **do**
4: $H \leftarrow$ k-nearest neighbor compatible gate groups of Γ_i
5: **for all** $\Gamma_j \in H$ in ascending order of the distance **do**
6: Merge two gate groups Γ_i and Γ_j
7: (Locally) maximize total slack under performance constraints
8: **if** constraints are satisfied **then**
9: **break**
10: **else**
11: Undo Step 6 and 7
12: **end if**
13: **end for**
14: **end for**
15: **until** no further change can be made

Figure 4.5 Cell count minimization procedure.

A pseudocode for the hill-climbing heuristic is presented in Figure 4.5. After the total slack maximization, all gate groups are sorted in descending order of slack and the gate group Γ_i with the largest slack is chosen. Next, the k-nearest neighbor compatible gate groups of Γ_i are computed where k is a user-defined parameter which controls the runtime, and the nearest gate group Γ_j is chosen. After merging the two gate groups into a gate group, the optimal parameter set is determined by maximizing the total slack under the performance constraints. For efficiency, this step can be performed locally, *i.e.*, only the merged gate group and its neighbor gate groups are optimized, as discussed in Section 4.3.3. If the constraints cannot be satisfied after this slack maximization, the gate group pair is discarded and the next nearest gate group of Γ_i is chosen as Γ_j. If there is no more k-nearest neighbor, the gate group with the next largest slack is chosen as Γ_i. The process is repeated until no more groups can be merged.

4.3.3 Implementation Issues

Total slack maximization (Step 2 and 7 in Figure 4.5) requires the computation of required times at all wires. Since the computation of required times includes subtraction and minimum operations, the formulation like (9) may not be possible. Instead, we use the sum of slacks at all endpoints (*i.e.* primary outputs) as total slack. Since required times at endpoints are all fixed, this problem is simply equivalent to the minimization of the sum of arrival times at all endpoints. After the total slack maximization, the slack of each internal wire can be computed for Step 3.

From our experiences, the local optimization (Step 7) dominates the overall runtime. In the current implementation, a local optimization is performed by a global optimization fixing the parameters other than the target gates. Even though the solution space is comparably small, the runtime is still large. In addition, in the case that the constraints cannot be met, typical nonlinear solvers do not give up until they reach the maximum iteration limit. A simple optimization, such as decent direction and conjugate direction methods, in conjunction with incremental timing analysis may speed up the local optimization dramatically.

For large-scale circuits, the nearest neighbor query (Step 5) may also be another bottleneck. A number of efficient nearest neighbor query algorithms are available [GG98], *e.g.*, region quadtree method [FB74].

4.4 Experimental Results

First, we constructed a continuously-sized cell library consisting of 24 logic types. The cells were characterized for the posynomial cell model described in Section 4.2.1 using an industrial 90nm technology as follows. For each logic type, P-type transistor widths were varied from 1μm to 8μm and beta ratios (the ratio of N-type transistor width to P-type transistor width) were varied from 0.5 to 2. Input slews were varied from 10ps to 1000ps, output loads were varied from 1fF to 100fF. Cell delays and slews were simulated using a prelayout cell characteristic estimator [YDB04] with HSPICE [HSP03] for 256 combinations of the parameters. Then, we fitted the data to a posynomial function and obtained the coefficients and exponents. Table 4.1 presents the average fitting error and the standard deviation of cell delays and slews of each logic type. Overall, the average fitting error was about 1.06% and the standard deviation was 1.19%. For cell areas, the average fitting and the standard deviation were both less than 0.01%. For input loads, the average fitting and the standard deviation were 0.23% and 0.19%, respectively.

Table 4.1 Fitting errors of posynomial gate delay models.

Logic Type	Average Error [%]	Standard Deviation [%]
INV	0.92	0.85
NAND2	1.55	2.12
NAND3	1.02	1.19
NAND4	0.75	0.82
NOR2	2.33	1.70
NOR3	1.55	1.88
NOR4	1.25	1.30
AOI21	1.56	2.25
AOI22	0.99	1.08
AOI211	1.17	1.33
AOI221	0.91	0.97
AOI31	1.11	1.53
AOI311	0.91	1.05
AOI32	0.79	0.85
AOI41	0.86	1.16
OAI21	1.23	1.41
OAI22	0.92	0.99
OAI211	0.82	0.90
OAI2111	0.62	0.65
OAI221	0.66	0.70
OAI31	1.02	1.13
OAI32	0.77	0.80
OAI311	0.65	0.71
OAI41	1.13	1.27
Overall	1.06	1.19

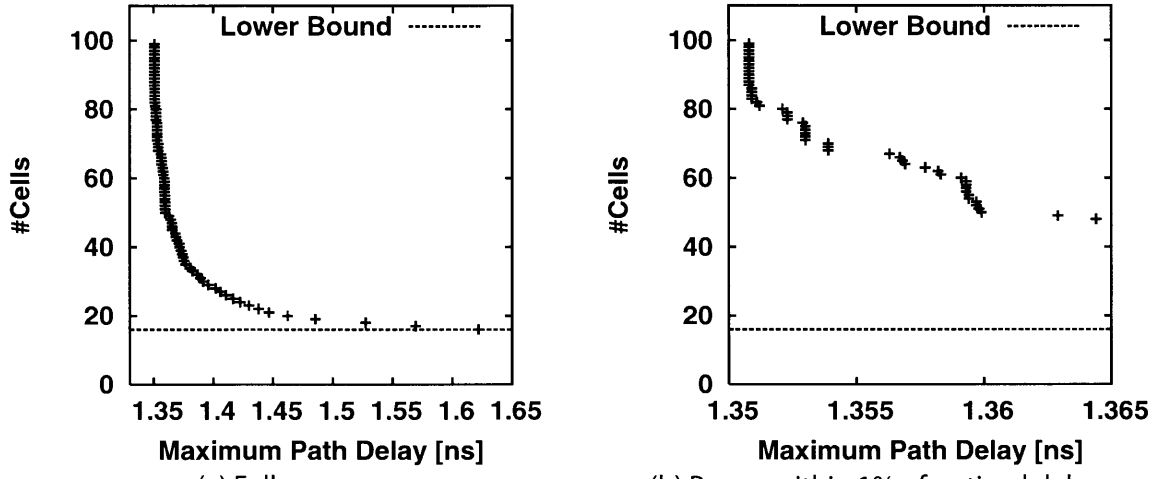
Next, we implemented the optimal continuous transistor sizing algorithm explained in Section 4.2.2 and the performance-constrained cell minimization algorithm proposed in Section 4.3.2. To solve the nonlinear problems, a state-of-the-art nonlinear optimizer IPOPT [WB06] is used. We then applied them to 10 circuits from the ISCAS 85 benchmark circuits [BF85]. The benchmark circuits were first synthesized for optimal delay using a discretely-sized cell library in the same 90nm technology. After replacing the cells with the continuously-sized cells, delay-optimal circuits were obtained by performing an unconstrained optimal-delay sizing followed by an optimal-area sizing under the optimal delay constraint. Then, we applied the proposed cell count minimization method to the delay-optimal circuits as follows. The cell count of each circuit is minimized with accepting 1% degradation of optimal delay and keeping the area, *i.e.*, under the constraints of the maximum path delay of $(D_{opt} * 1.01)$ and the maximum area of A_{opt} where D_{opt} and A_{opt} are the maximum path delay and the area of a delay-optimal circuit, respectively. Table 4.3 compares the cell counts of the delay-optimal circuits and the circuits after the cell count minimization. In the table, the second column shows the number of logic types used in the circuit. Note that the number of logic types is the lower bound on the cell count. The last column shows the cell count reduction rate calculated by $(N_{opt} - N_{1\%})/N_{opt} * 100$ where N_{opt} and $N_{1\%}$ are the cell counts of the delay-optimal circuit and the circuit after the cell count minimization, respectively. The results demonstrate that the cell counts could be reduced by 78.0% on average with accepting 1% degradation which is almost equivalent to the delay model error. Figure 4.6–Figure 4.11 present the tradeoff curves between the maximum path delay and the cell count. The curves were obtained by increasing the maximum path delay constraint from D_{opt} and keeping the area constraint the same. In the figures, (a) presents the curve in the full range from the optimal delay to the minimum cell count, and (b) presents the same curve in a range within 1% of the optimal delay. An important observation from these results is that the cell count can be reduced dramatically with accepting very little delay degradation. Figure 4.12 (a) and (b) show the cell size distributions of 2-input NOR gates in a circuit C499 after delay-optimal sizing and after cell count minimization with accepting 1% degradation of optimal delay, respectively. In the figures, a circle indicates the number of instances of the cell is 1, a triangle indicates between 2 and 10, and a square indicates more than 10. As can be seen from Table 4.3, the runtime of the current implementation is not small on large circuits. This is mainly due to the local total slack minimization step (Step 7 in Figure 4.5). By using the techniques in Section 4.3.3, the runtime can be improved dramatically and larger circuits should be optimized in a reasonably short runtime.

Table 4.2 Statistics of ISCAS 85 benchmark circuits implemented in an industrial 90nm technology.

Circuit	*#Used Logic Types*
C432	16
C499	9
C880	20
C1355	9
C1908	18
C2670	21
C3540	24
C5315	20
C6288	17
C7552	23

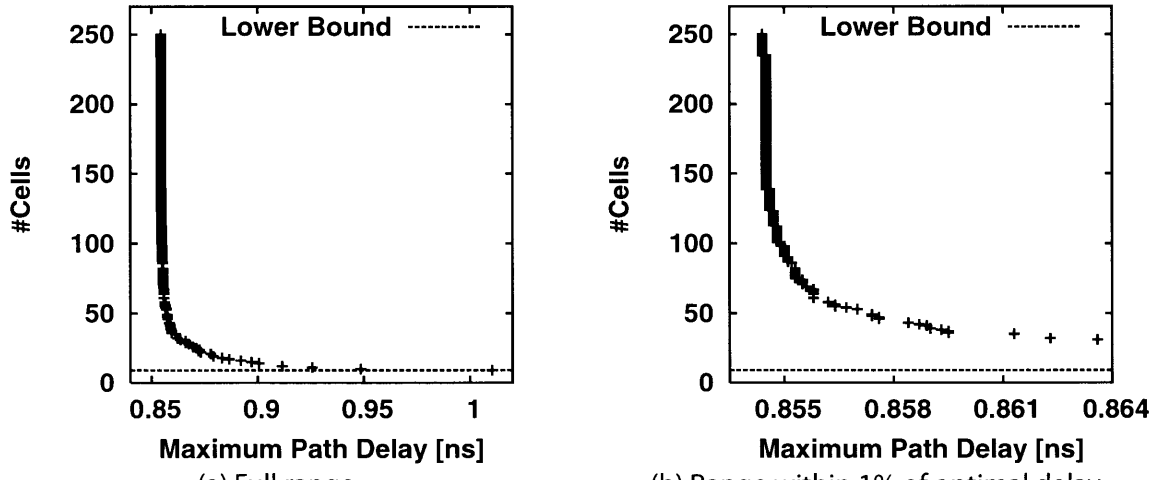
Table 4.3 Cell count minimization results on the ISCAS 85 benchmark circuits in an industrial 90nm technology.

Circuit	*Delay Optimal*			*Accepting 1% Degradation*				*Cell Count Reduction [%]*
	Area [μm^2]	*Delay [ns]*	*Cell Count*	*Area [μm^2]*	*Delay [ns]*	*Cell Count*	*CPU time [sec.]*	
C432	2955.4	1.3508	99	2955.4	1.3643	49	31.9	50.5
C499	7374.8	0.8545	250	7173.7	0.8632	32	71.6	87.2
C880	2884.2	1.1643	213	2884.2	1.1759	43	35.5	79.8
C1355	7343.3	0.8599	250	7343.3	0.8643	45	73.0	82.0
C1908	6145.6	1.2704	274	5813.0	1.2833	116	103.9	57.7
C2670	3712.8	1.0555	554	3708.9	1.0660	43	401.3	92.2
C3540	9512.5	1.7305	628	9512.5	1.7479	148	1203.2	76.4
C5315	8427.0	1.2830	941	8427.0	1.2959	153	1207.4	83.7
C6288	44636.1	4.8085	1587	44636.1	4.8567	265	9101.0	83.3
C7552	14268.6	1.4533	1341	14268.6	1.4678	172	12758.6	87.2
Average	—	—	—	—	—	—	—	78.0



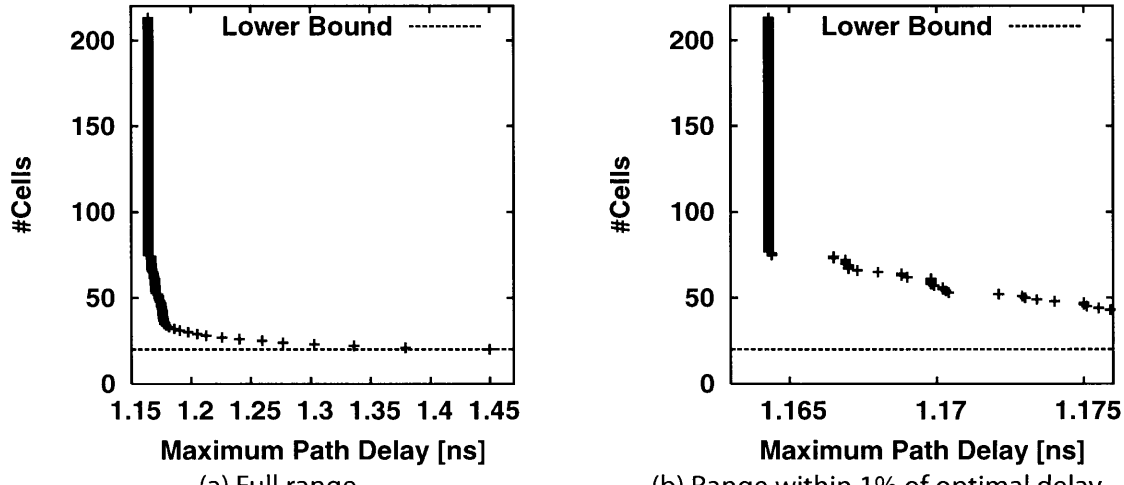
(a) Full range (b) Range within 1% of optimal delay

Figure 4.6 Delay vs. cell count tradeoff curve on C432.



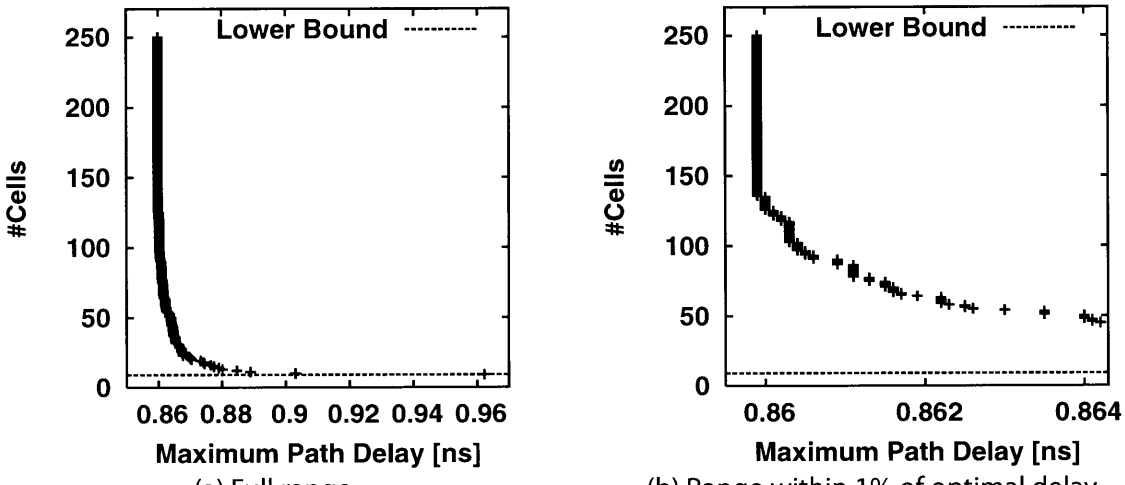
(a) Full range (b) Range within 1% of optimal delay

Figure 4.7 Delay vs. cell count tradeoff curve on C499.



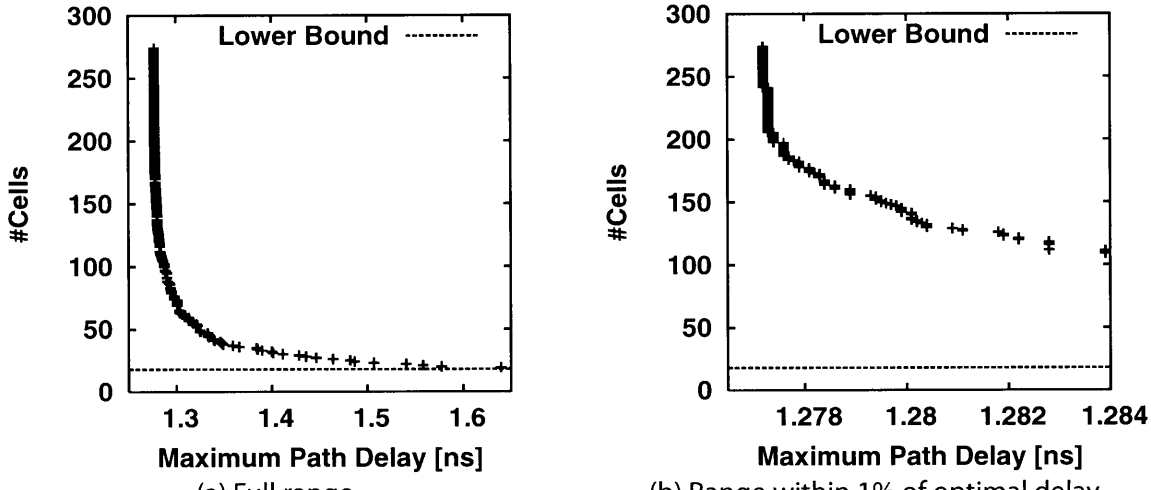
(a) Full range (b) Range within 1% of optimal delay

Figure 4.8 Delay vs. cell count tradeoff curve on C880.



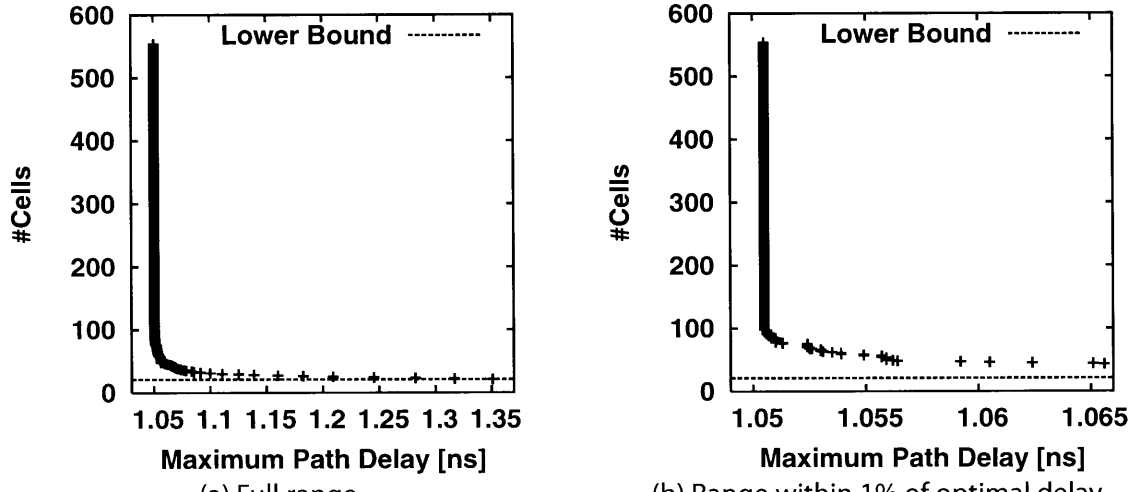
(a) Full range (b) Range within 1% of optimal delay

Figure 4.9 Delay vs. cell count tradeoff curve on C1355.



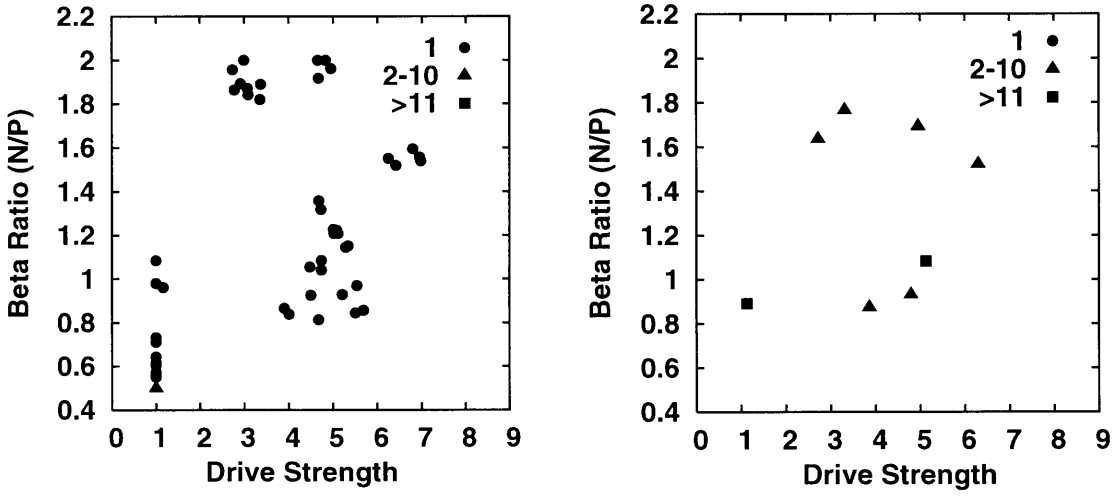
(a) Full range (b) Range within 1% of optimal delay

Figure 4.10 Delay vs. cell count tradeoff curve on C1908.



(a) Full range (b) Range within 1% of optimal delay

Figure 4.11 Delay vs. cell count tradeoff curve on C2670.



(a) After delay-optimal sizing

(b) After cell count minimization with accepting 1% degradation of optimal delay.

Figure 4.12 Cell size distributions of 2-input NOR gates in an ISCAS 85 benchmark circuit C499.

4.5 Conclusions

This chapter addressed a performance-constrained cell count minimization problem for continuously-sized circuits. After providing a formal formulation of the problem, we proposed an effective heuristic for the problem. The proposed hill-climbing heuristic iteratively minimizes the number of cells under performance constraints such as area, delay and power. The experimental results on a benchmark suite demonstrated its effectiveness. We also discussed several implementation issues towards a practical application of the proposed method to large-scale circuits. We expect that the solution presented in this chapter will be successfully used to achieve a practical realization of continuous sizing in the standard cell based design flow.

Chapter 5

Synthesis of Minimal Static CMOS Circuits

5.1 Introduction

In general, transistor-level optimization includes two decision problems: *transistor sizing* and *circuit topology synthesis*. It is well known that transistor sizing is one of the effective techniques for timing optimization. Recent semi-custom design methodologies for high-performance ASICs have employed cell libraries with a rich variation of drive strengths [NL01, RHH+02] to obtain a similar advantage of continuous transistor sizing. It is notable that their cell libraries has a small set of logic families. This fact implies that circuit topology optimization at the intra-cell level is not as effective as transistor sizing. In contrast, it is also known that the use of complex gates can reduce the design area. Although such complex gates are typically not available in cell libraries, circuit topology synthesis technique can generate such gates by combining several cells into a single cell [BB02]. Besides, the area reduction of non-critical regions improves the overall area utilization, which promotes a faster convergence of timing optimization. On this background, the following three chapters focus on area optimization at the transistor level. The methods proposed in the chapters can be used for the manual generation steps in the flow for generating design-specific cell libraries.

Area optimization is one of the most classical problems in the field of logic synthesis. Conventionally, an area optimization is achieved by reducing the literal count in a multi-level logic network. An early work [Law64] provided an exact multi-level logic minimization algorithm, however, the algorithm applies only to a factored form of a Boolean function, *i.e.*, a *single-output single-stage* static CMOS circuit. For synthesizing multi-output multi-stage static CMOS circuits, a number of heuristics have been developed [GGP+97, LA99]. There was also an attempt to synthesize an arbitrary static CMOS circuit by technology mapping with a rich set of library cells [DGR+87]. It requires tens of thousands of cells to be prepared in advance, which is too infeasible. Due to their heuristic nature, these methods don't guarantee any opti-

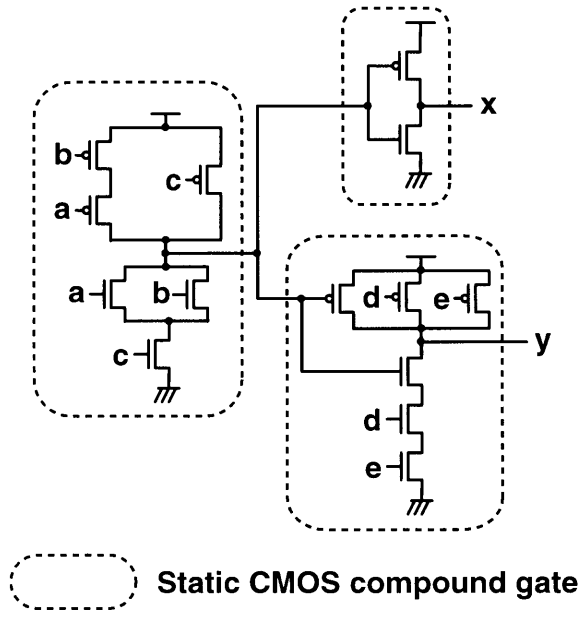
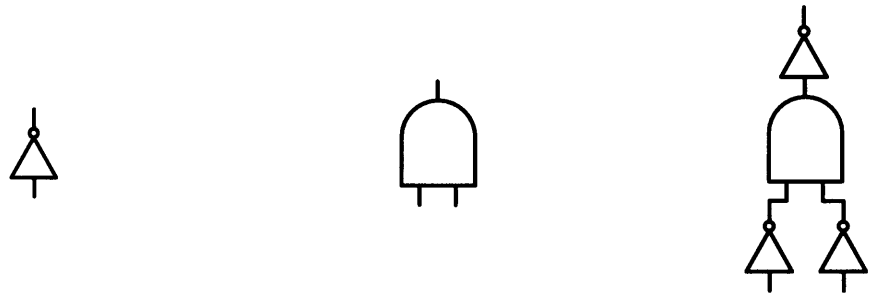


Figure 5.1 Static CMOS circuit (14 transistors)

mality of the solution. Apart from static CMOS circuits, there are a large number of literatures on the synthesis of more general transistor circuits such as non-series-parallel circuits [AM90] and pass-transistor circuits [YSS96, BNNSV97].

As mentioned above, transistor-level optimization targeting standard-cell based design flow is performed only at the intra-cell level. Hence, we believe that it is still worth developing a computationally-intensive algorithm even though it is applicable only to circuits as small as standard cells. This chapter presents an algorithm which synthesizes an arbitrary static CMOS circuits targeting the reduction of transistor counts. To make the problem tractable, the solution space is restricted to the circuit structures which can be obtained by performing algebraic transformations on an arbitrary prime-and-irredundant two-level circuit, as illustrated in Figure 1.3. The circuit structures are implicitly enumerated via structural transformations on a single graph structure, then a dynamic-programming based algorithm efficiently finds the minimum solution among them.

The rest of the chapter is organized as follows. In Section 5.2, we first show how static CMOS circuits are represented in our approach. After the problem formulation in Section 5.3, the proposed algorithm is described in Section 5.4 and Section 5.5. Section 5.6 presents experimental results on a benchmark suite targeting standard cell implementations and demonstrates the feasibility and effectiveness of the proposed approach. We also show the results of a numerical analysis on randomly-generated problems to demonstrate the efficiency of the proposed algorithm. Conclusions are drawn in Section 5.7.



(a) Inverter pattern (cost=2)

(b) AND pattern (cost=2)

(c) OR pattern (cost=2)

Figure 5.2 Primitive patterns in equivalent AND2/INV network

5.2 Representing a Static CMOS Circuit

A *static CMOS compound gate* is a channel connected component (CCC), which is a set of transistors connected at the sources and drains. It consists of two parts, a part of P-type transistors and a part of N-type transistors, which are structurally complementary. By regarding transistors as switches, it can be viewed as a series-parallel (*or* parallel-series) nested switch network which implements a Boolean function. Note that a static CMOS compound gate always implements a negative unate (*i.e.* monotonically decreasing) Boolean function. A *static CMOS circuit* is defined as a circuit of static CMOS compound gates. An example of a static CMOS circuit is shown in Figure 5.1.

A *Boolean network* is defined as a directed acyclic graph in which every node has an associated Boolean function. An *AND2/INV network* is a Boolean network in which the type of each node is limited to either a 2-input AND gate or an inverter. A *negative unate tree* is a subtree of an AND2/INV network with the following properties: 1) the root is an inverter, and 2) every path from the root to leaf has an odd number of inverters. In other words, a negative unate tree can be viewed as an AND/OR tree with an inverter at the root. A negative unate tree can be mapped into a static CMOS compound gate in a unique way by transforming the tree into the series-parallel nested network, and *vice versa*. An *equivalent AND2/INV network* is a network of disjoint negative unate trees.

The *cost* of an equivalent AND2/INV network is defined as the number of transistors in the corresponding static CMOS circuit. Figure 5.2 shows three primitive patterns which form a negative unate tree. Since each pattern has a cost of 2, the cost of a negative unate tree can be calculated as twice the number of patterns in the tree. Similarly, the cost of an equivalent AND2/INV network can be calculated as twice the number of patterns in the network. Fig-

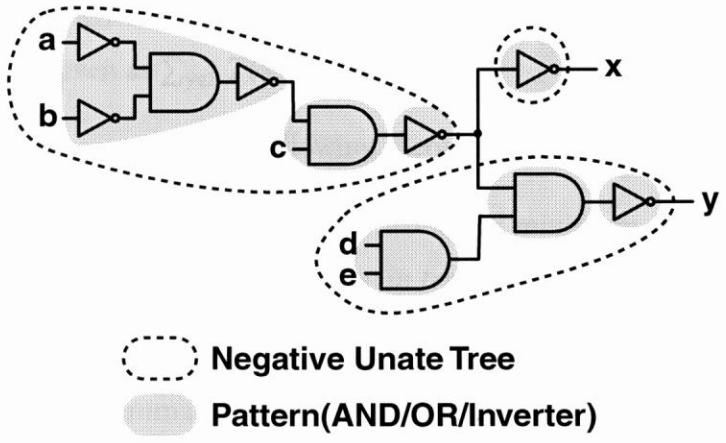


Figure 5.3 Equivalent AND2/INV network corresponding to static CMOS circuit in Figure 5.1. There are 7 patterns in the network and hence the cost is 14.

ure 5.3 shows the equivalent AND2/INV network corresponding to the static CMOS circuit in Figure 5.1. The cost of the equivalent AND2/INV network is 14 since there are 7 patterns. As can be seen, this matches the number of the transistors in Figure 5.1. Note that the notion of the cost of primitive cells is conceptual and it does not imply that an AND or OR gate can be implemented with 2 transistors.

The following lemmas show the relationship between the number of transistors and the cost.

Lemma 5.1. *For an arbitrary static CMOS compound gate γ with n transistors, there exists a negative unate tree with a cost of n.*

Proof. Let the function represented by γ be f. Since each input of γ is connected to a P-type transistor and an N-type transistor, the number of the inputs of γ is m where $2m = n$. By transforming the pull-up (pull-down) network of γ into a tree, an AND/OR tree with m leaves such that the tree represents the complement of f is obtained. By decomposing AND and OR nodes into 2-input nodes, we can obtain a binary AND/OR tree. The number of the nodes in a binary tree with m leaves is given as $m - 1$. By adding an inverter node at the root, the tree represents f and can be viewed as a negative unate tree by regarding the AND, OR and inverter nodes as primitive patterns. Since the number of the nodes in the tree is m, the cost is $n = 2m$ from the definition. ■

Lemma 5.2. *For an arbitrary static CMOS circuit χ with n transistors, there exists an equivalent AND2/INV network with a cost of n.*

Proof. Let Γ be the set of the static CMOS compound gates contained in χ and $T(\gamma)$ be the number of transistors in a static CMOS compound gate $\gamma \in \Gamma$. Then, $\sum_{\gamma \in \Gamma} T(\gamma) = n$. For each

$\gamma \in \Gamma$, there exists a negative unate tree with a cost of $T(\gamma)$ from Lemma 5.1. Therefore, the cost of the network is given as $\sum_{\gamma \in \Gamma} T(\gamma) = n$. ■

From these lemmas, we can prove the following theorem.

Theorem 5.1. *If an equivalent AND2/INV network ν is minimum in terms of the cost, the corresponding static CMOS circuit χ is minimum in terms of the number of transistors.*

Proof. The proof is by contradiction. Let the cost of ν be n. By performing the inverse transformation in the proof of Lemma 5.1, the corresponding static CMOS compound gate with n transistors is obtained from ν. Assume that χ is not minimum, *i.e.*, there exists a static CMOS circuit χ' with m transistors such that $m < n$. From Lemma 5.2, there exists an equivalent AND2/INV network ν' with a cost of m which represents χ'. However, this contradicts the definition that ν is minimum. ■

5.3 Problem Formulation

The problem addressed in this chapter can be formulated as follows:

Problem 5.1. *Given a set of Boolean functions, find a static CMOS circuit which implements the Boolean functions with the minimum number of transistors.*

From Theorem 5.1, we can re-formulate the problem as follows:

Problem 5.2. *Given a set of Boolean functions, find the minimum-cost equivalent AND2/INV network which implements the Boolean functions.*

The proposed algorithm is divided into two steps. The first step generates a mapping graph which implicitly enumerates possible AND2/INV networks via structural transformations. The second step produces the static CMOS circuit with the minimum number of transistors by finding the minimum-cost equivalent AND2/INV network encoded in the mapping graph.

5.4 Implicitly Enumerating AND2/INV Networks

5.4.1 Mapping Graph

A *mapping graph* proposed by Lehman *et al.* [LWGH97] efficiently encodes multiple AND2/INV networks in a single graph structure. A mapping graph is an AND2/INV network with a new type of node, called *choice node*. In a mapping graph, the output of a choice node

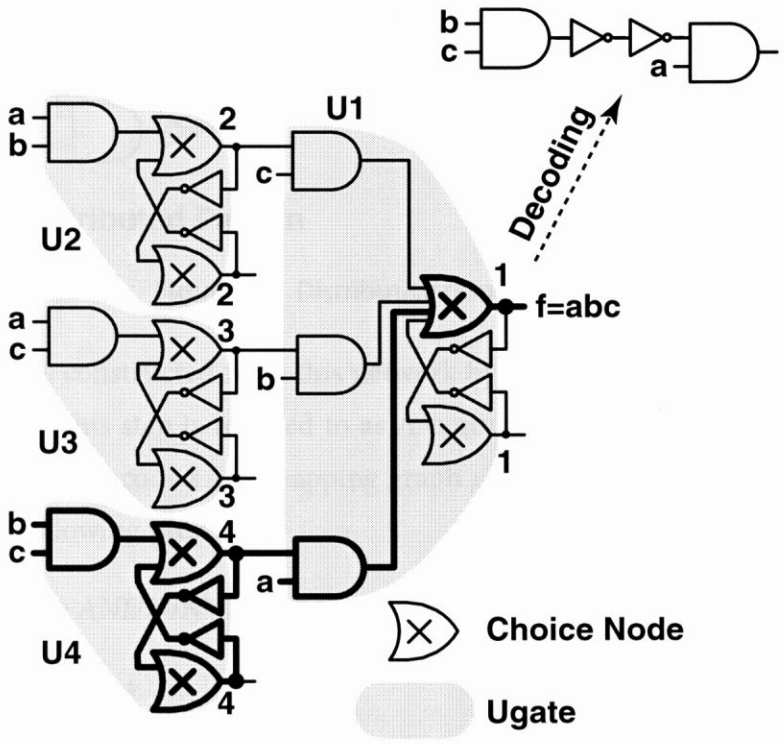


Figure 5.4 A mapping graph (lower left diagram) encoding different implementations of $f = abc$. The highlighted portion in the mapping graph generates the AND2/INV network shown in the upper right diagram. The number shown next to each choice node is the label assigned to the choice node.

represents a unique Boolean function. In other words, there cannot be two choice nodes which represent the same Boolean function. All inverters and 2-input AND gates with logically-equivalent outputs are connected to the corresponding choice node as its fanins. Given a mapping graph, an AND2/INV network is decoded by selecting one or more fanins at each choice node. Figure 5.4 shows a mapping graph encoding different implementations of $f = abc$. In the figure, a mapping graph is partitioned into disjoint subgraphs, called *ugates*. The cycles introduced by inverters in a ugate are a mechanism to encode an inverter chain with an arbitrary number of stages.

5.4.2 Constructing a Mapping Graph

Along with the mapping graph structure, Lehman *et al.* also provided the following procedure which encodes in a mapping graph all possible algebraic decompositions of a Boolean network. First, a Boolean network η is decomposed into an arbitrary AND2/INV network and then every adjacent AND gates are collapsed into a bigger AND gate as much as possible.

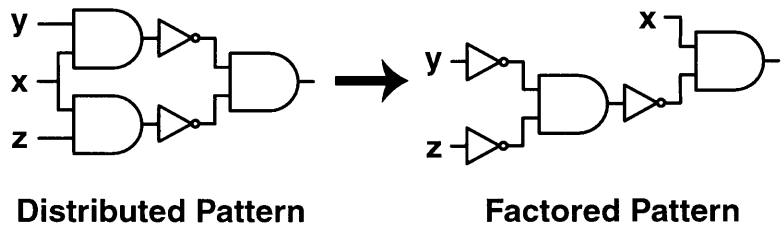


Figure 5.5 Distributive transformation.

A mapping graph is constructed from this network by encoding all possible decompositions of each AND gate. This step is referred to as Λ-construction step. We denote the set of all AND2/INV networks encoded in a mapping graph μ by $\Psi(\mu)$. Then, the resulting mapping graph μ_η^Λ has the following property:

Theorem 5.2. *Every AND2/INV decomposition of a Boolean network η is contained in $\Psi(\mu_\eta^\Lambda)$.*

Proof. Refer to Theorem 4.1 in [LWGH97]. ■

Then, the distributive transformation shown in Figure 5.5 is exhaustively applied to μ_η^Λ. This step is referred to as Δ-construction step and the resulting mapping graph μ_η^Δ has the following property:

Theorem 5.3. *Every AND2/INV decomposition of an arbitrary algebraic decomposition of a Boolean network η is contained in $\Psi(\mu_\eta^\Delta)$.*

Proof. Refer to Theorem 4.2 in [LWGH97]. ■

Obviously, the initial Boolean network η determines the set $\Psi(\mu_\eta^\Delta)$ of AND2/INV networks encoded in the final mapping graph μ_η^Δ. In our approach, we use a two-level network as an initial Boolean network. In the initial Boolean network, each output has a combinatorial node representing the *sum of all prime implicants* of the output function. A mapping graph is constructed from the Boolean network by the Λ-construction step. Similarly, another mapping graph is constructed from a Boolean network where each output has a combinatorial node representing the *sum of all prime implicants* of the *complementation* of the output function. The two mapping graphs are merged into a single mapping graph μ_P^Λ.

Lemma 5.3. *For an arbitrary prime-and-irredundant two-level Boolean network η, every AND2/INV decomposition of the Boolean network η is contained in $\Psi(\mu_P^\Lambda)$.*

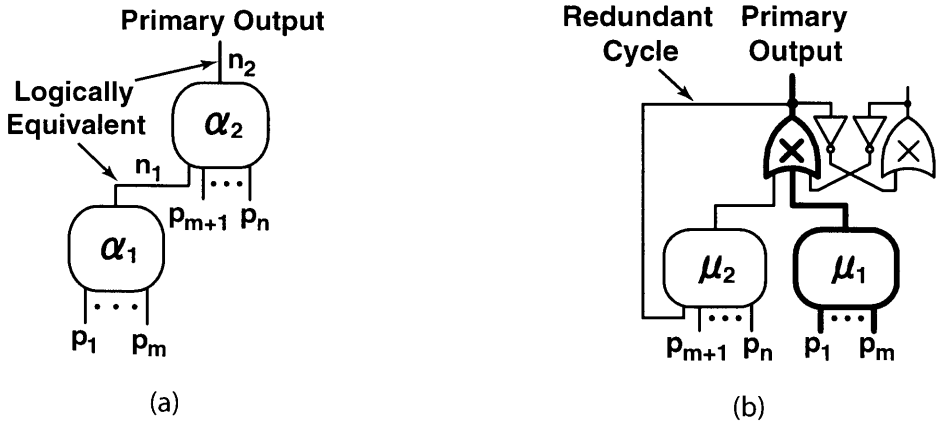


Figure 5.6 An illustration of the proof of Theorem 4.2: (a) an AND2/INV decomposition of two-level Boolean network where α_1 and α_2 are the AND2/INV networks representing the logic-OR of the inputs and (b) a mapping graph constructed from (a).

Proof. Let f be the Boolean function of a primary output or its complement and let $p_1, ..., p_n$ be the all prime implicants of f. Consider an arbitrary prime-and-irredundant two-level expression $F = p_1 + \cdots + p_m$ of the Boolean function f where $p_1, ...p_m$ are an irredundant set of the prime implicants. From Theorem 5.2, $\Psi(\mu_P^{\wedge})$ must contain an AND2/INV decomposition of η illustrated in Figure 5.6 (a). Since the nodes n_1 and n_2 in the network are logically equivalent, they are the fanins of the same choice node in the resulting mapping graph. Figure 5.6 (b) shows the mapping graph where μ_1 and μ_2 are the partial mapping graphs which are constructed from α_1 and α_2, respectively. Therefore, $\Psi(\mu_1) \subseteq \Psi(\mu_P^{\wedge})$. From Theorem 5.2, $\Psi(\mu_1)$ contains every AND2/INV decomposition of F. ■

Then, the Δ-construction step is performed on $\mu_P^{\wedge}$. The resulting mapping graph μ_P^{Δ} has the following property:

Theorem 5.4. *For an arbitrary prime-and-irredundant two-level Boolean network η, every AND2/INV decomposition of an arbitrary algebraic decomposition of the Boolean network η is contained in $\Psi(\mu_P^{\Delta})$.*

Proof. Suppose an arbitrary prime-and-irredundant two-level Boolean network η. From Lemma 4.1, $\mu_P^{\wedge}$ satisfies Theorem 5.2 for η. From this fact and Theorem 5.3, every AND2/INV decomposition of an arbitrary algebraic decomposition of η is contained in $\Psi(\mu_P^{\Delta})$. ■

As mentioned in the proof, the proposed procedure introduces redundant cycles in a mapping graph. Every redundant cycle except the cycles in a ugate can be removed.

Figure 5.7 shows an informal illustration of the relationships between the circuit structures encoded in μ_P^{Δ} and other circuit structures. For the reduction of transistor counts, we are par-

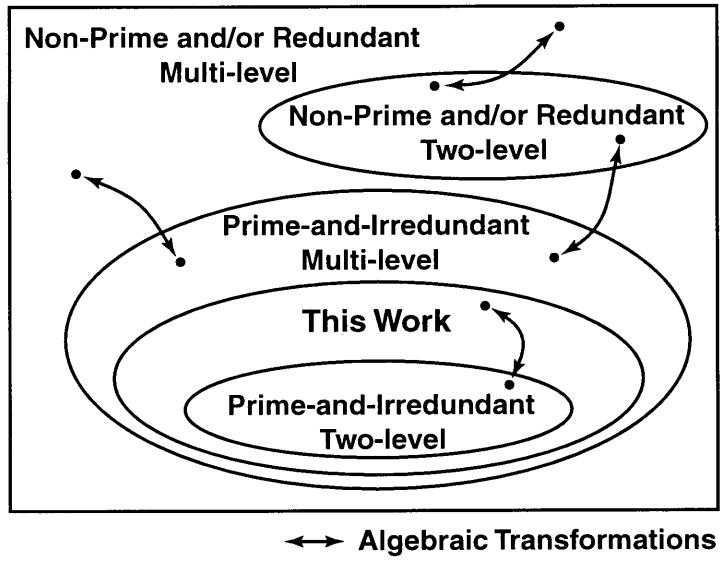


Figure 5.7 A Venn diagram which informally illustrates the relationships between the circuit structures encoded in μ_P^{Δ} and other circuit structures. Note that the sets of non-prime and/or redundant circuits are infinite sets.

ticularly interested in the set of prime-and-irredundant circuits. $\Psi(\mu_P^{\Delta})$ contains every circuit structures which can be obtained by performing algebraic transformations on a prime-and-irredundant two-level circuit. However, there can exist prime-and-irredundant multi-level circuits which are not contained in $\Psi(\mu_P^{\Delta})$. Those circuits can be obtained only by performing algebraic transformations on a non-prime and/or redundant circuit, or by performing non-algebraic (*i.e.* Boolean) transformations.

5.5 Finding the Minimum Circuit

5.5.1 Naive Approach

Given a mapping graph μ, the objective of this step is to find the minimum-cost equivalent AND2/INV network encoded in μ. One naive approach for finding the minimum solution is to exhaustively enumerate all possible equivalent AND2/INV networks encoded in μ and pick up the minimum-cost network. The choice nodes are visited in a topological order starting from the primary outputs. At each choice node, all possible matches are identified using the patterns shown in Figure 5.2. A choice node can be duplicated if there are two or more fanouts. If the choice node is a fanin of a primary output or has multiple fanouts after the duplication, only the inverter pattern is allowed to match at the choice node. Both of the AND and OR patterns match only at single-fanout choice nodes. This guarantees any resulting AND2/INV

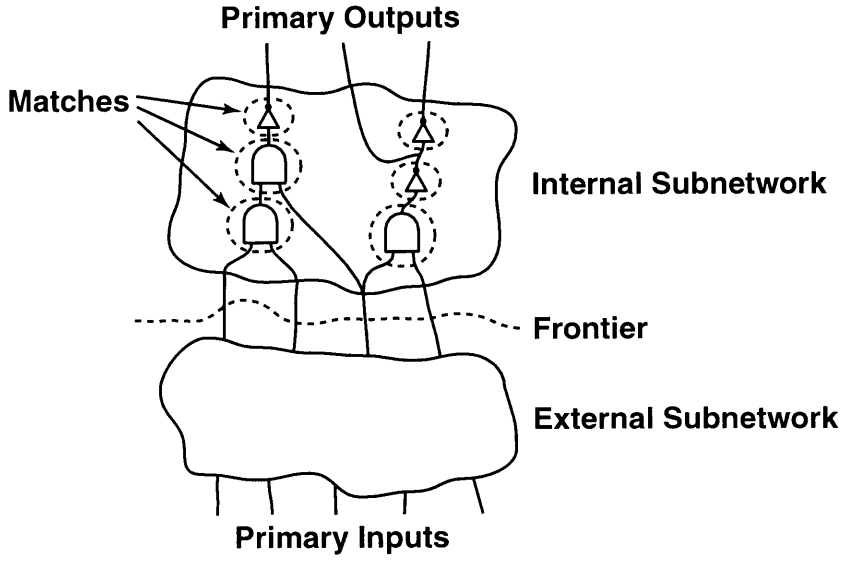


Figure 5.8 A partially-covered mapping graph

network to be an equivalent AND2/INV network.

Obviously, this naive approach is computationally too expensive. The runtime complexity of this approach is $O(s)$ where s is the number of structures explored during the search. For a mapping graph with n choice nodes where each choice node has k fanins, there are k^n AND2/INV networks in general. Based on the observation that the minimum solution for a subnetwork can be obtained independently of the solution for the remaining portion of the network, the proposed algorithm is based on *dynamic programming* [KV02]. Here, note that the proposed dynamic programming based algorithm is different from that of the tree covering [DGR+87] in the context of the technology mapping.

5.5.2 Dynamic Programming Based Algorithm

Suppose that a mapping graph μ is partially covered by the matched patterns from the primary outputs as illustrated in Figure 5.8. We define a *(partial) cover* γ as a circuit consisting of the matches. A match ϕ is a network of 2-input AND gates and inverters, and corresponds to one of the patterns shown in Figure 5.2. A match ϕ is an *input match* if and only if every input of ϕ is also an input net of γ, and we denote the set of input matches by $\Phi(\gamma)$. A frontier λ is a set of the nets in μ which correspond to the input nets of a partial cover. An internal subnetwork μ_λ^I is defined as a subnetwork of μ which consists of the transitive fanouts of λ. Similarly, an external subnetwork μ_λ^E is defined as a subnetwork of μ which consists of the transitive fanins of λ. A frontier λ is used to specify a partial cover and a subnetwork (*e.g.* γ_{λ_1},

$\mu^I_{\lambda_2}, \mu^E_{\lambda_3}$). The following lemma shows that the problem can be solved efficiently using dynamic programming.

Lemma 5.4. *Let γ_λ be the minimum-cost cover of μ^I_λ and let $\gamma_{\lambda-\phi}$ be the cover by removing $\phi \in \Phi(\gamma_\lambda)$ from γ_λ. Then, $\gamma_{\lambda-\phi}$ is the minimum-cost cover of $\mu_{\lambda-\phi}$.*

Proof. The proof is by contradiction. Let $\gamma^*_{\lambda-\phi}$ be the minimum-cost cover of $\mu^I_{\lambda-\phi}$ and γ^*_λ be the cover by adding ϕ to $\gamma^*_{\lambda-\phi}$. Assume that $\gamma_{\lambda-\phi}$ is not the minimum-cost cover of $\mu_{\lambda-\phi}$:

$$C(\gamma_{\lambda-\phi}) > C(\gamma^*_{\lambda-\phi}) \tag{5.1}$$

where $C(\gamma)$ is the cost of γ. Since every match has a cost of 2 by the definition, $C(\gamma_\lambda) = C(\gamma_{\lambda-\phi}) + 2$ and $C(\gamma^*_\lambda) = C(\gamma^*_{\lambda-\phi}) + 2$. Under the assumption of the inequality 5.1, we can derive

$$C(\gamma_\lambda) > C(\gamma^*_\lambda). \tag{5.2}$$

However, the inequality 5.2 contradicts the definition that γ_λ is the minimum-cost cover of μ^I_λ. ■

The lemma implies that it is sufficient to record only the minimum solution for each internal subnetwork μ^I. Based on this property, we developed the dynamic programming based algorithm. The pseudo-code for the algorithm is shown in Figure 5.9. In a mapping graph, a label $L(c)$ is assigned to each choice node c in the mapping graph, according to the topological order starting from the primary outputs. If there is only a directed path from c to d, then $L(c) > L(d)$. If there is also a directed path from d to c, then $L(c) = L(d)$. If there is no directed path in either direction between c and d, then $L(c) \neq L(d)$. Since a mapping graph does not have any cycle except the cycles in a ugate, no two choice nodes in different ugates can have the same label. A label $L(n)$ of a net n is defined as $L(c_n)$ where c_n is the choice node whose output is connected to n. The label $L(\lambda)$ of a frontier λ is defined as the smallest label in the set of the nets in the frontier. A frontier λ is said to be *interior* of a frontier λ' if $L(\lambda) < L(\lambda')$. For instance, in Figure 5.4, the number assigned to each choice node in the mapping graph shown is the label of the choice node. In the figure, two choice nodes in a ugate have the same label, and the choice nodes in the ugate U1 have a smaller label since U1 is the fanout of the ugates U2, U3 and U4.

The algorithm starts with an initial frontier λ_1 consisting of the primary output nets. The minimum-cost solution associated with λ_1 is an empty circuit. Suppose a frontier λ with an

Minimum-Cost Cover

Input: Mapping graph

Output: Minimum-cost equivalent AND2/INV network

```
1: Assign labels (1 ≤ l ≤ l_max) to choice nodes
2: Clear the sorted queues Q[1], ..., Q[l_max]
3: Create an initial frontier λ_1 of the primary output nets
4: S[1] ← ∅
5: Q[1] ← λ_1
6: for l = 1 to l_max do
7:    while Q[l] ≠ ∅ do
8:       λ ← the first item in Q[l]
9:       Q[l] ← Q[l] − λ
10:      n ← a net in λ such that L(n) = l
11:      c ← a choice node whose output is n
12:      for all match φ at c do
13:         λ' ← the expanded frontier by including φ
14:         if C(S[L(λ)]) + 2 < C(S[L(λ')]) then
15:            S[L(λ')] ← S[L(λ)] ∪ φ
16:            Q[L(λ')] ← Q[L(λ')] ∪ λ'
17:         end if
18:      end for
19:   end while
20: end for
21: l_final ← the biggest l such that Q[l] ≠ ∅
22: return S[l_final]
```

Figure 5.9 A pseudo-code for the dynamic programming based algorithm. The frontiers in a queue are sorted in ascending order of labels.

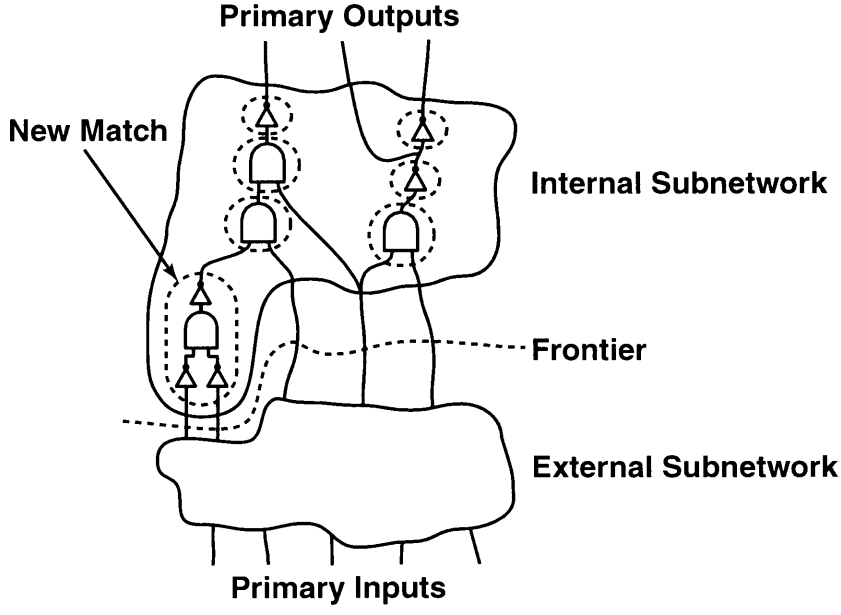


Figure 5.10 A frontier expansion on the partially-covered mapping graph shown in Figure 5.8

associated minimum-cost solution of the internal subnetwork μ_λ^I is given. First, a net n with the smallest label is picked up from λ. Then, the frontier is expanded by performing the matching procedure at c where c is the choice node whose output is connected to n. For each match ϕ, an expanded frontier λ' is generated by including ϕ in μ_λ^I. Assuming that the cost of the minimum-cost solution at the current frontier is S, then the cost at the expanded frontier is $S + 2$. The expanded frontier is recorded with its associated minimum-cost solution. If the same frontier is already visited, the solution is updated only if the new solution is better. A frontier can be expanded only if there is no other interior frontier. Figure 5.10 illustrates a frontier expansion on the partially-covered mapping graph shown in Figure 5.8.

The solution associated with the frontier consisting only of the primary input nets corresponds to the minimum-cost solution. Finally, a static CMOS circuit with the minimum number of transistors is obtained by transforming each negative unate tree in the minimum-cost AND2/INV network into a static CMOS compound gate. The following theorem guarantees the optimality of the algorithm.

Theorem 5.5. *The algorithm finds the minimum-cost equivalent AND2/INV network encoded in a mapping graph.*

Proof. Regardless of whether at each frontier all possible covers are recoded or only the minimum cover is recorded, the set of frontiers explored by the procedure remains the same. If all possible covers are recorded at each frontier, the procedure is equivalent to the naive approach

and hence finds the minimum solution. From Lemma 5.4, the optimality of the procedure is preserved even if only the minimum cover is recorded at each frontier. ■

The computational complexity of the procedure is approximated as follows. The runtime complexity is given as $O(m)$ where m is the total number of matches performed during the procedure. The space complexity is dominated by the size of the set of queues $\{Q[1], ..., Q[l_{max}]\}$ which contains all frontiers visited during the procedure. Therefore, the space complexity is approximated as $O(f)$ where f is the total number of frontiers visited during the procedure. Due to the nature of the problem, m and f are expected to be exponential to the problem size. Since it is difficult to analyze m and f theoretically, a numerical analysis using randomly-generated problems will be performed in the next section.

5.6 Experimental Results

The proposed procedure has been implemented on top of SIS [BRSVW87]. The platform was a Linux system on AMD Athlon 64 X2 4400 processor with 2 GB main memory. First, we conducted an experiment on a subset of MCNC91 benchmark circuits [Yan91]. Some of them are subcircuits of the original circuits, consisting of a subset of the primary outputs and their transitive fanins. For instance, `cm42a,e,f` is a subcircuit of `cm42a` consisting of the primary outputs `e` and `f` and their transitive fanins. It is noticeable that the sizes of the benchmark circuits in Table 5.1 are reasonably big as standard cells. Besides, in a layout implementation, transistors in a cell may be divided into smaller transistors to fit the cell height. Since standard cells have a fixed height, a cell with large number of transistors results in a very long shape and will cause difficulties in placement.

For comparison, we synthesized static CMOS circuits on the same set of benchmark circuits using SIS technology mapper as follows. A rich cell library was prepared in a similar way to that used in [DGR+87]. We generated all single-stage static CMOS cells such that the maximum number of inputs is limited to 6 and the maximum number of series-connected P-type (N-type) transistors to 4. The number of the logic functions is 461. In the library, the area of each cell is substituted with the number of transistors in the cell. By using this trick, a total cell area corresponds to the number of transistors in a circuit. Ideally, an area optimization with this library is supposed to generate a circuit with the minimum number of transistors. First, we performed an initial multi-level logic minimization using `script.algebraic` and `script.rugged`. Then, a transistor circuit is synthesized by performing an area-optimal tree mapping (`map -m 0.0`). We also implemented one of the algorithms presented most

Table 5.1 Comparison between SIS and the proposed algorithm.

Circuit	*#inputs*	*#outputs*	*SIS*		*Proposed*		*Reduction [%]*
			#transistors	*CPU time [sec]*	*#transistors*	*CPU time [sec]*	
b1	3	4	24	0.1	24	0.0	0.0
C17	5	2	24	0.1	**22**	3.3	**8.3**
cm42a,e,f	4	2	16	0.1	16	0.1	0.0
cm42a,g,h	4	2	22	0.1	**18**	0.1	**18.2**
cm42a,i,j	4	2	22	0.1	**18**	0.1	**18.2**
cm42a,k,l	4	2	20	0.1	**18**	0.1	**10.0**
cm42a,m,n	4	2	22	0.1	**18**	0.1	**18.2**
decod,f,g	5	2	18	0.1	18	5.9	0.0
decod,h,i	5	2	20	0.1	20	6.0	0.0
decod,j,k	5	2	20	0.1	20	6.0	0.0
decod,l,m	5	2	22	0.1	**20**	5.9	**9.1**
decod,n,o	5	2	20	0.1	20	5.9	0.0
decod,p,q	5	2	22	0.1	**20**	5.9	**9.1**
decod,r,s	5	2	20	0.1	20	5.9	0.0
majority	5	1	26	0.1	**20**	1.1	**23.1**
t	5	2	24	0.1	**22**	3.2	**8.3**
x2,k,l	3	2	22	0.1	**20**	0.0	**9.1**
x2,m,o	6	2	22	0.1	**20**	1.4	**9.1**
z4ml,27	3	1	20	0.1	20	0.1	0.0

recently [LA99]. Since the algorithm requires an optimized Boolean network as an input, we used the Boolean networks generated by the same logic minimization described above. Based on our experiments on the same set of problems, our implementation of the algorithm produced almost the same results as those of the SIS-based method and no better results were obtained.

Table 5.1 shows the comparison between the SIS-based method and the proposed algorithm. In the table, the rightmost column shows the reduction rates of transistor counts. Table 5.2 shows the statistics of the mapping graph constructed during the synthesis. The number of ugates in the constructed mapping graph and the numbers of frontiers and matches during the minimum solution search are also presented. The CPU time includes the time consumed by the mapping graph construction and the minimum solution search. As can be seen from the table, the proposed procedure could reduce the number of transistors up to 23.1%, and the runtime is reasonably small. The proposed procedure failed to solve other bigger problems in the MCNC91 benchmark circuits.

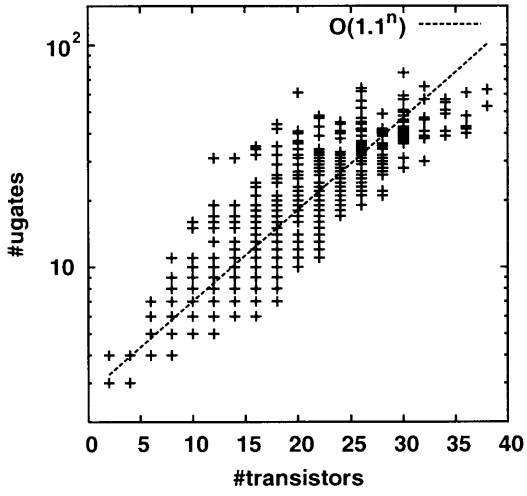
Next, we conducted another experiment on randomly-generated problems in order to

Table 5.2 Statistics of the mapping graphs.

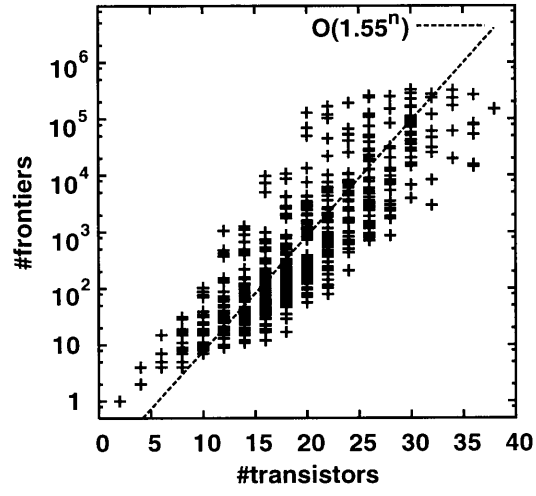
Circuit	*#ugates*	*#frontiers*	*#matches*
b1	15	292	610
C17	36	35557	83349
cm42a,e,f	22	1535	4064
cm42a,g,h	22	1535	4064
cm42a,i,j	22	1535	4064
cm42a,k,l	22	1535	4064
cm42a,m,n	22	1535	4064
decod,f,g	46	63806	199104
decod,h,i	46	63806	199104
decod,j,k	46	63806	199104
decod,l,m	46	63806	199104
decod,n,o	46	63806	199104
decod,p,q	46	63806	199104
decod,r,s	46	63806	199104
majority	63	12939	33772
t	36	35557	83349
x2,k,l	12	226	457
x2,m,o	66	16731	53126
z4ml,27	33	6407	18809

demonstrate the efficiency of the proposed algorithm. We generated 1000 problems randomly as follows. First, the numbers of inputs and outputs are randomly determined. Then, for each output, a sum-of-products expression is generated as the output function by randomly determining the number of products and generating the products randomly. In this experiment, the number of inputs is limited up to 6 and the number of outputs is limited up to 3. Next, the proposed algorithm is applied to the problems. If the total number of the matches exceeded 10^6 during the minimum solution search, the procedure is terminated and the problem is discarded. Figure 5.11 (a)-(d) show the statistics of the problems.

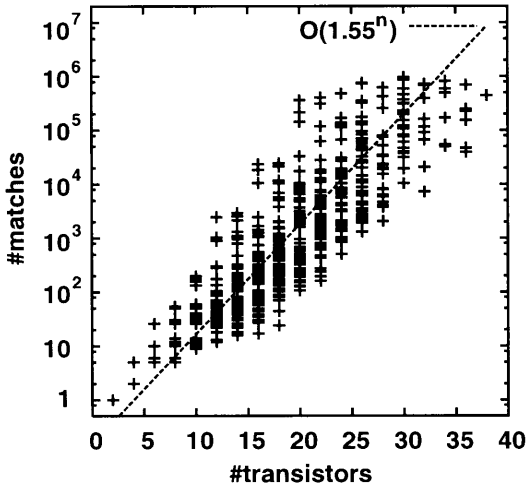
Since 146 problems were discarded in this experiment, the 854 points are plotted in the graphs. In all graphs, the X-axis correspond to the number of transistors in the resulting static CMOS circuit. Figure 5.11 (a) shows the number of ugates in the constructed mapping graph, which is approximately $O(1.1^n)$ where n is the number of transistors. Figure 5.11 (b) and (c) show f and m, where f and m are the total numbers of frontiers and matches during the dynamic-programming based algorithm, respectively. As explained in the previous section, f and m correspond to the space and runtime complexities of the dynamic-programming based algorithm respectively, and both are approximated by $O(1.55^n)$. To show the efficiency, we also



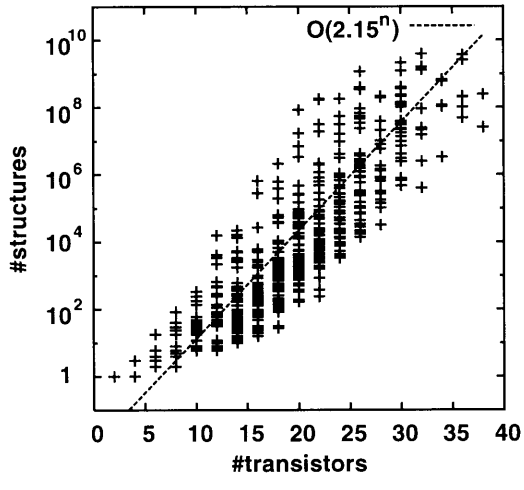
(a) Number of ugates in the constructed mapping graph.



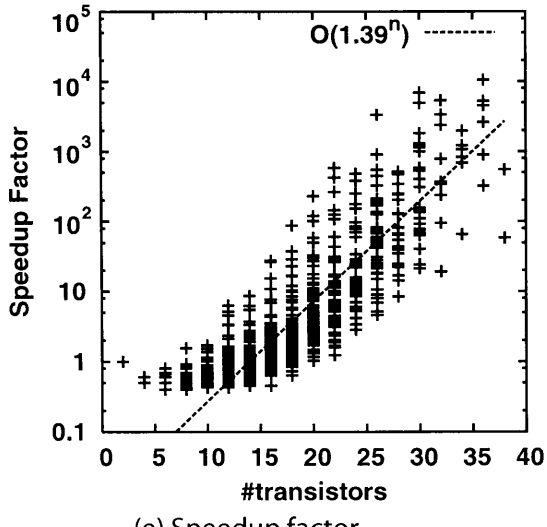
(b) Number of frontiers visited during the dynamic-programming based search.



(c) Number of matches during the dynamic-programming based search.



(d) Number of structures explored during the naive approach.



(e) Speedup factor.

Figure 5.11 Statistics on randomly generated problems.

applied the naive approach explained in Section 5.5.1 to the same set of the problems. Figure 5.11 (d) show s, where s is the number of structures explored during the naive approach. The runtime complexity of the naive approach is approximated by s and hence $O(2.15^n)$. Figure 5.11 (e) shows the speedup factor which is calculated as the ratio of s to m. The speedup factor quantifies the runtime efficiency of the dynamic-programming based algorithm against the naive approach, and is approximated by $O(1.39^n)$. Since the current implementation allocates ~1000 bytes for each frontier, the memory is exhausted when solving problems with more than 10^6 frontiers. Based on this observation, the current implementation of the proposed algorithm can solve problems with 30-40 transistors. This also explains the reason why big problems could not be solved in the previous experiment on the MCNC91 benchmark circuits.

5.7 Conclusions

Transistor-level optimization is known as a powerful technique to improve the circuit area and performance beyond gate-level optimization. In this chapter, we presented a structural approach for synthesizing an arbitrary static CMOS circuits targeting the reduction of transistor counts. The circuit structures are implicitly enumerated via structural transformations on a single graph structure, then a dynamic-programming based algorithm efficiently finds the minimum solution among them. We also showed that the solution space contains the circuit structures which can be obtained by performing algebraic transformations on an arbitrary prime-and-irredundant two-level circuit, and the proposed algorithm is guaranteed to find the optimal solution within it. The experimental results on a benchmark suite targeting standard cell implementations demonstrated the feasibility of the proposed procedure. We also demonstrated the efficiency of the proposed algorithm by a numerical analysis on randomly-generated problems. It is also shown that the proposed procedure sometimes generates significantly smaller circuits compared to SIS-based approach. This fact reconfirms a potential of transistor-level optimization for area minimization.

Chapter 6

Exact Minimum Logic Factoring via Quantified Boolean Satisfiability

6.1 Introduction

Logic factoring is an operation to find a factored form from a Boolean function. The factored form is known as one of the efficient representation styles of Boolean functions and forms a basis of multiple-level logic. Since the factored form corresponds to a static CMOS compound gate, it has also been used to estimate the circuit area required to implement a Boolean network. For example, the static CMOS compound gate illustrated in Figure 6.1 implements the complementation of a factored form expression: $y = \overline{((ab + c) * d) + ef}$. Although logic factoring is one of the most fundamental problems in multiple-level logic synthesis, the exact solution to this problem is still challenging. Early works [Law64, Dav69] provided exact multiple-level logic minimization algorithms, however, the algorithms are very inefficient and apply to very small functions. An improved algorithm of [Dav69] requires a couple of hours to synthesize small circuits with ~12 2-input NAND gates [DG98]. In addition to their computational costs, these algorithms cannot be applied to the minimization of the number of literals in the factored form.

Recently, Boolean satisfiability solvers have made a dramatic improvement [SS97, MMZ+01] and have been successfully applied to industrial-scale EDA problems such as automatic test pattern generation [Lar92] and symbolic model checking [BCCZ99]. Quantified Boolean formula, which is a generalization of propositional logic, is known as a more natural way to model such problems. As a consequence, a number of efficient QBF decision algorithms have been proposed [FMS00, Bie05].

In this chapter, we present an exact factoring method which transforms the factoring problem into a QBF and solving it using general-purpose QBF solver. The solution space explored in this method is illustrated in Figure 1.3. Since the size of the QBF dominantly determines

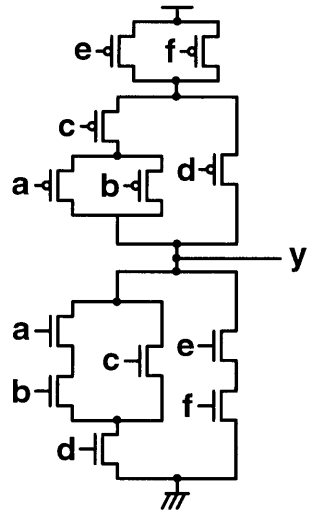


Figure 6.1 A static CMOS compound gate.

the runtime of the decision procedure, the QBF must be constructed as compact as possible. For this purpose, we propose a novel graph structure, called as an X-B tree, which implicitly enumerates all possible binary trees with a given number of leaf nodes. Experimental results show that the proposed method generated the exact minimum solutions to the problems with up to 12 literals in an hour.

The rest of the chapter is organized as follows. In Section 6.2, we introduce a novel graph structure, called as an X-B (eXchanger Binary) tree, and then present a method for generating X-B trees. In Section 6.3, we propose an exact method for minimum logic factoring. After formulating the problem, we explain how to transform the problem into a structural representation using an X-B tree and how to represent the structural representation as a quantified Boolean formula. Section 6.4 presents experimental results on a set of artificially-created benchmark functions. Conclusions are drawn in Section 6.5.

6.2 X-B Tree and Its Generation

6.2.1 X-B Tree

A *binary tree* is a graph with *internal nodes* and *leaf nodes* in which every internal node has two children. A *child* is either an internal node or a leaf node. An *X-B (eXchanger Binary) tree* is a rooted unordered binary tree with a new type of nodes, called an *exchanger node*. An exchanger node has the same number of inputs $\{i_1, ..., i_n\}$ and outputs $\{o_1, ..., o_n\}$. Only one of the children is an internal node, and the others are either a leaf node or an exchanger node. An exchanger node has an associated value, called an *exchange index* which determines how the

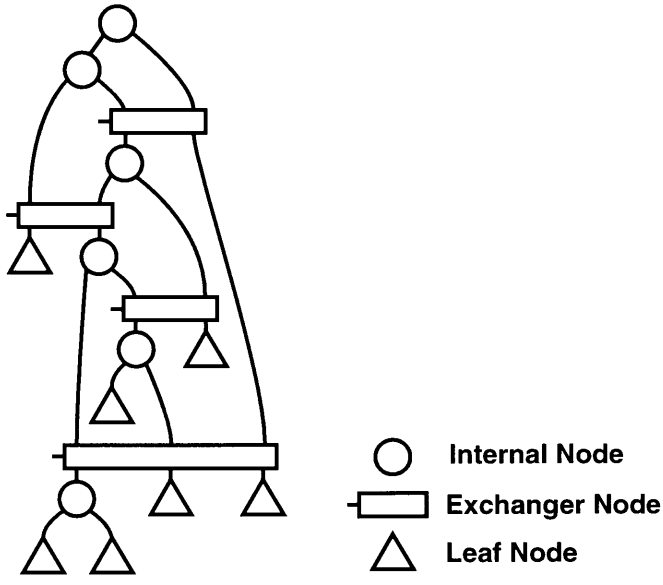


Figure 6.2 An X-B tree with 7 leaf nodes.

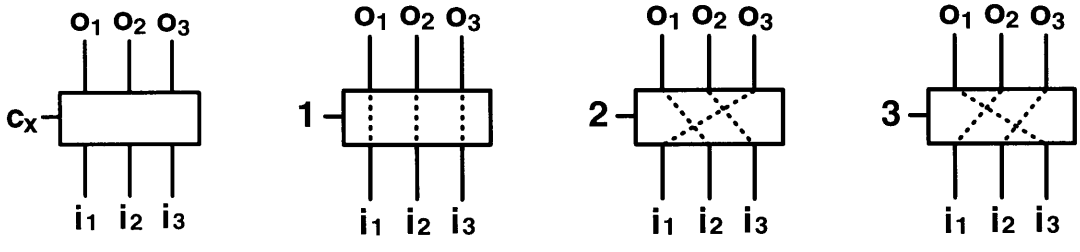


Figure 6.3 An example of 3-input exchanger node.

inputs and the outputs are connected. An exchange index c_x is a positive integer $1 \leq c_x \leq n$ where n is the number of the inputs (outputs) of the exchanger. Then, the relation between the inputs and the outputs is given as follows:

$$o_j = i_{((c_x+j-2) \bmod n)+1} \quad (1 \leq j \leq n). \tag{6.1}$$

where i_j is the j-th input and o_j is the j-th output. In other words, an exchanger node can be viewed as a bit shifter. An X-B tree with 7 leaf nodes is shown in Figure 6.2 and a 3-input exchanger node is shown in Figure 6.3.

Once the assignment of the exchange indices is determined, a binary tree is obtained. By exploring such assignments, all possible binary trees can be obtained. Note that there is not necessarily a one-to-one correspondence from an assignment of exchange indices to a binary tree, *i.e.*, different assignments of exchange indices can result in the same binary tree.

When the logic factoring problem is formulated as a QBF, as explained in the next section,

the exchange index of each exchanger node is represented as a vector of binary variables. The *number of total exchange bits* is the number of binary variables required to represent all exchange indices in an X-B tree:

$$n_b = \sum_i \lceil \log_2 n_i \rceil \tag{6.2}$$

where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x, and n_i is the number of inputs of i-th exchanger node in an X-B tree.

6.2.2 Signatures of Binary Trees

During the construction of X-B trees, it is necessary to check the isomorphism between two binary trees. We use the *bitstring representation* [Pro80, Zak80] as the signature of binary trees. Since the original bitstring representation is for ordered binary trees, we extend it for unordered binary trees. The bitstring representation is a binary sequence $b_1 b_2 \ldots b_{2n}$ obtained recursively, as described in Figure 6.4. In the procedure, the last bit is always 0 and hence is omitted.

6.2.3 Generating X-B Trees

Given the number of leaf nodes, there are many choices of X-B trees depending on how the exchanger nodes are connected. Since we are particularly interested in the minimum X-B tree, all possible X-B trees are first enumerated and the minimum one is chosen.

The proposed generation procedure is constructive in the sense that the X-B trees with L leaf nodes are constructed from the X-B trees with $L-1$ leaf nodes. The procedure starts with a binary tree with one internal node and two children. Given an X-B tree, a set of leaf nodes and their parents is identified as an insertion point. Then, an exchanger node and an internal node are inserted at the point, as illustrated in Figure 6.5.

The insertion points are computed as follows. First, a leaf node is replaced with an internal node and two leaf nodes. Then, all the signatures are computed by exploring all possible assignments of the exchange indices. After computing the signatures for all leaf nodes, a covering table is constructed where the rows correspond to the signatures and the columns to the leaf nodes. By solving the covering problem, the minimum set of leaf nodes is found and an exchanger node is inserted between the leaf nodes and their parents.

ComputeSignature

Input: Tree T

Output: Bitstring representation S

```
1: V ← the root node of T
2: S ← ComputeSignatureNode(V)
3: Omit the last bit from S
4: return S
```

(a) Top-level procedure ComputeSignature.

ComputeSignatureNode

Input: Tree node V

Output: Bitstring representation S

```
1: if V is a leaf node then
2:     return "0"
3: end if
4: V_L ← the left child of V
5: V_R ← the right child of V
6: S_L ← ComputeSignatureNode(V_L)
7: S_R ← ComputeSignatureNode(V_R)
8: if S_L > S_R then
9:     return "1"+S_L+S_R
10: else
11:     return "1"+S_R+S_L
12: end if
```

(b) Procedure ComputeSignatureNode which is called by ComputeSignature.

Figure 6.4 Basic procedure for signature computation.

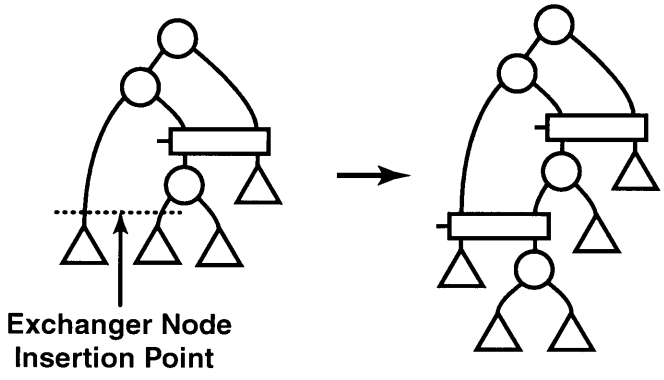


Figure 6.5 Inserting an exchanger node.

6.2.4 Complexity of X-B Trees

Table 6.1 shows the characteristics of the minimum X-B trees obtained by the method in the previous section. Obviously, the numbers of leaf nodes L, internal nodes N and exchanger nodes X hold the following relation:

$$L = N + 1 = X + 3. \tag{6.3}$$

An upper bound on the number of total exchange bits is derived as follows. Suppose that an X-B tree with l leaf nodes is being constructed by inserting an exchanger node into an X-B tree with $l - 1$ leaf nodes. In the worst case, the number of inputs of the exchanger node is $l - 2$. Hence, an upper bound on the number of total exchange bits n_b is calculated from the equation 6.2:

$$n_b = \sum_{l=4}^{L} \lceil \log_2(l-2) \rceil < \sum_{l=4}^{L} \lceil \log_2 L \rceil = O(L \log_2 L). \tag{6.4}$$

Thus, X-B trees can efficiently encode exponential number of binary trees in a single graph.

6.3 Exact Minimum Factoring

6.3.1 Problem Formulation

A *literal* is a variable or its negation. A *factored form* is a representation of a Boolean function and defined recursively as follows: 1) a literal is a factored form; 2) a sum of factored forms is a factored form; 3) a product of factored forms is a factored form. In general, the factored form of a Boolean function is not unique. For example, the following expressions; $abc + abd + cd$, $ab(c + d) + cd$ and $abc + (ab + c)d$ are all the factored forms of a Boolean function. A factored

Table 6.1 Characteristics of minimum X-B trees.

#leaf nodes	*#internal nodes*	*#exchangers*	*#total exchange bits*	*#encoded binary trees*
2	1	0	0	1
3	2	0	0	1
4	3	1	1	2
5	4	2	2	3
6	5	3	3	6
7	6	4	5	11
8	7	5	7	23
9	8	6	9	46
10	9	7	11	98
11	10	8	13	207
12	11	9	15	451
13	12	10	17	983
14	13	11	20	2179
15	14	12	22	4850
16	15	13	25	10905

form is *minimum* if and only if the number of literals is the least among all possible factored forms.

An AND/OR binary tree is a rooted binary tree where the type of each internal node is either a 2-input AND operator or a 2-input OR operator. By regarding leaf nodes as literals, an arbitrary factored form can be represented as an AND/OR binary tree.

The problem addressed in this chapter can be formulated as follows: *Given an incompletely specified Boolean function* (f, d, r) *of variables* $V = \{v_1, \ldots, v_{|V|}\}$, *find a factored form with the minimum number of literals.* Alternatively, we can formulate it as follows: *Given an incompletely specified Boolean function* (f, d, r), *find an AND/OR binary tree with the minimum number of leaf nodes which implements the Boolean function.*

6.3.2 Constructing a QBF

The problem is modeled as a miter structure [Bra93] illustrated in Figure 6.6. It checks the equivalence between the given Boolean function and an AND/OR X-B tree. An AND/OR X-B tree is an X-B tree with the following modifications. An internal node, called as an *operator node*, has its associated variable $c_o \in \{0, 1\}$ to specify whether the node type is AND or OR.

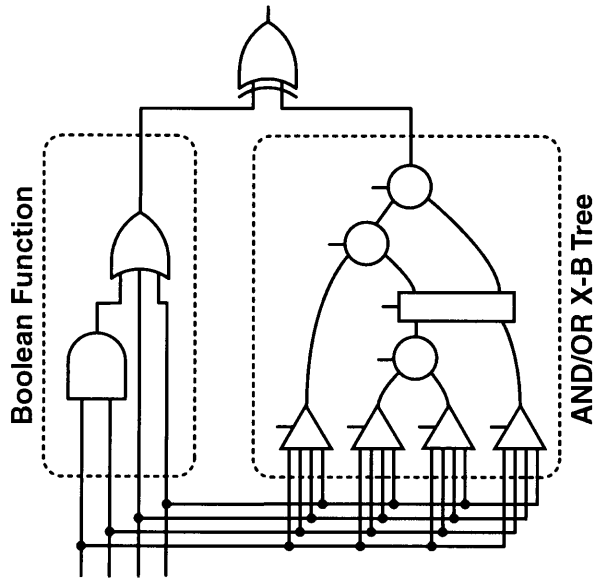


Figure 6.6 A miter structure.

Figure 6.7 shows an operator node and its equivalent logic circuit. A leaf node, called as a *literal node*, has its associated variable $c_l \in \{1, \ldots, 2|V|\}$ to specify a literal $l \in \{v_1, \overline{v_1}, \ldots, v_{|V|}, \overline{v_{|V|}}\}$. Figure 6.8 shows a literal node and its equivalent logic circuit. The three types of variables, c_x, c_o and c_l, are called as *configuration variables* $C = \{c_1, \ldots, c_{|C|}\}$. An arbitrary AND/OR binary tree with a given number of leaf nodes can be represented by an AND/OR X-B tree with an assignment of the configuration variables.

A quantified Boolean formula is constructed based on this model. The clauses of the quantified Boolean formula consist of four categories: *function constraints*, *operator node constraints*, *exchanger node constraints* and *literal node constraints* where each constraint corresponds to a node in the miter structure.

Function Constraints

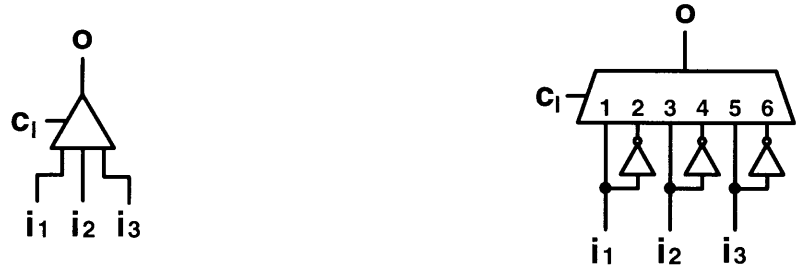
Let o_{root} be the variable corresponding to the output of the root operator node in the AND/OR X-B tree. The function constraints check the equivalence of the Boolean function and the AND/OR X-B tree:

$$\xi_f = (f \equiv o_{root}) + d \tag{6.5}$$

where f and d are the on set and the *don't care* set of the given Boolean function, respectively. If the assignment of the input variables is *don't care*, ξ_f is true regardless of the values of f and o_{root}. Thus, the *don't care* condition is taken into account.

(a) (b)

Figure 6.7 (a) operator node and (b) its equivalent logic circuit.



(a) (b)

Figure 6.8 (a) literal node and (b) its equivalent logic circuit.

Operator Node Constraints

The operator node constraints represent the operator nodes in the AND/OR X-B tree. For each operator node, the following formula is constructed:

$$\xi_o = \overline{c_x}(o \equiv (i_1 + i_2)) + c_x(o \equiv (i_1 \cdot i_2)) \tag{6.6}$$

where i_1 and i_2 are the variables corresponding to the outputs of the child nodes of the operator node.

Exchanger Node Constraints

Let n be a positive integer $1 \leq n \leq m$. Then, a cube representation of n is defined as follows:

$$\mathcal{CUBE}(x, m, n) = \prod_{i=1}^{\lceil \log_2 m \rceil} (x_i \equiv b_i) \tag{6.7}$$

where $b_1 b_2 \ldots b_{\lceil \log_2 m \rceil}$ is a binary bit-vector representation of a decimal integer $n - 1$. For example, $\mathcal{CUBE}(4, 1) = \overline{x_1} \cdot \overline{x_2}$, $\mathcal{CUBE}(4, 2) = x_1 \cdot \overline{x_2}$, $\mathcal{CUBE}(4, 3) = \overline{x_1} \cdot x_2$ and $\mathcal{CUBE}(4, 4) = x_1 \cdot x_2$.

The exchanger node constraints represent the exchanger nodes in the AND/OR X-B tree. For each exchanger node, the following formula is constructed:

$$\begin{aligned}\xi_x \;=\; & \sum_{i=1}^{n}(\mathcal{CUBE}(c_o, n, i)\cdot \\ & \prod_{j=1}^{n}(o_j \equiv i_{((i+j-2)\ mod\ n)+1}))\cdot \\ & \frac{}{\sum_{i=n+1}^{2^{\lceil \log_2 n\rceil}} \mathcal{CUBE}(c_o, n, i)}\end{aligned} \tag{6.8}$$

where n is the number of the inputs (outputs) of the exchanger node, o_j is the variable corresponding to the j-th output of the operator node, and i_j is the variable corresponding to the output of the j-th child node of the operator node.

Literal Node Constraints

The literal node constraints represent the literal nodes in the AND/OR X-B tree. For each literal node, the following formula is constructed:

$$\begin{aligned}\xi_l \;=\; & \sum_{i=1}^{|V|}(\mathcal{CUBE}(c_l, 2|V|, 2i-1)(o \equiv v_i) + \\ & \mathcal{CUBE}(c_l, 2|V|, 2i)(o \equiv \overline{v_i}))\cdot \\ & \frac{}{\sum_{i=2|V|+1}^{2^{\lceil \log_2 2|V|\rceil}} \mathcal{CUBE}(c_l, 2|V|, i).}\end{aligned} \tag{6.9}$$

Constructing a Final QBF

Let $O = \{o_1, \ldots, o_{|O|}\}$ be the variables corresponding to the outputs of the exchanger, operator and literal nodes. Then, a quantified Boolean formula ξ is constructed by combining all the constraints 6.5, 6.6, 6.8 and 6.9 and introducing existential and universal quantifiers:

$$\xi = \exists C \forall V \exists O\ \xi_f\left(\prod_{i=1}^{L-1}\xi_{o_i}\right)\left(\prod_{i=1}^{L-3}\xi_{x_i}\right)\left(\prod_{i=1}^{L}\xi_{l_i}\right) \tag{6.10}$$

where ξ_{o_i}, ξ_{s_i} and ξ_{l_i} are the constraints corresponding to the i-th node of each type, and L is the numbers of the leaf nodes in the AND/OR X-B tree. Note that the number of the exchanger nodes X is given as $L - 3$ from the equation 6.3.

Minimum Logic Factoring

Input: Incompletely-specified Boolean function (f, d, r)

Output: Minimum factored form expression F

```
1: L ← |V|
2: loop
3:    X ← an AND/OR X-B tree with L nodes
4:    Construct a QBF ξ from X and (f, d, r)
5:    Solve the QBF ξ
6:    if ξ is satisfiable then
7:       A ← a satisfiable assignment of ξ
8:       T ← an AND/OR tree by assigning A to X
9:       Transform T to a factored form F
10:      return F
11:   end if
12:   L ← L + 1
13: end loop
```

Figure 6.9 Basic procedure for finding minimum factored form.

6.3.3 Finding the Minimum Factored Form

Given the number L of the leaf nodes in the AND/OR X-B tree, a QBF ξ is constructed as described in the previous section. If ξ is satisfiable, it implies that there is a factored form with L or less literals. To find the minimum factored form, we start with $L = |V|$ literals. Note that there does not exist any factored form with less than $|V|$ literals because every variable must appear in the factored form. If the QBF is satisfiable, the assignment of the configuration variables is computed and the minimum factored form is obtained. Otherwise, L is incremented by one and the procedure is repeated until the minimum factored form is obtained. Figure 6.9 describes a basic procedure for finding the minimum factored form.

6.3.4 Complexity of QBFs

First, we derive an upper bound on the number of variables used in the QBFs constructed by the proposed method. Due to space limitations, the details of the derivations are not provided. Upper bounds on the numbers of configuration variables C, function variables V and output

Table 6.2 Upper bounds on QBF sizes.

Type	*#clauses*	*#literals*
Function Constraints	$O(Prod(f) + Prod(r))$	$O(Lit(f) + Lit(r))$
Operator Constraints	$O(L)$	$O(L)$
Exchanger Constraints	$O(L^3)$	$O(L^3 log_2 L)$
Literal Constraints	$O(L\|V\|log_2\|V\|)$	$O(L\|V\|^2 log_2\|V\|)$
Overall	$O(L^3)$	$O(L^3 log_2 L)$

variables O are given as follows:

$$\begin{aligned} |C| &= (L-1) + L\lceil log_2 2V \rceil + O(L log_2 L) \\ &= O(L log_2 L) \end{aligned} \tag{6.11}$$

$$|V| = O(|V|) \tag{6.12}$$

$$|O| = L + (L-1) + \sum_{i=1}^{L} (i-1) = O(L^2). \tag{6.13}$$

where L is the number of literals, *i.e.*, the number of leaf nodes in the corresponding AND/OR X-B tree. Hence, an upper bound on the QBF variables is $O(L^2)$ since $L \geq |V|$.

Table 6.2 presents upper bounds on the numbers of clauses and literals of each constraint type in conjunctive normal form. In the table, $Prod(g)$ and $Lit(g)$ are the numbers of products and literals in disjunctive normal form (sum-of-product form) of Boolean function g. Also, f and r are the on set and off set of the incompletely specified Boolean function given as an input to the factoring problem. Overall, upper bounds on the numbers of clauses and literals are $O(L^3)$ and $O(L^3 log_2 L)$, respectively.

6.4 Experimental Results

We have implemented the proposed method called *Exact Factor* in C++ on top of the logic manipulation class library *Logica* which we have recently developed. As a QBF solver, we have examined a number of state-of-the-art QBF solvers: ssolve [FMS00], SEMPROP [Let02], sKizzo [Ben04], and Quantor [Bie05]. Among these solvers, we chose sKizzo which solved our QBF problem instances in the shortest runtime.

We conducted experiments on a set of artificially-created benchmark functions and a function `majority` from MCNC91 benchmark suite. The artificial benchmark functions are categorized into two groups: *algebraic group* and *Boolean group*. The functions in the algebraic group are the functions of which the minimum factored form can be obtained without

Table 6.3 Experimental results on benchmark functions.

Function	*#inputs*	*ESPRESSO* [BSVHM84]	*Good Factor* [BRSVW87]	*Exact Factor*	
		#literals	*#literals*	*#literals*	*CPU time [sec]*
majority	5	13	10	**9**	2.6
algebraic1	8	14	8	8	1.8
algebraic2	6	36	11	11	483.9
algebraic3	6	15	11	**10**	200.7
algebraic4	9	19	9	9	12.0
boolean1	5	11	8	**6**	0.5
boolean2	6	26	16	**10**	65.0
boolean3	5	13	10	**8**	10.2
boolean4	6	22	18	**11**	466.3
boolean5	6	30	20	**12**	319.9

the specific features of Boolean algebra. In contrast, the functions in the Boolean group are the functions of which the minimum factored form can be obtained only if the specific features of Boolean algebra are used.

The results are shown in Table 6.3. As a reference, we present the results of ESPRESSO two-level minimizer [BSVHM84] and Good Factor factoring algorithm [BRSVW87]. In the table, the first two columns give the name of the function and the number of the input variables, respectively. Columns 3, 4 and 5 show the numbers of literals of the sum-of-products form generated by ESPRESSO, the factored form generated by Good Factor, and the factored form generated by the proposed method. The last column shows the CPU time in seconds. Table 6.4 shows the statistics of the constructed QBFs when the minimum solution is found. Columns 2, 3 and 4 show the numbers of variables, clauses and literals of the final satisfiable QBF.

Since it is guaranteed that the proposed method provides the minimum factored form, we focus particularly on the quality of the results and the runtime. As can be observed, the Good Factor provides near-minimum results on the functions in the algebraic group. However, on the functions in the Boolean group, the results of the Good Factor are far from the minimum solution. For example, the resulting expressions of `boolean4` are as follows:

$$\begin{aligned}
\text{ESPRESSO:} &\quad ab\bar{c}\bar{d} + ab\bar{e} + \bar{a}\bar{b}cd + \bar{a}\bar{b}ef + cd\bar{e} + \bar{c}\bar{d}ef \\
\text{Good Factor:} &\quad cd\bar{e} + \bar{c}\bar{d}ef + \bar{a}\bar{b}(ef + cd) + ab(\bar{e} + \bar{c}\bar{d}) \\
\text{Exact Factor:} &\quad (ab + cd + ef)(\bar{a}\bar{b} + \bar{c}\bar{d} + \bar{e}).
\end{aligned}$$

Table 6.4 Statistics of the QBFs.

Function	*#variables*	*#clauses*	*#literals*
majority	90	251	825
algebraic1	81	245	991
algebraic2	115	458	2014
algebraic3	103	409	1796
algebraic4	103	299	1046
boolean1	54	194	813
boolean2	103	413	1817
boolean3	78	287	1176
boolean4	115	458	2014
boolean5	127	383	1349

It can also be observed that the size of QBFs does not simply follow the number of variables or literals. The reason is that our implementation of the logic factoring method performs some simple reductions on the QBF size as a preprocessing step if a redundancy is found in a given problem (*e.g.* a given Boolean function is unate). The proposed method could not solve the problems bigger than 12 literals in *an hour* mainly due to large CPU time of QBF satisfiability checking. QBF decision algorithm is under a heavy development and is still improving further. Hence it should be expected that bigger problems are solved in the near future.

6.5 Conclusions

Logic factoring is a fundamental but still challenging problem in multiple-level logic synthesis. In this chapter, we presented an exact method which finds the minimum factored form of an incompletely specified Boolean function. The problem is formulated as a quantified Boolean formula and is solved by general-purpose QBF solver. We also proposed a novel graph structure, called an X-B tree, which implicitly enumerates binary trees. Using this graph structure, the factoring problem is compactly transformed into a QBF. Experimental results showed that the proposed method successfully found the exact minimum solutions to the problems with up to 12 literals. Even though the size of solvable problems is limited, the proposed method is still useful for a number of applications such as standard cell design.

Chapter 7

Synthesis of Read-Once Switch Networks

7.1 Introduction

Although a switch network is known as one of the efficient representation styles of a Boolean function, its synthesis is still a challenging problem. In contrast, synthesis of a Boolean expression, which corresponds to a series-parallel switch network, has been extensively studied including both exact and heuristic methods in the last century [Qui52, Law64, BSVHM84, BRSVW87]. In addition, other representation/implementation styles such as BDDs and pass transistor logic have been also well studied [Bry86, PS88]. It is well-known that non-series-parallel switch networks can represent some classes of Boolean functions more efficiently. One familiar example of a non-series-parallel circuit is a bridge circuit as shown in Figure 7.1. A minimum series-parallel switch network implementing the same Boolean function represents a factored form $x1(x3x5 + x4) + x2(x3x4 + x5)$ and hence requires 8 switches. This efficiency comes mainly from its *bidirectionality*, however, at the same time this property complicates its manipulation. Like a series-parallel switch network, a non-series-parallel switch network can be used as a PMOS (NMOS) network of static/dynamic CMOS logic and a pass-transistor logic *etc.* There are a large number of literatures on the synthesis of more general transistor circuits such as non-series-parallel circuits [AM90] and pass-transistor circuits [YSS96, BNNSV97]. Due to their heuristic nature, these methods don't guarantee any optimality of the solution.

Our ultimate goal is to synthesize a non-series-parallel switch network representing a given Boolean function with the minimum number of switches. As a first step towards the goal, this chapter focuses on the synthesis of a read-once switch network in which every variable appears only once. The solution space explored in this method is illustrated in Figure 1.3. The proposed procedure is based on the notions of prime implicants and unateness, which establish a basis for Boolean expression synthesis. We also propose a pruning technique for an efficient search.

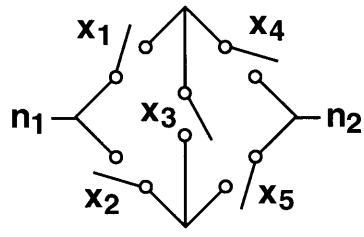


Figure 7.1 A switch network representing $x_1x_4 + x_2x_5 + x_1x_3x_5 + x_2x_3x_4$.

The rest of the chapter is organized as follows. In Section 7.2, we provide the definition of switch networks and their graph representation. In Section 7.3, we propose a method for synthesizing read-once networks. After formulating the problem, Section 7.3.1 presents how to compute the connectivity function of a switch network. Then, Section 7.3.2 proposes a synthesis procedure of read-once networks. Section 7.4 presents experimental results on randomly-generated problems to demonstrate the effectiveness and efficiency of the proposed method. Conclusions are drawn in Section 7.5.

7.2 Switch Network and Its Representation

An n-input m-output *completely-specified Boolean function* is a mapping between Boolean space $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$. In the remainder of this chapter, we will focus only on n-input 1-output completely-specified Boolean function. The cofactor of $f(x_1, ..., x_i, ..., x_n)$ with respect to x_i is $f_{x_i} = f(x_1, ..., 1, ..., x_n)$ and the cofactor with respect to $\overline{x_i}$ is $f_{\overline{x_i}} = f(x_1, ..., 0, ..., x_n)$. A Boolean function $f(x_1, ..., x_i, ..., x_n)$ is *unate in variable* x_i if $f_{x_i} \subseteq f_{\overline{x_i}}$. A Boolean function $f(x_1, ..., x_i, ..., x_n)$ is *unate* if it is unate in all support variables. A Boolean function can be expressed as a sum of products. A product term c is an *implicant* of a Boolean function f if $c \subseteq f$. An implicant is *prime* if it is not contained by any other implicant of the function.

A *switch network* is a circuit of switches where each switch is either open or close depending on the value of its associated Boolean variable. If x is associated with a switch, it is close when $x = 1$ and open otherwise. Similarly, if $\overline{x}$ is associated, it is close when $x = 0$ and open otherwise. The connectivity function of a switch network is a Boolean function f such that $f = 1$ if and only if there is a conducting path between a specified pair of terminals in the network. In the switch network shown in Figure 7.1, the connectivity function between the terminals n_1 and n_2 is $x_1x_4 + x_2x_5 + x_1x_3x_5 + x_2x_3x_4$. A switch network is *read-once* if and only if every variable appears only once in the network. Note that the connectivity function of a read-once switch network is always unate.

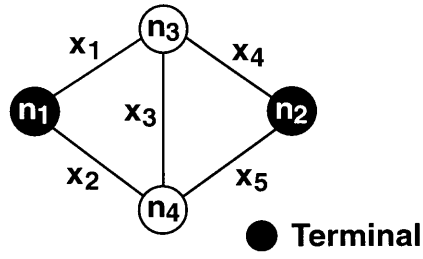


Figure 7.2 A network graph corresponding to the switch network in Figure 7.1.

A switch network can be mapped to an undirected graph $G = (V, E)$ where each vertex V corresponds to a connection point of switches and each edge E corresponds to a switch. We will refer the graph as a *network graph*. A *path* in a network graph is a set of edges between two terminals. A path corresponds to an implicant of the Boolean function represented by the graph. A network graph is *read-once* if and only if every variable appears only once in the graph. In this chapter, the two terminals for defining the connectivity function are always mapped to nodes n_1 and n_2. Figure 7.2 shows the network graph corresponding to the switch network in Figure 7.1.

7.3 Proposed Synthesis Method

The problem addressed in this chapter can be formulated as follows:

Problem 7.1. *Given a completely-specified Boolean function f, find a read-once network graph representing f.*

Note that there does *not always* exist a read-once network graph representing f. First, we explain how to compute the connectivity function of a switch network.

7.3.1 Computing the Connectivity Function

The proposed synthesis procedure described in the next section requires the computation of the connectivity function. A naive way of computing the connectivity function of a network graph is to enumerate all paths and summing the corresponding implicants to the paths. Since the number of paths can be exponential to the graph size, this approach is inefficient.

In our approach, we use a connectivity matrix to compute the connectivity function. A connectivity matrix C is an $n \times n$ matrix where n is the number of nodes in the graph and (i, j)-th entry $c_{i,j}$ is the connectivity function between the nodes n_i and n_j. A connectivity matrix is symmetric, *i.e.* $c_{i,j} = c_{j,i}$, and the main diagonal entries $c_{i,i}$ $(1 \leq i \leq n)$ is always 1.

To compute a connectivity matrix, an initial connectivity matrix C_{init} is first constructed as follows. The main diagonal entries $c_{i,i}$ $(1 \leq i \leq n)$ are set to 1 and the other entries are set to 0. For each edge x_k connecting two nodes n_i and n_j, x_k is added to both the (i, j)-th and (j, i)-th entries. By multiplying C_{init} by itself n times, the connectivity matrix C is obtained. By performing $C \times C$, $C^2 \times C^2$, $C^4 \times C^4$,..., the number of multiplications can be reduced to $\lceil \log_2 n \rceil$ times where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x. For example, the initial connectivity matrix of the network graph in Figure 7.2 is as follows.

$$\begin{pmatrix} 1 & 0 & x_1 & x_2 \\ 0 & 1 & x_4 & x_5 \\ x_1 & x_4 & 1 & x_3 \\ x_2 & x_5 & x_3 & 1 \end{pmatrix} \tag{7.1}$$

Then, the initial connectivity matrix is multiplied by itself n times. $(1, 2)$-th entry in the resulting matrix C^n gives the connectivity function. The connectivity matrix obtained by multiplying the matrix 7.1 is as follows.

$$\begin{pmatrix} 1 & x_1x_4+ \\ & x_2x_5+ \\ & x_1x_3x_5+ \\ & x_2x_3x_4 \end{pmatrix} \quad \begin{pmatrix} x_1+ \\ x_2x_3+ \\ x_2x_4x_5 \end{pmatrix} \quad \begin{pmatrix} x_1x_3+ \\ x_1x_4x_5+ \\ x_2 \end{pmatrix}$$

$$\begin{pmatrix} x_1x_4+ \\ x_2x_5+ \\ x_1x_3x_5+ \\ x_2x_3x_4 \end{pmatrix} \quad 1 \quad \begin{pmatrix} x_1x_2x_5+ \\ x_3x_5+ \\ x_4 \end{pmatrix} \quad \begin{pmatrix} x_1x_2x_4+ \\ x_3x_4+ \\ x_5 \end{pmatrix} \tag{7.2}$$

$$\begin{pmatrix} x_1+ \\ x_2x_3+ \\ x_2x_4x_5 \end{pmatrix} \quad \begin{pmatrix} x_1x_2x_5+ \\ x_3x_5+ \\ x_4 \end{pmatrix} \quad 1 \quad \begin{pmatrix} x_1x_2+ \\ x_3+ \\ x_4x_5 \end{pmatrix}$$

$$\begin{pmatrix} x_1x_3+ \\ x_1x_4x_5+ \\ x_2 \end{pmatrix} \quad \begin{pmatrix} x_1x_2x_4+ \\ x_3x_4+ \\ x_5 \end{pmatrix} \quad \begin{pmatrix} x_1x_2+ \\ x_3+ \\ x_4x_5 \end{pmatrix} \quad 1$$

By performing $C \times C$, $C^2 \times C^2$, $C^4 \times C^4$,..., the number of multiplications can be reduced to $\lceil \log_2 n \rceil$ times where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x.

7.3.2 Finding a Read-Once Network

The following theorem provides a basis for the proposed procedure:

Theorem 7.1. *Given a read-once network graph G which represents a Boolean function f and let p be an arbitrary prime implicant of f. Then, there exists a path in G such that the path represents p.*

Proof. Without loss of generality, f is assumed to be positive unate, *i.e.* every prime implicant consists only of Boolean variables but does not include any complement. By enumerating the paths in G, a sum-of-products expression $f = \sum c_i$ where c_i is a product of Boolean variables can be obtained. By removing any c_i such that $c_i \subseteq c_j$ $(i \neq j)$, one can obtain $f = \sum d_i$ where d_i is a product of Boolean variables and $d_i \not\subseteq d_j$ $(i \neq j)$. Then, the following lemma states that the set of all d_i is the set of all prime implicants.

Lemma 7.1. *Suppose $f = \sum d_i$ where d_i is a product of Boolean variables and $d_i \not\subseteq d_j$ $(i \neq j)$. Then, the set of all d_i is precisely the set of all prime implicants of f.*

Proof. Refer to Proposition 3.3.7 in [BSVHM84]. Recall that f is unate and $\sum d_i$ is minimal with respect to single cube containment. ■

Therefore, for an arbitrary prime implicant p, there exists d_i such that d_i is equivalent to p. Since there exists a path corresponding to d_i, there exists a path representing p. ■

The theorem states that a network graph can be constructed by adding the paths corresponding to the prime implicants of a given Boolean function. Since the proposed procedure basically enumerates all possible paths, the completeness is guaranteed. That is, it is guaranteed that the proposed procedure always finds the solution if it exists. As explained below, the procedure includes a pruning technique for an efficient search.

A pseudo code for the procedure is presented in Figure 7.3 and Figure 7.4. The top-level procedure FindReadOnceNetwork starts with an initial graph G^I consisting only of two terminals n_1 and n_2. The procedure AddPrimeImplicants is called with the initial graph. Suppose that the partial network graph G^P shown in Figure 7.5 is passed as an input of the procedure. First, a prime implicant p to be added is picked up from the set of ordered prime implicants P. Since all prime implicants must be added, the order of addition is not important and the exactness is preserved regardless of the order. Then, the subgraph G^S of G^P with respect to p is constructed. A subgraph $G^S = (V^S, E^S)$ of a graph G^P with respect to a prime implicant p is defined as a subgraph of G^P where E^S is the set of edges corresponding to the literals in p

FindReadOnceNetwork	
Input:	Completely-specified Boolean function f
Output:	Read-once network N
Variables:	Initial network graph $G^I = (V^I, E^I)$
	Set of ordered prime implicants $P = (p_1, \ldots, p_n)$
	Read-once network graph $G = (V, E)$

1: $P \leftarrow$ all prime implicants of f
2: Sort P
3: $V^I \leftarrow \{n_1, n_2\}$
4: $E^I \leftarrow \emptyset$
5: $G \leftarrow$ AddPrimeImplicants(f, P, G^I)
6: Transform G to a read-once network N
7: **return** N

Figure 7.3 Top-level procedure FindReadOnceNetwork.

and V^S is the set of vertices induced by E^S. Figure 7.6 shows the subgraph of Figure 7.5 with respect to a prime implicant $x_1x_3x_4x_7x_8x_9$. Next, all possible bridges are enumerated. A bridge for subgraph $G^S = (V^S, E^S)$ with respect to a prime implicant p is defined as a graph $G^B = (V^B, E^B)$ such that the graph $G^T = (V^S \cup V^B, E^S \cup E^B)$ forms the path corresponding to the prime implicant p. Figure 7.7 (a) and (b) illustrate two example bridges of the subgraph in Figure 7.6 with respect to a prime implicant $x_1x_3x_4x_7x_8x_9$ and the resulting network graph. In Figure 7.7 (a), the bridge consists of the edges x_7, x_8, x_9 and the vertex n_7. If the subgraph includes a vertex with more than two edges (*e.g.* node n_4 in Figure 7.9), there does not exist a bridge.

For each bridge, a new network graph G^T is obtained by adding the bridge to G^P. Figure 7.8 (a) and (b) show the network graphs after adding the bridges to the network graph in Figure 7.5. Since the addition of the bridge may introduce paths other than the path corresponding to the prime implicant, the connectivity function f^T of G^T is computed and is checked whether it is valid. If f^T is not contained by f, the bridge is discarded and the next bridge is examined. If this check is not performed, all possible solutions are explored. However, once the connectivity function of a graph is not contained by f, the graph is no longer considered due to the additive nature of the procedure. In this way, the search is efficiently performed while the completeness of the procedure is preserved. If f^T is equivalent to f, the solution is

AddPrimeImplicants

Input: Completely-specified Boolean function f
Set of ordered prime implicants $P = (p_1, \ldots, p_n)$
Partial network graph $G^P = (V^P, E^P)$

Output: Read-once network graph $G = (V, E)$ or $\emptyset$ if there exists no such a network graph

Variables: Subgraph $G^S = (V^S, E^S)$
Bridge $G^B = (V^B, E^B)$
Temporary network graphs $G^T = (V^T, E^T)$, $G^U = (V^U, E^U)$

```
1: p ← head[P]
2: P' ← P - {p}
3: G^S ← the subgraph of G^P with respect to p
4: for all bridge G^B of G^S do
5:    V^T ← V^P ∪ V^B
6:    E^T ← E^P ∪ E^B
7:    f^T ← connectivity function of G^T
8:    if f^T ⊈ f then
9:       continue
10:   else if f^T ⊇ f then
11:      return G^T
12:   end if
13:   G^U ← AddPrimeImplicants(f, P', G^T)
14:   if G^U ≠ ∅ then
15:      return G^U
16:   end if
17: end for
18: return ∅
```

Figure 7.4 Procedure AddPrimeImplicants which is called by FindReadOnceNetwork.

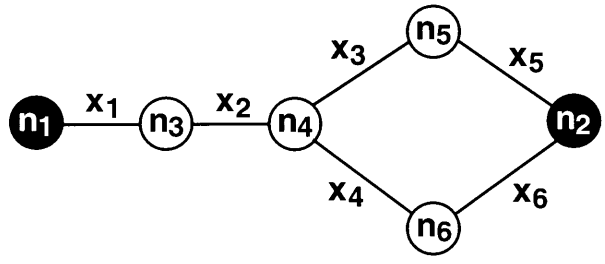


Figure 7.5 A partial network graph.

found and hence it is returned. Otherwise, AddPrimeImplicants is called recursively to add another prime implicant.

7.4 Experimental Results

7.4.1 Generating Random Problems

In this experiment, all problems are randomly generated such that they can be represented as a read-once switch network as follows. It starts with a network graph consisting only of two terminals. Then, a switch is randomly inserted between two nodes. At this point, a Boolean variable is not assigned to the switch. Once all switches are inserted, Boolean variables are randomly assigned to the switches. Finally, a Boolean function as a problem is obtained from the network graph. For each number of variables, 100 problems were generated.

7.4.2 Synthesis Results

We have implemented the proposed procedure in C++ on top of the logic manipulation class library *Logica* which we have recently developed. The platform was a Linux system on AMD Athlon 64 X2 4400 processor with 2 GB main memory. We conducted an experiment on the set of randomly-generated problems.

The results are shown in Figure 7.10 (a) and (b). Figure 7.10 (a) shows a scatter plot which compares the number of variables of the problem and the runtime required to find the solution. The procedure was terminated when the runtime of the procedure exceeds 600 seconds. In this experiment, 182 problems were discarded out of 1600 problems. From this graph, the worst-case runtime complexity can be approximated by $O(9.5^n)$ where n is the number of the variables. We also synthesized series-parallel switch networks on the same set of problems using SIS Good Factor [BRSVW87]. The factored form of a Boolean function can be viewed as a series-parallel switch network. The number of switches is obtained as the number of

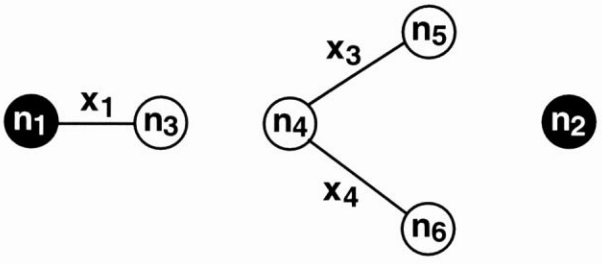
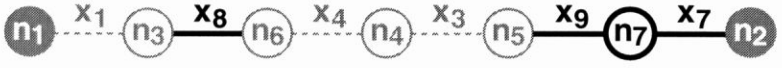


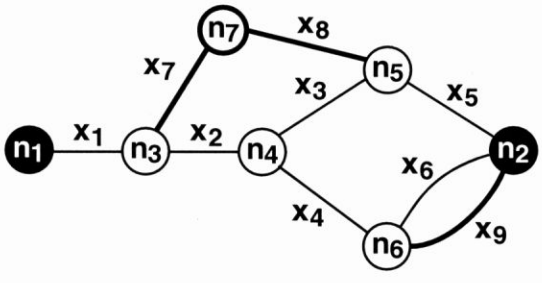
Figure 7.6 The subgraph of Figure 7.5 with respect to a prime implicant $x_1x_3x_4x_7x_8x_9$.

(a)

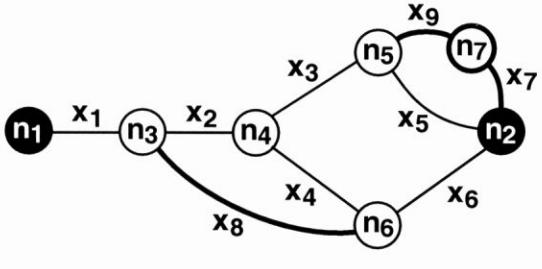


(b)

Figure 7.7 Two example bridges (thick portion) of the subgraph in Figure 7.6 with respect to a prime implicant $x_1x_3x_4x_7x_8x_9$.



(a)



(b)

Figure 7.8 The resulting network graphs by adding the two bridges in Figure 7.7 to the subgraph in Figure 7.6. (a) and (b) correspond to Figure 7.7 (a) and (b), respectively.

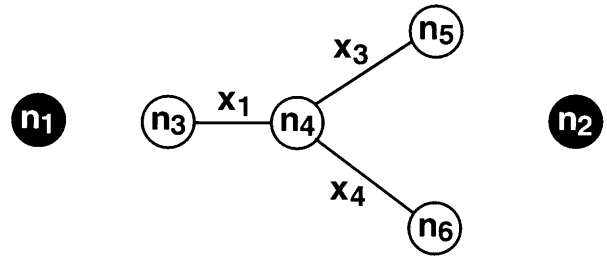
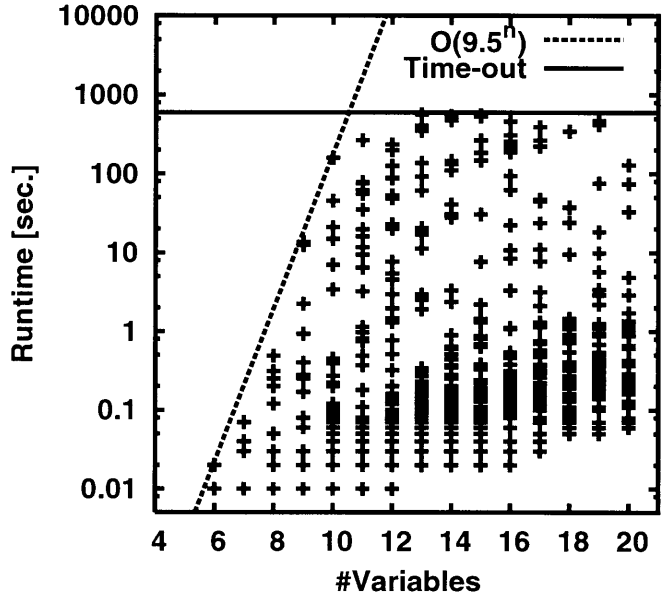


Figure 7.9 An example subgraph such that there does not exist a bridge to implement a prime implicant $x_1x_3x_4x_7x_8x_9$.

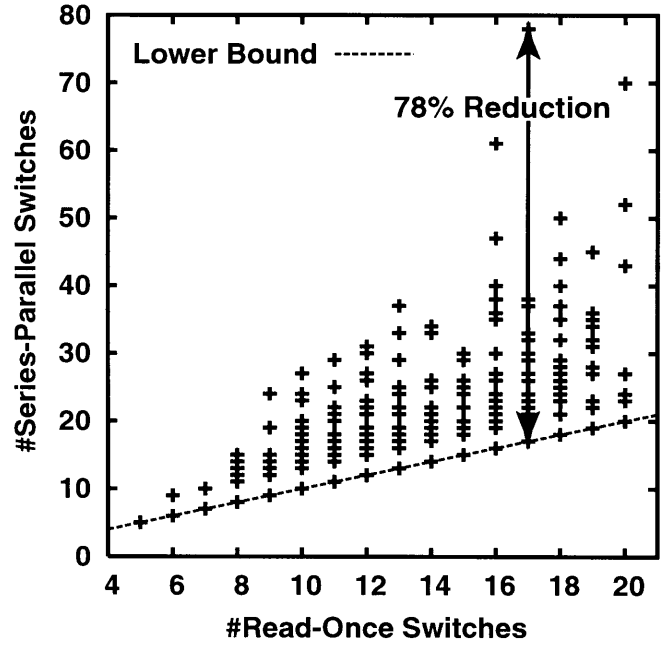
literals in the factored form. Figure 7.10 (b) shows a scatter plot which compares the number of switches in a read-once switch network synthesized by the proposed procedure and the number of switches in a series-parallel switch network synthesized by SIS Good Factor. As can be seen from the table, a read-once switch network can represent the same Boolean function up to 78% smaller number of switches compared to series-parallel switch network.

7.5 Conclusions

Although a switch network is known as one of efficient representation styles of a Boolean function, its synthesis is still a challenging problem. In this chapter, we presented a procedure which efficiently synthesizes a read-once switch network representing a given Boolean function. The experimental results on randomly-generated problems with up to 20 switches demonstrated that the proposed procedure successfully solved about 90% of the problems in 10 minutes each and the resulting read-once switch networks are up to 78% smaller compared to series-parallel switch networks. Future work includes a further development of efficient techniques and an extension to general (*i.e.* non-read-once) switch networks.



(a) Runtime



(b) Comparison of the proposed method and SIS Good Factor.

Figure 7.10 Experimental results on randomly-generated problems.

Chapter 8

Optimal Generation of Design-Specific Cell Libraries: A Case Study

8.1 Introduction

In the preceding chapters, we have studied the components required to realize the flow for optimal generation of design-specific cell libraries which is described in Figure 1.2. In this chapter, we conduct an experimental study on the design-specific cell library generation flow by actually going through the flow using a design example. As a design example, we use a circuit consisting of C432 and C499 from the ISCAS 85 benchmark suite [BF85] The technology used in the case study is an industrial 90nm technology. Finally, we demonstrate that the effectiveness of the proposed approach by comparing against the circuits synthesized with a typical cell library. Several practical issues on the manual generation phase are also discussed.

Before proceeding to the case study, we review the flow for optimal generation of design-specific cell libraries (Figure 8.1) introduced in Chapter 1. Given an initial circuit description and a set of design constraints, an initial circuit is synthesized with a logic-rich cell library generated by the method described in Section 3.2. The logic types are reduced under performance constraints (Section 3.3), and the set of logic types for the final library and the circuit using the library are obtained. Then, the circuit is continuously sized by the optimal transistor sizer (Section 4.2.2) and the total number of cells is minimized under the same performance constraints by minimizing the drive strength count for each logic type. Thus, the optimal design-specific library and the optimized circuit using the library are obtained automatically. If the design requirements are not met at this point, a manual aggressive optimization can be performed as follows. First, a portion of the circuit is identified manually, and then the optimal cell for the portion is synthesized by the transistor-level synthesis methods proposed in Chapter 5, Chapter 6 and Chapter 7. This step is repeated until the requirements are met. Every optimization step in this flow includes the cell characteristics evaluation upon which the

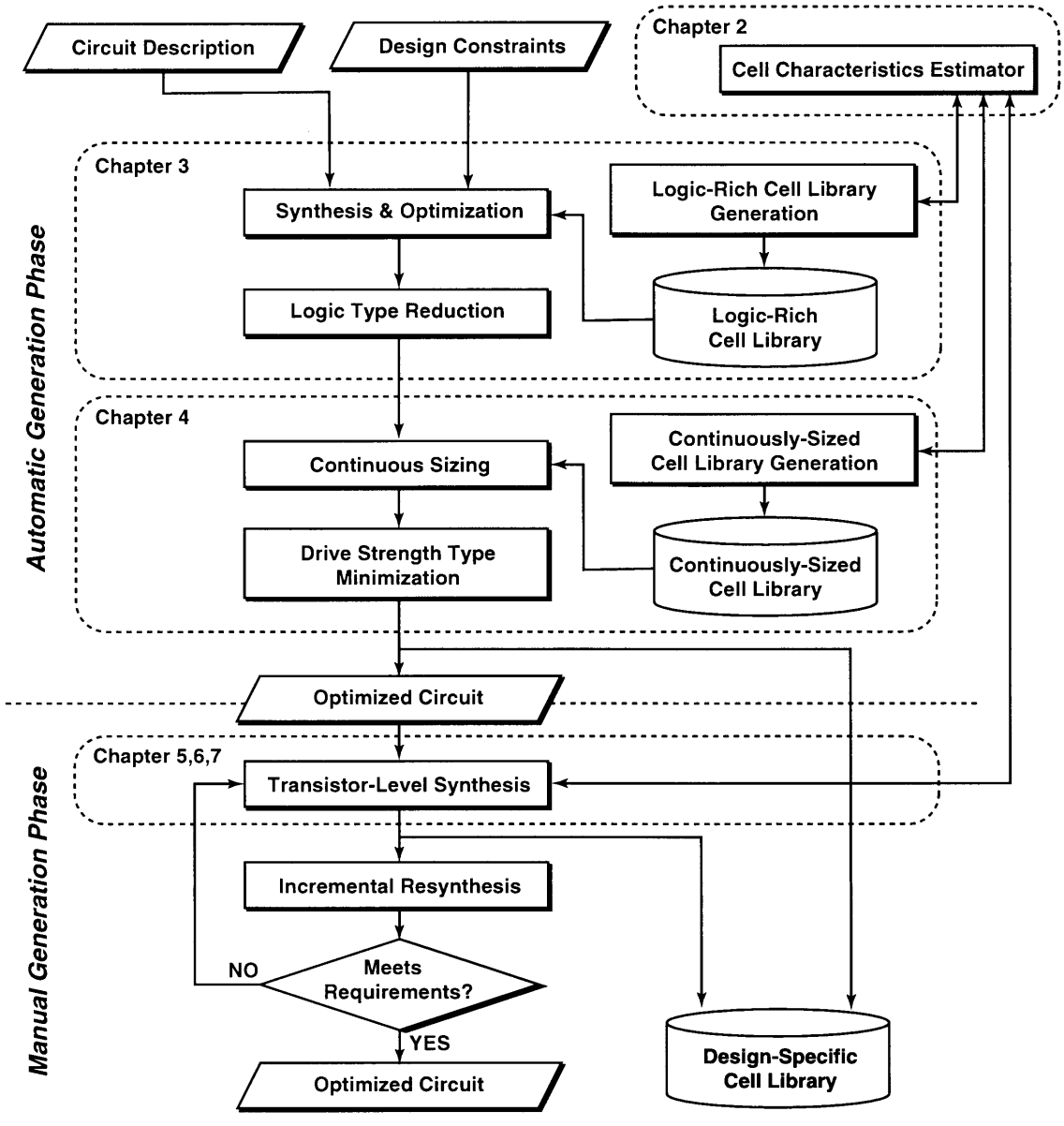


Figure 8.1 Overall flow for optimal generation of design-specific cell libraries.

overall runtime and the final quality depend. The fast and accurate evaluation of cell characteristics is accomplished by the prelayout cell characteristics estimation method in Chapter 2. Thus, all components proposed in this dissertation are combined together in this flow.

The remainder of this chapter is organized as follows. Section 8.2 provides the details of the experimental study on the optimal generation of design-specific cell libraries using a design example. As a reference, we also construct a typical cell library and synthesize the circuits using the library. In Section 8.3, several issues towards a practical application of the manual generation phase are discussed. Conclusions are drawn in Section 8.4.

Table 8.1 Statistics of a typical cell library in an industrial 90nm technology. The number of logic types is 15 and the total number of cells is 50.

Logic Type	*Function*	*Drive Strengths*
INV	$\overline{A+B}$	1x, 2x, 4x, 8x, 16x
NAND2	$\overline{A \cdot B}$	1x, 2x, 4x, 8x
NAND3	$\overline{A \cdot B \cdot C}$	1x, 2x, 4x, 8x
NAND4	$\overline{A \cdot B \cdot C \cdot D}$	1x, 2x, 4x
NOR2	$\overline{A+B}$	1x, 2x, 4x, 8x
NOR3	$\overline{A+B+C}$	1x, 2x, 4x
NOR4	$\overline{A+B+C+D}$	1x, 2x, 4x
AOI21	$\overline{A \cdot B+C}$	1x, 2x, 4x
AOI211	$\overline{A \cdot B+C+D}$	1x, 2x, 4x
AOI22	$\overline{A \cdot B+C \cdot D}$	1x, 2x, 4x
AOI221	$\overline{A \cdot B+C \cdot D+E}$	1x, 2x, 4x
OAI21	$\overline{(A+B) \cdot C}$	1x, 2x, 4x
OAI211	$\overline{(A+B) \cdot C \cdot D}$	1x, 2x, 4x
OAI22	$\overline{(A+B) \cdot (C+D)}$	1x, 2x, 4x
OAI221	$\overline{(A+B) \cdot (C+D) \cdot E}$	1x, 2x, 4x

8.2 Design Example — ISCAS 85 benchmark circuits C432 + C499

As a design example, we use a circuit consisting of C432 and C499 from the ISCAS 85 benchmark suite. C432 is a 27-channel interrupt controller and C499 is a 32-bit SEC circuit [HYH99]. In the circuit, C432 and C499 are placed independently with one another. Throughout this chapter, we used Cadence PKS (Physically-Knowledgeable Synthesis) [PKS04] to synthesize circuits from the register-transfer-level description and optimize the synthesized circuits. We conduct the experiment as follows. First, we construct a typical cell library as a reference using an industrial 90nm technology. Using this library, several circuits are synthesized with different area and delay constraints. Then, we synthesize several circuits by actually going through the design-specific cell library generation flow. The maximum number of cells in every design-specific cell library is limited to 50 which is equivalent to the number of cells in the typical library. Finally, two sets of the synthesized circuits are compared to demonstrate the effectiveness of the proposed flow.

8.2.1 A Typical Cell Library

We constructed a typical cell library in an industrial 90nm technology as follows. The statistics of the library is shown in Table 8.1. In the table, each row corresponds to a logic type. The

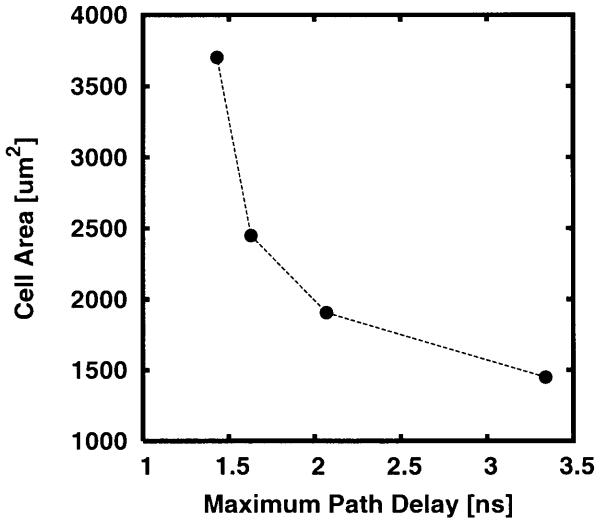


Figure 8.2 Area-delay tradeoff curve on the typical cell library.

second column shows the Boolean function and the third column shows the drive strengths of the cell. The set of logic types and drive strengths is compliant to typical industrial libraries. The transistor sizes are determined by the method described in Section 3.2.2. The number of logic types is 15 and the total number of cells is 50. The cell characteristics were obtained using the prelayout cell characteristic estimator proposed in Chapter 2 with HSPICE [HSP03]. Using this library, the circuits are synthesized under different area and performance constraints. Figure 8.2 shows the area-delay tradeoff curve. In the figure, the fastest point corresponds to the delay-optimal circuit (*i.e.* no area constraint) and the smallest point to the area-optimal circuit (*i.e.* no timing constraint).

8.2.2 Design-Specific Cell Libraries

First, we synthesized the area-optimal circuit using the automatic flow in Figure 8.1. Using the logic-rich library which is constructed in Chapter 3, the area-optimal circuit was synthesized. Table 8.2 shows the statistics of the cell logic types for the area-optimal circuit. The number of logic types is 26. Since all cells have the smallest sizes, the total number of cells is also 26. As can be seen from the table, many complex cells with 6 inputs are used in the area-optimal circuit. This fact reconfirms that complex cells are beneficial for area reduction.

Next, we synthesized the delay-optimal circuit as follows. Using the logic-rich library, an initial delay-optimal circuit was synthesized. Then, the logic types are reduced to 6 types. Table 8.3 shows the statistics of the cell logic types for the delay-optimal circuit. In contrast to that for the area-optimal circuit, the logic types are all simple ones. Besides, two different

Table 8.2 Statistics of cell logic types for the area-optimal circuit.

Cell Function	*#Instances*
$\overline{A+B}$	70
$\overline{A \cdot B + C}$	53
$\overline{A \cdot (B+C)}$	50
$\overline{A}$	36
$\overline{(A+B+C) \cdot D}$	32
$\overline{A \cdot B}$	19
$\overline{A \cdot B \cdot C}$	17
$\overline{(A+B \cdot C) \cdot D}$	8
$\overline{A \cdot B + C + D}$	6
$\overline{A \cdot B + C + D \cdot E}$	5
$\overline{A \cdot B \cdot (C+D)}$	4
$\overline{A \cdot B + C \cdot D}$	3
$\overline{(A+B) \cdot C}$	3
$\overline{A \cdot B + C \cdot D + E \cdot F}$	2
$\overline{A+B+C+D}$	2
$\overline{A \cdot B \cdot (C+D) \cdot (E+F)}$	1
$\overline{A \cdot B \cdot (C+D \cdot E) + F}$	1
$\overline{A \cdot B \cdot C \cdot D}$	1
$\overline{A \cdot B + C + D + E \cdot F}$	1
$\overline{A \cdot (B+C) + D + E}$	1
$\overline{A \cdot B + (C+D) \cdot (E+F)}$	1
$\overline{A \cdot (B+C+D \cdot E)) + F}$	1
$\overline{(A+B) \cdot (C+D)}$	1
$\overline{(A+B \cdot C \cdot D) \cdot E}$	1
$\overline{(A+B+C \cdot D) \cdot E \cdot F}$	1
$\overline{A+B+C}$	1

Table 8.3 Statistics of cell logic types for the delay-optimal circuit.

Cell Function	*#Instances*
$\overline{A}$	263
$\overline{A \cdot B}$	180
$\overline{A+B}$	70
$\overline{(A+B) \cdot C}$	63
$\overline{A \cdot (B+C)}$	47
$\overline{A+B \cdot C}$	46

topologies, $\overline{(A + B) \cdot C}$ and $\overline{A \cdot (B + C)}$, of OAI21 are used. Since these two types have different input-to-output delays, they are used according to the timing criticality of gate inputs. Then, the circuit was continuously and optimally sized, and then the number of cells was minimized to 50 which is equivalent to the number of cells in the typical library. Figure 8.3 (a)-(f) show the cell size distributions of 6 logic types. An important observation from these distributions is that the sizes are almost within the range of the cell sizes in the typical cell library. In other words, the performance improvement is achieved not by using bigger cells than those in the typical cell library, but by using intermediate sizes and beta ratio variations.

Finally, four points between the area-optimal and delay-optimal circuits were generated in a similar way under the constraint that the maximum number of cells is 50. The final area-delay tradeoff curve is superimposed to the tradeoff curve using the typical cell library (Figure 8.2) and shown in Figure 8.4. As can been seen from the figure, the tradeoff curve is shifted to the left-bottom. Comparing between the area-optimal circuits, the area was improved by 27.3%. Also, comparing between the delay-optimal circuits, the maximum delay was improved by 22.4%. This indicates that the circuits are intrinsically improved by using the design-specific cell libraries.

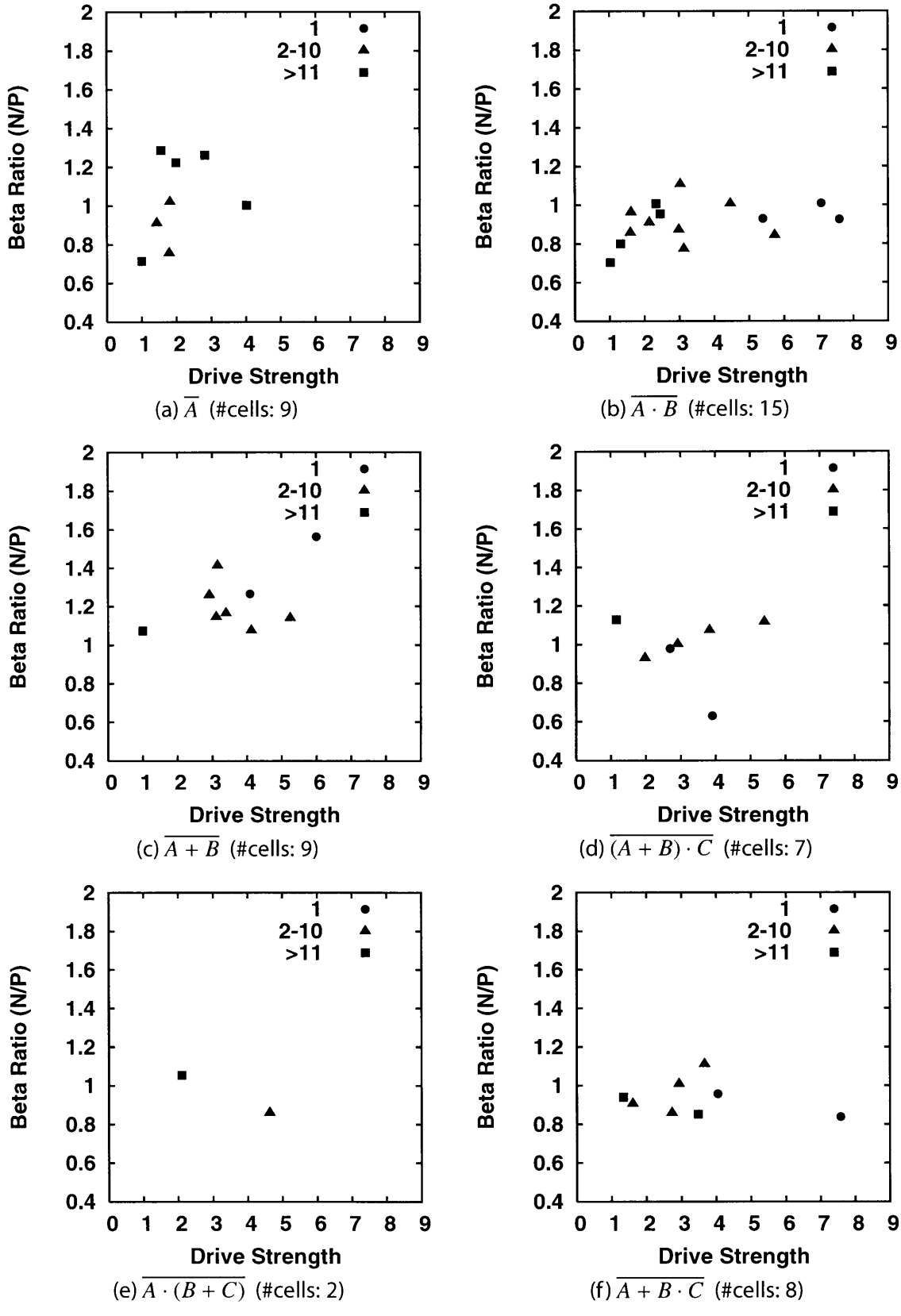
8.3 Discussions on Manual Cell Generation

In this case study, the manual generation of design-specific cells was not performed. Practically, this manual process is infeasible mainly because it is difficult to identify a circuit portion for transistor-level resynthesis. Automating this identification process is the key to a successful realization of the overall flow. Ideally, a circuit portion should be selected such that

1. the maximal area and/or performance improvement is expected by transistor-level resynthesis of the portion, and

2. the resulting cell can be maximally utilized at other portions.

Regarding the item 1, a further study may be necessary to evaluate the area and performance improvement by transistor-level synthesis without performing any actual synthesis. Or, the transistor-level synthesis must be fast enough to explore all possible circuit portions in a feasible runtime. The item 2 can be viewed as a problem to find a substructure appearing frequently in a circuit. This problem has been well studied and a number of efficient algorithms are available. For a fine-grained structural analysis, a target circuit may be decomposed into smallest gates, such as inverters and 2-input NANDs.



(a) $\overline{A}$ (#cells: 9)

(b) $\overline{A \cdot B}$ (#cells: 15)

(c) $\overline{A + B}$ (#cells: 9)

(d) $\overline{(A + B) \cdot C}$ (#cells: 7)

(e) $\overline{A \cdot (B + C)}$ (#cells: 2)

(f) $\overline{A + B \cdot C}$ (#cells: 8)

Figure 8.3 Cell size distributions for the delay-optimal circuit. A circle indicates the number of instances of the cell is 1, a triangle indicates between 2 and 10, and a square indicates more than 10.

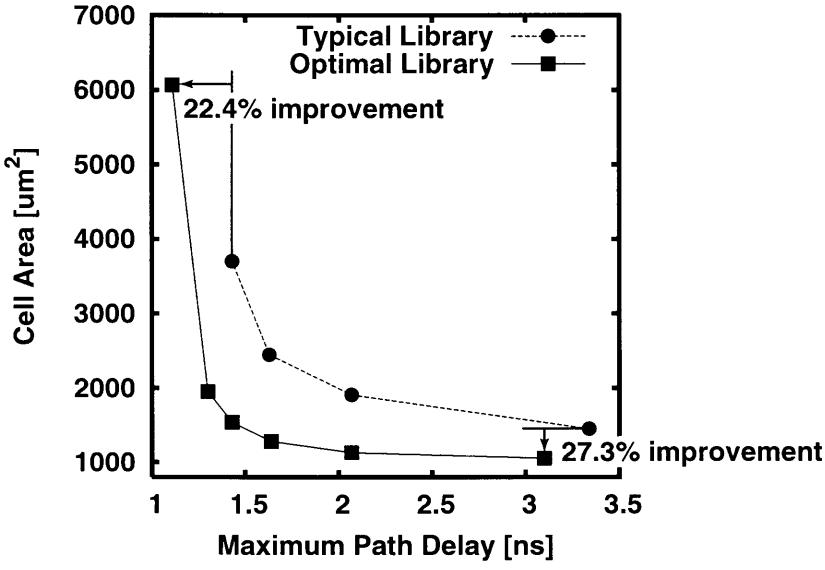


Figure 8.4 Area-delay tradeoff curves on the typical cell library and the optimal design-specific cell library.

8.4 Conclusions

In this chapter, we conducted an experimental study to demonstrate the effectiveness of the flow proposed in this dissertation. As a design example, a circuit consisting of C432 and C499 from the ISCAS 85 benchmark suite was used with an industrial 90nm technology. The experimental results demonstrated that using the design-specific cell libraries, the area-delay tradeoff curve was shifted to the left-bottom from that using a typical cell library. Comparing between the area-optimal circuits, the area was improved by 27.3%. And, comparing between the delay-optimal circuits, the maximum delay was improved by 22.4%. From these results, we draw a conclusion that the proposed flow achieved an intrinsic improvement.

Chapter 9

Conclusions

We have studied various problems regarding optimal generation of design-specific cell libraries. The contribution of this dissertation is summarized below.

The goal of the first part of the dissertation was to provide the key components required to successfully realize the automatic generation phase (the upper half part of the overall flow in Figure 1.2), which consists of the cell logic type selection and the drive strength type selection.

Chapter 2 addressed feasibility issues on transistor-level optimization. During transistor-level optimization, the cell layout synthesis and characterization steps are the major bottlenecks with respect to runtime. To resolve this drawback, we presented a fast and accurate prelayout estimation technique of cell characteristics. Our estimation technique is based on quick transistor placement. Given a transistor-level circuit of a cell, the layout parasitics are estimated using quick transistor placement. Then, the cell is characterized by simulating an estimated circuit which is built according to the estimated layout parasitics. The experimental results on a 0.13μm industrial standard cell library demonstrated that the proposed technique estimated the cell characteristics with a reasonable accuracy in a negligibly small amount of time.

Chapter 3 addressed a cell logic type selection problem for design-specific cell libraries. The methodology consists of two steps: the construction of a logic-rich library and the cell logic type count minimization. The proposed cell logic type count minimization method minimizes the logic type count iteratively under performance constraints. The experimental results on the ISCAS 85 benchmark suite in an industrial 90nm technology demonstrated that it is feasible to find the minimal set of logic types under performance constraints.

Chapter 4 addressed a performance-constrained cell count minimization problem for continuously-sized circuits. After providing a formal formulation of the problem, we proposed an effective heuristic for the problem. The proposed hill-climbing heuristic iteratively

minimizes the number of cells under performance constraints such as area, delay and power. The experimental results on the ISCAS 85 benchmark suite in an industrial 90nm technology demonstrated its effectiveness. We also discussed several implementation issues towards a practical application of the proposed method to large-scale circuits.

The second part of the dissertation focused on transistor-level topology synthesis, which is an important component in the manual generation phase (the lower half portion of the flow in Figure 1.2). We presented three transistor-level topology synthesis methods. Although their objectives are to minimize the transistor count, they have different solution spaces. Combining these methods, the minimum solution in larger solution space can be obtained.

Chapter 5 presented a method for synthesis of minimal static CMOS circuits where the solution space is restricted to the circuit structures which can be obtained by performing algebraic transformations on an arbitrary prime-and-irredundant two-level circuit. The circuit structures are implicitly enumerated via structural transformations on a single graph structure, then a dynamic-programming based algorithm efficiently finds the minimum solution among them. The experimental results on a benchmark suite targeting standard cell implementations demonstrated the feasibility of the proposed procedure. We also demonstrated the efficiency of the proposed algorithm by a numerical analysis on randomly-generated problems. It is also shown that the proposed procedure sometimes generates significantly smaller circuits compared to conventional approach.

Chapter 6 presented an exact method for minimum logic factoring which can be viewed as the synthesis of a static CMOS compound gate. We first introduced a novel graph structure, called an X-B (eXchanger Binary) tree, which implicitly enumerates binary trees. Using this X-B tree, the factoring problem is compactly transformed into a quantified Boolean formula (QBF) and is solved by general-purpose QBF solver. Experimental results on artificially-created benchmark functions showed that the proposed method successfully found the exact minimum solutions to the problems with up to 12 literals.

Chapter 7 studied the synthesis of a read-once switch network in which every variable appears only once. The proposed procedure is based on the notions of prime implicants and unateness, which establish a basis for Boolean expression synthesis. We also proposed a pruning technique for an efficient search. The experimental results on randomly-generated problems with up to 20 switches demonstrated that the proposed procedure successfully solved about 90% of the problems in 10 minutes each and the resulting read-once switch networks are up to 78% smaller compared to series-parallel switch networks.

Chapter 8 conducted an experimental study using a circuit consisting of C432 and C499 from the ISCAS 85 benchmark suite as a design example. We compared the circuits synthesized with a typical cell library and optimal design-specific libraries in an industrial 90nm technology, and demonstrated that using the design-specific cell libraries, the area-delay tradeoff curve was shifted to the left-bottom from that using the typical library. Comparing between the area-optimal circuits, the area was improved by 27.3%. And, comparing between the delay-optimal circuits, the maximum delay was improved by 22.4%. These results clearly proved the effectiveness of the flow and the key components for optimal generation of design-specific cell libraries.

Bibliography

[abr03] *abraCAD Documentation.* Synopsys, Inc., 2003.

[AM90] K. Asada and J. Mavor. MOSYN: A MOS circuit synthesis program employing 3-way decomposition and reduction based on seven-valued logic. *IEE Proc. Computers and Digital Techniques*, 137(6):451–461, November 1990.

[BB02] D. Bhattacharya and V. Boppana. Design optimization with automated flex-cell creation. *Closing the Gap Between ASIC & Custom*, pages 14–23, 2002.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. ACM/IEEE Design Automation Conf.*, pages 193–207, June 1999.

[Ben04] M. Benedetti. sKizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. *ITC-Irst Tech. Rep. TR04-11-03*, November 2004.

[Ber00] D. P. Bertsekas. *Nonlinear Programming 2nd Edition.* Athena Scientific, 2000.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits. In *Proc. IEEE Int. Symp. Circuits and Systems*, pages 695–698, June 1985.

[BF98] J. L. Burns and J. A. Feldman. C5M - a control-logic layout synthesis system for high-performance microprocessors. *IEEE Trans. Computer-Aided Design*, 17(1):14–23, January 1998.

[BHSA03] C. Bittlestone, A. M. Hill, V. Singhal, and NV Arvind. Architecting ASIC libraries and flows in nanometer era. In *Proc. ACM/IEEE Design Automation Conf.*, pages 776–781, June 2003.

[Bie05] A. Biere. Resolve and expand. In *Proc. Intl. Conf. Theory and Applications of Satisfiability Testing, LNCS, Springer*, 2005.

[BKKS98] F. Beeftink, P. Kudva, D. Kung, and L. Stok. Gate size selection for standard cell libraries. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 545–550, November 1998.

[BNNSV97] P. Buch, A. Narayan, A. R. Newton, and A. Sangiovanni-Vincentelli. Logic synthesis for large pass transistor circuits. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 663–670, November 1997.

[Boe88] M. Boehner. LOGEX — an automatic logic extractor from transistor to gate level for CMOS technology. In *Proc. ACM/IEEE Design Automation Conf.*, pages 517–522, July 1988.

[Bra93] D. Brand. Verification of large synthesized designs. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 534–537, November 1993.

[Bre77] M. A. Breuer. A class of min-cut placement algorithms. In *Proc. ACM/IEEE Design Automation Conf.*, pages 284–290, June 1977.

[BRSVW87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. Computer-Aided Design*, 6(6):1062–1081, November 1987.

[Bry86] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, August 1986.

[Bry91] R. E. Bryant. Extraction of gate level models from transistor circuits by four-valued symbolic analysis. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 350–353, November 1991.

[BSVHM84] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and C. McMullin. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, 1984.

[Cal04] *Performing Transistor-level Parasitic Extraction*. Mentor Graphics Corporation, 2004.

[CCW99] C. P. Chen, C. C. N. Chu, and D. F. Wong. Fast and exact simultaneous gate and wire sizing by lagrangian relaxation. *IEEE Trans. Computer-Aided Design*, 18(7):1014–1025, July 1999.

[CEWWM+99] A. R. Conn, I. M. Elfadel, Jr. W. W. Molzen, P. R. O'Brien, P. N. Strenski, C. Visweswariah, and C. B. Whan. Gradient-based optimization of custom circuits using a static-timing formulation. In *Proc. ACM/IEEE Design Automation Conf.*, pages 452–459, June 1999.

[CK00] D. G. Chinnery and K. Keutzer. Closing the gap between ASIC and custom: an ASIC perspective. In *Proc. ACM/IEEE Design Automation Conf.*, pages 637–642, June 2000.

[CK02] D. G. Chinnery and K. Keutzer. *Closing the Gap Between ASIC & Custom: Tools and Techniques for High-Performance ASIC Design.* Kluwer Academic Publishers, 2002.

[CMC+01] R. Carragher, R. Murgai, S. Chakraborty, M. R. Prasad, A. Srivastava, N. Vemuri, H. Yoshida, T. Shibuya, and Y. Kanazawa. Layout-driven logic optimization. In *Designer's Forum Proc. IEEE Design, Automation and Test in Europe*, March 2001.

[CNK01] D. G. Chinnery, B. Nikolic, and K. Keutzer. Achieving 550MHz in an ASIC methodology. In *Proc. ACM/IEEE Design Automation Conf.*, pages 420–425, June 2001.

[Dav69] E. Davidson. An algorithm for NAND decomposition under network constraints. *IEEE Trans. Computers*, 18:1098–1109, 1969.

[DG98] R. Drechsler and W. Gunther. Exact circuit synthesis. In *IEEE Int. Workshop on Logic Synthesis*, 1998.

[DGR+87] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 116–119, November 1987.

[Eck80] J. G. Ecker. Geometric programming: Methods, computations and applications. *SIAM Review*, 22(3):338–362, July 1980.

[FB74] R. A. Finkel and J. L. Bentley. Quad trees, a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[FD85] J. P. Fishburn and A. E. Dunlop. Tilos: A posynomial programming approach to transistor sizing. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 326–328, November 1985.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. ACM/IEEE Design Automation Conf.*, pages 175–181, June 1982.

[FMS00] R. Feldmann, B. Monien, and S. Schamberger. A distributed algorithm to evaluate quantified Boolean formulas. In *Proc. National Conf. on Artificial Intelligence*, pages 285–290, 2000.

[GG98] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[GGP+97] S. Gavrilov, A. Glebov, S. Pullela, S. Moore, A. Dharchoudhury, R. Panda, G. Vijayan, and D. Blaauw. Library-less synthesis for static CMOS combinational logic circuits. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 658–662, November 1997.

[GH97] A. Gupta and J. Hayes. CLIP: An optimizing layout generator for two-dimensional CMOS cells. In *Proc. ACM/IEEE Design Automation Conf.*, pages 452–457, June 1997.

[GMD+97] M. Guruswamy, R. L. Maziasz, D. Dulitz, S. Raman, V. Chiluvuri, A. Fernandez, and L. G. Jones. CELLERITY: A fully automatic layout synthesis system for standard cell libraries. In *Proc. ACM/IEEE Design Automation Conf.*, pages 327–332, June 1997.

[GS96] B. Guan and C. Sechen. Large standard cell libraries and their impact on layout area and circuit performance. In *Proc. IEEE Int. Conf. on Computer Design*, pages 378–383, October 1996.

[HSP03] *HSPICE Data Sheet*. Synopsys, Inc., 2003.

[HYH99] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design and Test*, 16(3):72–80, July 1999.

[IMK+00] R. Inanami, S. Magoshi, S. Kousai, M. Hamada, T. Takayanagi, K. Sugihara, K. Okumura, and T. Kuroda. Throughput enhancement strategy of maskless electron beam direct writing for logic device. In *IEEE Int. Electron Devices Meeting Technical Digest*, pages 833–836, December 2000.

[KKL87] K. Keutzer, K. Kolwicz, and M. Lega. Impact of library size on the quality of automated synthesis. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 120–123, November 1987.

[KKS00] M. Ketkar, K. Kasmasetty, and S. S. Sapatnekar. Convex delay models for transistor sizing. In *Proc. ACM/IEEE Design Automation Conf.*, pages 655–660, June 2000.

[KP99] D. S. Kung and R. Puri. Optimal P/N width ratio selection for standard cell libraries. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 178–184, November 1999.

[KR92] A. B. Kahng and G. Robins. A new class of iterative steiner tree heuristics with good performance. *IEEE Trans. Computer-Aided Design*, 11(7):893–902, July 1992.

[KV02] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms.* Springer-Verlag, 2002.

[L+98] W. Liu et al. *BSIM3v3.2 MOSFET Model Users' Manual.* University of California, Berkeley, 1998.

[LA99] C.-P. L. Liu and J. A. Abraham. Transistor level synthesis for static CMOS combinational circuits. In *Proc. Great Lakes Symposium on VLSI*, pages 116–119, March 1999.

[Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. Computer-Aided Design*, 11(1):4–15, January 1992.

[Law64] E. L. Lawler. An approach to multilevel Boolean minimization. *J. ACM*, pages 283–295, 1964.

[Let02] R. Letz. Lemma and model caching in decision procedures for quantified Boolean formulas. In *Proc. TABLEAUX2002, vol. 2381 of LNAI*, pages 160–175, 2002.

[Lib03] *Library Compiler User Guide: Modeling Timing and Power Technology Libraries.* Synopsys, Inc., 2003.

[LWGH97] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. *IEEE Trans. Computer-Aided Design*, 16(8):813–834, August 1997.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. ACM/IEEE Design Automation Conf.*, pages 530–535, June 2001.

[NL01] G. A. Northrop and P.-F. Lu. A semi-custom design flow in high-performance microprocessor design. In *Proc. ACM/IEEE Design Automation Conf.*, pages 426–431, June 2001.

[PDE+98] R. Panda, A. Dharchoudhury, T. Edwards, J. Norton, and D. Blaauw. Migration: A new technique to improve synthesized designs through incremental customization. In *Proc. ACM/IEEE Design Automation Conf.*, pages 388–391, June 1998.

[Pfe79] H. C. Pfeiffer. Recent advances in electron-beam lithography for the high-volume production of VLSI devices. *IEEE Trans. Electron Devices*, 4:663–674, 1979.

[PKS04] *PKS User Guide.* Cadence Design Systems, 2004.

[Pro80] A. Proskurowski. On the generation of binary trees. *J. ACM*, 27(1):1–2, January 1980.

[PS88] C. Pedron and A. Stauffer. Analysis and synthesis of combinational pass transistor circuits. *IEEE Trans. Computer-Aided Design*, 7(7):775–786, July 1988.

[Qui52] W. Quine. The problem of simplifying truth functions. *American Math. Monthly*, 59:521–531, 1952.

[RC05] S. Roy and W. Chen. ConvexFit: an optimal minimum-error convex fitting and smoothing algorithm with application to gate-sizing. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 196–203, November 2005.

[RC06] S. Roy and C. C.-P. Chen. ConvexSmooth: A simultaneous convex fitting and smoothing algorithm for convex optimization problems. In *Proc. IEEE Int. Symp. on Quality of Electronic Design*, pages 665–670, March 2006.

[RHH+02] N. Richardson, L. B. Huang, R. Hossain, J. Lewis, T. Zounes, and N. Soni. The iCORE^{TM} 520 mhz synthesizable CPU core. *Closing the Gap Between ASIC & Custom, Kluwer Academic Publishers*, pages 225–240, 2002.

[RPS01] S. E. Rich, M. J. Parker, and J. Schwartz. Reducing the frequency gap between ASIC and custom designs: A custom perspective. In *Proc. ACM/IEEE Design Automation Conf.*, pages 432–437, June 2001.

[RS03] M. A. Riepe and K. A. Sakallah. Transistor placement for noncomplementary digital VLSI cell synthesis. *ACM Trans. Design Automation of Electronic Systems*, 8(1):81–107, January 2003.

[Sem04] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors 2004 Update*. 2004.

[SID+99] N. Shenoy, M. Iyer, R. Damiano, K. Harer, H.-K. Ma, and P. Thilking. A robust solution to the timing convergence problem in high-performance design. In *Proc. IEEE Int. Conf. on Computer Design*, pages 250–257, October 1999.

[SK94] K. Scott and K. Keutzer. Improving cell libraries for synthesis. In *Proc. IEEE Custom Integrated Circuits Conf.*, pages 128–131, May 1994.

[SRVK93] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S.-M. Kang. An exact solution to the transistor sizing problem for CMOS circuits using convex optimization. *IEEE Trans. Computer-Aided Design*, 12(11):1621–1634, November 1993.

[SS97] J. P. M. Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 220–227, November 1997.

[SS99] T. Serdar and C. Sechen. AKORD: Transistor level and mixed transistor/gate level placement tool for digital datapaths. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 91–97, November 1999.

[TS05] H. Tennakoon and C. Sechen. Efficient and accurate gate sizing with piecewise convex delay models. In *Proc. ACM/IEEE Design Automation Conf.*, pages 807–812, June 2005.

[WB06] A. Wachter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[Yan91] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. *Tech. rep., Microelectronics Center of North Carolina*, January 1991.

[YDB04] H. Yoshida, K. De, and V. Boppana. Accurate pre-layout estimation of standard cell characteristics. In *Proc. ACM/IEEE Design Automation Conf.*, pages 208–211, June 2004.

[YSS96] K. Yano, Y. Sasaki, and K. Seki. Top-down pass-transistor logic design. *IEEE J. Solid-State Circuits*, 31(6):792–803, June 1996.

[Zak80] S. Zaks. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 10:63–82, 1980.

List of Publications

Journal Papers

[1] **H. Yoshida**, M. Ikeda and K. Asada, "Exact Minimum Logic Factoring via Quantified Boolean Satisfiability," *IEICE Transactions on Fundamentals. (to be submitted)*

[2] **H. Yoshida**, M. Ikeda and K. Asada, "Cell Characteristics Estimation Using Quick Transistor Placement," *IEICE Transactions on Fundamentals. (to be submitted)*

[3] **H. Yoshida**, M. Ikeda and K. Asada, "Synthesis of Read-Once Switch Networks," *IEICE Transactions on Fundamentals. (submitted)*

[4] **H. Yoshida**, M. Ikeda and K. Asada, "A Structural Approach for Transistor Circuit Synthesis," *IEICE Transactions on Fundamentals*, vol. E89-A, no.12, pp. 3529-3537, Dec. 2006.

International Conference Papers

[1] **H. Yoshida**, M. Ikeda and K. Asada, "Performance-Constrained Cell Count Minimization for Continuously-Sized Circuits," *ACM/IEEE Design Automation Conference*, Jun. 2007. (*submitted*)

[2] **H. Yoshida**, M. Ikeda and K. Asada, "Synthesis of Read-Once Switch Network," *ACM Great Lakes Symposium on VLSI*, Mar. 2007. (*submitted*)

[3] **H. Yoshida**, M. Ikeda and K. Asada, "Exact Minimum Logic Factoring via Quantified Boolean Satisfiability," in *Proc. IEEE International Conference on Electronics, Circuits and Systems*, Dec. 2006.

[4] **H. Yoshida**, M. Ikeda and K. Asada, "An Algebraic Approach for Transistor Circuit Synthesis," in *Proc. IEEE International Conference on Electronics, Circuits and Systems*, Dec. 2005.

[5] **H. Yoshida**, K. De, and V. Boppana, "Accurate Pre-layout Estimation of Standard Cell Characteristics," in *Proc. of ACM/IEEE Design Automation Conference*, pp. 208–211, Jun. 2004.

Domestic Conference Papers

[1] **H. Yoshida**, M. Ikeda, and K. Asada, "Synthesis of Read-Once Switch Network," in *Proc. of IEICE Society Conference 2006*, A-3-9, pp. 53, Sep. 2006. (*in Japanese*)

[2] **H. Yoshida**, M. Ikeda, and K. Asada, "Exact Minimum Logic Factoring via Quantified Boolean Satisfiability," *IEICE Technical Report*, vol. 105, no. 443, pp. 41–46, Dec. 2005. (*in Japanese*)

[3] **H. Yoshida**, M. Ikeda, and K. Asada, "An Algebraic Approach for Synthesizing Circuits with Minimum Number of Transistors," in *Proc. of IPSJ DA Symposium 2005*, pp. 133–138, Aug. 2005. (*in Japanese*)

[4] **H. Yoshida**, K. De, V. Boppana, M. Ikeda, and K. Asada, "Accurate Pre-Layout Estimation of Intra-cell Parasitics Using Fast Transistor-level Placement," *IEICE Technical Report*, vol. 104, no. 478, pp. 7–12, Dec. 2004. (*in Japanese*)

Patents

[1] **H. Yoshida** and V. Boppana, System and method for automated accurate pre-layout estimation of standard cell characteristics, U.S. Patent Application 20050229142.

[2] P. Majumder, B. Kumthekar, N. R. Shah, J. Mowchenko, P. A. Chavda, Y. Kojima, Y. Jiang, **H. Yoshida**, and V. Boppana. Method, system and apparatus of IC design optimization via creation of design-specific cell from post-layout patterns, U.S Patent Application Serial No. 60/809,132.

Other Publications/Awards

[1] U. Ekinciel, H. Yamaoka, **H. Yoshida**, M. Ikeda, and K. Asada, "A Performance Driven Module Generator for a Dual-Rail PLA with Embedded 2-Input Logic Cells," *IEICE Transactions on Information & Systems*, vol. E88-D, no. 6, pp. 1159-1167, Jun. 2005.

[2] H. Yamaoka, **H. Yoshida**, M. Ikeda and K. Asada, "A Logic-Cell-Embedded PLA (LCPLA): An Area-Efficient Dual-Rail Array Logic Architecture," *IEICE Transactions on Electronics*, vol. E87-C, no.2, pp.238-245, Feb. 2004.

[3] H. Yamaoka, **H. Yoshida**, M. Ikeda and K. Asada, "A Dual-Rail PLA with 2-Input Logic Cells," in *Proc. IEEE European Solid-State Circuits Conference*, pp. 203-206, Sep. 2002.

[4] **H.Yoshida**, H. Yamaoka, M. Ikeda, and K. Asada, "Logic Synthesis for PLA with 2-input Logic Elements," in *Proc. IEEE International Symposium on Circuits and Systems*, May 2002.

[5] **H. Yoshida**, M. Sera, M. Kubo and M. Fujita "Simultaneous Circuit Transformation and Routing," in *Proc. Asia and South Pacific Design Automation Conference and International Conference on VLSI Design*, pp. 479–483, Jan. 2002.

[6] **H. Yoshida**, H. Yamaoka, M. Ikeda, and K. Asada, "Logic Synthesis for AND-XOR-OR type Sense-Amplifying PLA," in *Proc. Asia and South Pacific Design Automation Conference and International Conference on VLSI Design*, pp. 166–171, Jan. 2002.

[7] **H. Yoshida**, M. Sera, M. Kubo and M. Fujita, "Integration of Logic Synthesis and Layout processes by Generating Multiple Choices of Circuit Transformation," *IEEE International Workshop on Logic and Synthesis*, Jun. 2001.

[8] R. Carragher, R. Murgai, S. Chakraborty, M. R. Prasad, A. Srivastava, N. Vemuri, **H. Yoshida**, T. Shibuya and Y. Kanazawa, "Layout-driven Logic Optimization," in *Proc. IEEE Design, Automation and Test in Europe*, Mar. 2001.

[9] K. Seto, **H. Yoshida**, M. Ikeda and K. Asada, "Logic Minimization Using Node Complementation," *IEEE International Workshop on Logic Synthesis*, Jun. 2000.

[10] **H. Yosida**, H. Yamaoka, M. Ikeda, and K. Asada, "Logic Synthesis for PLA with 2-input Logic Elements," *IEICE Technical Report*, CPSY2001-72, Nov. 2001. (*in Japanese*)

[11] **H. Yoshida**, H. Yamaoka, M. Ikeda, and K. Asada, "Logic Synthesis for XOR-Based Dual=Rail PLA," in *Proc. of IPSJ DA Synposium*, pp. 31-36, Jul. 2001. (*in Japanese*)

[12] H. Yamaoka, **H. Yoshida**, U. Ekinciel and K. Asada, "A Module Generator for a Dual-Rail PLA with 2-input Logic Cells," *5th Nikkei-BP LSI IP Design Award (IP Award)*, Jun. 2003.

[13] H. Yamaoka, **H. Yoshida**, U. Ekinciel and K. Asada, "A Module Generator for a Dual-Rail PLA with 2-input Logic Cells," *4th Nikkei-BP LSI IP Design Award (Challenge Award)*, May. 2002.