PERFORMANCE OPTIMIZATION OF DIGITAL CIRCUITS

by

Kanwar Jit Singh

Memorandum No. UCB/ERL M92/149

15 December 1992

PERFORMANCE OPTIMIZATION OF DIGITAL CIRCUITS

by

Kanwar Jit Singh

Memorandum No. UCB/ERL M92/149

15 December 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Performance Optimization of Digital Circuits

Kanwar Jit Singh

University of California
Berkeley, California

Department of Electrical Engineering
and Computer Sciences

Abstract

The design of complex, high performance systems requires automation of the design process. At the outset of automation, the focus of synthesis was to obtain small realizations that could be implemented as integrated circuits. However, with improvements in fabrication technology, the area of a design is of secondary importance and the performance of the circuit is the primary criterion that the designer wishes to maximize. Circuit performance is affected by various decisions taken during different phases of the design process.

This thesis provides an understanding of the factors that affect the delay of logic circuits. Logic circuits are represented as an interconnection of functional units and registers. To improve the speed of an initial implementation, the designer may apply different types of optimizations. Using a set of bounded-input functions, functional units can be implemented by circuit structures that have small depth by repeated application of local transformations on the circuit. This phase is referred to as *technology-independent* optimization. Reduction in circuit depth usually leads to smaller delay. However, circuit speed also depends on the choice of gates used to implement the function. Gates differ in their delay characteristics. To utilize this flexibility *technology-dependent* optimizations are required. The paradigm of applying local transformations is extended to exploit the delay characteristics of gates during optimization. In circuits that contain registers, the freedom in positioning registers allows for further *synchronous* optimizations.

Prof. Alberto Sangiovanni-Vincentelli
Thesis Committee Chairman

Performance Optimization of Digital Circuits

Kanwar Jit Singh

Acknowledgements

When I started graduate work, oh some six years ago, I was floundering through classes trying to decide what area to work in. Within an year I had it narrowed the possibilities to communication and computer-aided-design. It was the excellent environment of the CAD-group and guidance from faculty that led me to join the CAD-group.

Graduate work at Berkeley been an interesting and very fruitful experience primarily due to my association with Professors Sangiovanni and Brayton. I am greatly indebted to Prof. Sangiovanni-Vincentelli, my research advisor, for the support and direction that he has provided. By giving me flexibility in deciding how my research would develop he helped me learn the importance of a systematic approach to defining and solving problems. I have also benefited immensely from his excellent comments on writing and presentation style. Prof. Brayton has been a second advisor to me. His probing and insightful questions helped me foresee some of the problems and pitfalls during the early stages of my research. I learnt a lot from the courses on logic synthesis and circuit analysis that he taught.

Professors Jan Rabaey and Dorit Hochbaum agreed to sit on my qualifying examination and helped me focus my research. I would like to thank Prof. Hochbaum for accepting to be a member of my thesis committee.

My friends in the department, from those currently here to those who are out of graduate school and raking in big money, have been very instrumental in making graduate school a pleasant experience. All my current colleagues, especially Ellen Sentovich, Rajeev Murgai, Alexander Saldanha, Luciano Lavagno and Narendra Shenoy have been very helpful particularly during the last few months of thesis writing. Interactions with Rick McGeer and Sharad Malik on issues of timing optimization were very productive and I thank them for taking the time to discuss issues with me. Many thanks are are also due to Albert Wang, Richard Rudell, Peter Moore, Rick Spickelmier, Andrea Casotto the Toms — Quarles and Laidig — and to David Harrison for creating the software infrastructure that supports the research in the group. If I have failed to recognize the support of my other friends in the group, and I am sure I must have, it is only due to my haste in writing this section.

Life outside the department has been considerable fun — thanks to the wonderful California weather, the great facilities on campus and above all my friends. I would like to acknowledge the support of my fellow Indian graduate students who provided a strong support network. Abjhijit Sahay, Tarun Verma, Rajeev Motwani, Ananth Jhingran, Huzur

Saran, Vedant Sampat, Arun Majumdar, G. Srini and Sharad Malik were graduate students when I joined Berkeley. They took me under their wing and made the transition to graduate school very easy. Jayavardhan Pandit, Pratap Khedkar, Madhu Sudan, Savita and Usha Sahay, Diane Bailey, Sushil Verma, Sajina and Sushil Warrier have been very good friends, always willing to help and party. Special thanks are due to my roommates over the years — Narendra Shenoy and Arunava Majumdar — who have put up with my late night work schedules and watching ESPN. My field-hockey friends and street-hockey buddies have a been the source of many lasting friendships and I wish them well.

Last, but definitely the most important, is the support and love of my family. I was greatly influenced by my grandfathers — Sarain Singh and Balwant Singh Nag — who were role models to me as I was growing up. My sisters have been very supportive of all my decisions and I wish them and their families well. The unlimited sacrifices that my parents underwent to assure that I got a good education is a debt that I cannot repay. Mom and Dad, this thesis is dedicated to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of Integrated Circuits (IC's), system manufacturers can pack a lot of functionality on a single chip. With present day synthesis techniques system designers feel quite confident that the desired functionality can be effectively implemented in a small area. But the same level of confidence is not seen in predictions about meeting the target performance. The reason for this lack of confidence is not hard to trace. For many years now, research on the automated synthesis of circuits has focussed primarily on reducing the area of the design and being able to test them. This was a valid objective when the number of transistors that could be packed into a single chip was limited. In addition, a primary objective of synthesis was to improve the design time and to ensure a correct implementation of the design. As the technology for area-based synthesis has matured, designers are looking to ways of squeezing out the last bit of performance from automatically synthesisized designs to get an edge over their competitors.

In most design environments, designers synthesize an initial circuit implementing the desired circuit functionality. Timing analysis techniques then predict the critical paths in the circuit. These paths determine the speed of operation. If the critical paths do not meet the performance specifications, then local changes are made manually to improve circuit speed. This process is time consuming and typically is the bottleneck in the design process. It is therefore important to find methods that will guarantee (with a large degree of confidence) that synthesized circuits will meet the performance objectives.

For a design, an important metric of performance is the clocking speed. This reflects the rate at which data can be presented to the system. Of course, in devices such as microprocessors, the clocking speed has to be considered with caution. A small clocking period is meaningless if only very simple operations are performed during a cycle. The true measure of performance in these systems is the clock period multiplied by the number of clock cycles it takes to accomplish the task. In digital design, it is clear that the system architecture is an important issue in performance optimization. For example, by organizing the tasks into simpler ones (pipelining the architecture), a designer may achieve the desired performance. However, once the architecture has been fixed, the only option available to the designer is to optimize the logic so that the clock speed can be increased.

To put this research in perspective, it is useful to look at the overall design process and the types of decisions made at each step to improve circuit performance.

1.1 The design process

The performance of a design is influenced by the design style used and the steps that constitute the synthesis procedure. Synthesis is typically done in three stages — high-level synthesis, logic synthesis and physical synthesis. In this research the focus has been on understanding the factors that determine the delay of a circuit during the logic synthesis stage. However, to provide a complete picture of performance optimization, the technology considerations and the individual stages of synthesis are described briefly.

1.1.1 Technology considerations

Different design styles, such as full-custom, standard-cell based, gate-array based and more recently programmable-device based designs, are used by different designers. The design style affects the choice of optimization techniques and the metrics used to evaluate the design during optimization. Full-custom designs allow freedom to modify the circuit at the transistor level and consequently one tool that is crucial to generating high-speed circuits is transistor sizing [19]. Transistor sizing allows a very precise control of the circuit delay and assumes the availability of parasitic capacitances and transistor models. The domain of full-custom circuits consists of the fastest designs made using considerable design time and expertise. Programmable circuits use a repetitive array of fixed devices and are ideal for rapid prototyping of the design. However, due to the fixed architecture of such devices, there is only a coarse control of circuit delay. In addition, the overheads that go along with providing programmability makes these devices slow. Cell-based designs provide a compromise where a fixed set of custom-designed gates are used in a regular layout style (standard-cell or gate-array). These form an important part of the semiconductor market since they allow good control over the circuit performance and can be designed with relative ease. This methodology is used widely in the design of application specific integrated circuits (ASIC's). The problems being addressed in this research are primarily from the cell-based design style although some aspects are applicable to all three design styles.

1.1.2 High-level synthesis

High-level synthesis is the process of transforming a behavioral description of a design into a more concrete functional description. The behavioral specification is typically in the form of a Hardware Description Language, e.g. VHDL, Verilog, Ella, Hardware-C, etc., and consists of a program-like description of the design. The specification is transformed into an interconnection of functional units, e.g. adders, comparators, multipliers, etc., and an associated controller. The allocation of functional units and scheduling of operations are directed to satisfy constraints specified by the user. These constraints may be area related (the design should not use more than 2 adders and 1 multiplier) or performance related (the operations should be scheduled in no more than 5 cycles). These decisions affect the architecture of the design and have a significant effect on the system performance. In addition to scheduling and allocation, considerations such as pipelining, determining the granularity of the operations are also the domain of high-level synthesis.

In this research we assume that the architecture of the system and the specification of the functional description is fixed. This assumes a "top-down" design style where all the high-level decisions are made prior to logic synthesis. Once the high-level decisions have been made the implementation of each logic module must satisfy the functional and timing specifications generated for it.

1.1.3 Logic synthesis

Transforming the functional description of the system into an interconnection of gates is done during the logic synthesis stage. The types of optimizations made during this stage are strongly dependent on the technology and the design style used to implement the circuit. For example, the optimizations performed for cell-based designs differ from the optimizations that are required for designs that use programmable devices. The cell-based design methodology is becoming increasingly important as the need to market fast circuits in a short period of time increases. Short design time is a result of algorithms that allow the automatic synthesis and verification of the circuits. The fully-automated or "push-button" synthesis scenario has been very successful in reducing the size of the logic implementation so that designs with small area are generated routinely. During area minimization there is little consideration of circuit speed so that area minimal designs often fail to meet the timing specifications. As a result, a large part of the design time is spent on fixing problems related to timing violations. This research is directed towards alleviating the problem of circuit redesign by considering the timing issues during the synthesis of the logic. If accepted, this approach can reduce the need to undertake manual correction of the circuit to meet performance constraints.

1.1.4 Physical synthesis

Once the netlist of the circuit has been designed the next important task is that of realizing the circuit on silicon. The components need to be placed (the placement problem) and interconnected (the routing problem). The choices made at this level of design also affect circuit performance. As an example, a poorly placed design may place two connected gates far apart which leads to longer wires connecting them and an increased capacitive loading that may degrade the circuit performance. There are two broad approaches to avoid this. The first is to impose constraints on the lengths of wires [47] and pass them on as an input to the placement routines. By meeting these constraints, the physical design is guaranteed to meet the projected performance. The second approach uses mathematical programming techniques to reduce the critical path delays by controlling the placement [28, 21].

Results quoted in literature suggest that the higher the level of abstraction the larger is the possibility of a dramatic increase in system performance. Thus architectural decisions would impact the performance the most while optimizations at the physical design stage have a limited scope for improving the performance. However, the further one gets into the design process the more crucial become the optimizations since a failure to meet the performance targets results in a redesign of all the earlier stages — a very time consuming process. The logic level provides a good compromise for attempting the performance optimizations — it is early enough in the design process to allow changes to be made to the behavioral models yet there is enough information to be able to predict system performance after physical design.

1.2 Assumptions and abstractions

As is clear from the previous section, the delay of a circuit is affected by many factors. It is thus essential that when focusing on some aspect of the design process, e.g. logic synthesis, assumptions be made about what decisions have been made and also be able to abstract and quantify the effect of decisions yet to be made.

We assume that the following decisions have been made before the logic synthesis process is initiated.

1. The design style and the target technology are fixed.
2. The architecture of the circuit is fixed. What we mean here is that all architectural decisions, such as whether to implement a multiplication as a loop of partial additions or as a combinational circuit, have been made.
3. The clocking discipline is fixed.

To model circuit delays we use the following abstractions.

1. There exists a delay model that will predict the delay of the circuit. The accuracy of the delay prediction increases as the circuit approaches its final form.
2. The effect of layout on the delay can be estimated by including an additional delay based on the number of gates that are driven by a gate. This estimation is pessimistic in that the predicted delay is an upper bound on the circuit delay.
3. The delay of a circuit is determined by the longest path in the circuit. This assumption is pessimistic in that it overestimates the circuit delay [27]. This overestimation may lead to predicting a circuit speed that is slower than the actual one.

Optimizations performed during the logic synthesis phase affect the circuit area, the delay and the testability of the circuit. These objectives are not independent — optimizing one may affect the others. The work in [53] provides an excellent account of the interaction of the three optimization criteria. The focus of this research is to develop techniques to reduce the circuit delay. This often entails degradation of the circuit area and testability. Fortunately techniques exist to ensure testability without degrading circuit performance. As a result, it is acceptable to reduce circuit delay without regard to testability considerations since 100% testability can be guaranteed by appropriate post-processing. As regards to circuit area, an attempt will be made to keep the area increase small while trying to reduce the delay.

1.3 Thesis overview

At the logic level, the task of optimizing the performance of a circuit is partitioned, to simplify the exposition and reduce the complexity of the problem, into several subproblems. One partition can be made based on the presence or absence of feedback in the circuit. Acyclic circuits are called *combinational circuits* while those with feedback are called *sequential.* The sequential circuits that we consider fall into the category of *synchronous* sequential circuits since each cycle is required to have at least one "memory element". Another problem decomposition is based on the circuit representation. The circuit may be represented explicitly using gates chosen from a user-defined set or, alternately, it may be represented using an interconnection of arbitrary functions. The representation based on gates is the *technology-dependent* representation while the more abstract representation is called the *technology-independent* representation.

The background material on logic synthesis and delay modeling is described in Chapter 2. Chapters 3 and 4 are devoted to understanding the performance optimization of combinational logic at the technology-independent and technology-dependent stages respectively. Synchronous circuit optimization is the subject of Chapter 5. An overview of Chapters 3, 4 and 5 is provided next.

1.3.1 Technology-independent optimizations

Combinational circuits compute some Boolean function defined over circuit inputs. There are many different ways of defining the circuit function. The traditional way is to specify the function as a sum-of-products representation. This is a two-level function — the first level consisting of AND gates whose outputs are combined using an OR gate at the second level. There are functions for which the two-level representation has size which is exponential in the number of inputs. For such functions, a multi-level representation is more appropriate. It consists of an acyclic interconnection of smaller functions.

The structure chosen to represent a function determines its delay. As an example, consider the addition of two numbers. The addition can be performed either serially by adding the carry generated from the lower order bits with the next bit or by using a parallel look-ahead structure. These represent, respectively, the small-but-slow and the fast-but-large ends of the possible designs. What a designer wants is a circuit that meets the speed requirements and has an acceptable area. To solve this problem there are two obvious approaches — 1) start with a small circuit and then tradeoff area for better speed, and 2) start with a fast circuit and then recover area while sacrificing the speed till the design objectives are met. Due to the unavailability of algorithms and techniques to design close to the fastest circuit for a given function, the first approach is the only one that we can use. Recall that current techniques can generate circuits with area close to the smallest implementation.

Techniques used for area optimization also affect the circuit delay. This interaction between area and delay is complex. On one hand, a smaller circuit offers smaller capacitive loading and helps to speedup the circuit. On the other, the reduction in area is achieved by sharing logic which results in increased delay due to greater circuit depth and increased capacitive loading on some nodes. These conflicting aspects need to be reconciled in an optimization procedure that can generate circuits with acceptable area and delay. Some of the area optimizations at the technology-independent stage that will be studied are simplification of the logic function using don't cares, kernel extraction and cube extraction. How these techniques can also be used to reduce the circuit delay will be described. Combining the different transformations that reduce delay into a unified algorithm is a major contribution of this research and will be presented in Chapter 3. The proposed approach determines a lower bound on the improvement in delay and then meets this improvement with a increase in area.

1.3.2 Technology-dependent optimizations

After the description of a Boolean function has been converted into an interconnection of functions with a small depth and acceptable area, a circuit is obtained by *mapping* the functions onto a predefined set of gates. The predefined set of gates is called a *cell-library* and typically contains more than one gate that can be used to implement a Boolean function. All the gates in the library are pre-characterized so that the delay through the gate, for a given capacitive loading at its output, can be accurately predicted.

This phase of delay optimization exploits the characteristics of the gates in the cell library to reduce the circuit delay. An important optimization is to enable gates to drive a large number of signals by building *fanout-trees* at the outputs of the gates. In addition to improving the circuit speed it also prevents defects like metal-migration during circuit operation. A polynomial-time algorithm that generates fanout trees is presented in Chapter 4. Chapter 4 also studies the interaction between the fanout-correction and the gate-selection algorithm. Some ideas are common to the application of local transformations at the technology-independent and technology-dependent stages. However since the delay model used during technology-dependent optimization is more complex, different strategies are used in the two cases to select the local transformations that will improve the circuit performance.

1.3.3 Synchronous circuit optimization

Once the performance optimization of combinational circuits has been addressed, the optimization of synchronous circuits can be studied. These circuits contain combinational logic interspersed with registers. The speed of operation is determined by some long path between registers. So, it is clear that the reduction in combinational delay would decrease the cycle-time for the system. One question that will be addressed in Chapter 5 is how to generate constraints on the combinational logic to ensure that the circuit will operate at the desired speed. In addition to *combinational resynthesis*, circuit performance may be improved by repositioning registers to better distribute the delay among the various clock phases. This technique of *retiming*, proposed by Leiserson [34], is applicable to single-phase edge-triggered systems. In order to be applicable to multiple-phase designs, positions of registers are changed in response to the distribution of delays in the circuit.

There is no reason to restrict these resynthesis and retiming techniques to be applied independently. Wherever possible, a maximal chunk of combinational logic with registers at its periphery can be identified. Then by resynthesis on this larger piece of combinational logic we have greater freedom in reducing the delay and can obtain results better than those that were possible by applying separately the techniques of retiming and combinational resynthesis. The conditions under which a circuit can be *peripherally retimed* were presented in [41]. The extension of these techniques to timing optimization was presented in [42] and will be described in this chapter as well. This technique is also used to optimize finite-state machines.

Finally, in Chapter 6, experiences gained during this research will be presented and avenues for further research described.

Chapter 2

Background

This chapter is devoted to reviewing some basic notions in the area of logic synthesis and timing analysis.

2.1 Logic synthesis preview

A Boolean function is a mapping $F : \{0,1\}^n \rightarrow \{0,1\}$. Each of the n inputs of the function is called a **variable.** The set of input conditions for which the function F evaluates to 1 (0) is called the **onset** (**offset**). The onset is typically represented as f and the offset as r. An **incompletely specified** Boolean function has the flexibility of producing a 0 or 1 at the output under some input conditions. These inputs represent **don't care** conditions and are represented by d. An incompletely specified function $\mathcal{F}$ represents a range of functions. The minimum is the onset and the maximum is when the entire don't care set is included. Thus $f \subset \mathcal{F} \subset f \cup d$.

Boolean functions, both completely and incompletely specified, with more than one output are called **multi-output** Boolean functions. The term Boolean function will be used to describe both single and multiple-output functions, and also completely and incompletely specified functions. When required, the exact description will be used.

Boolean functions can be represented in a variety of ways. A representation that describes the output value for every input combination is called the **truth-table** and it has a size exponential in the number of inputs. An improvement is to group together the input combinations in the onset (or offset) into **cubes** that represent a conjunction of literals (a literal is a variable or its complement). A collection of cubes that represents the

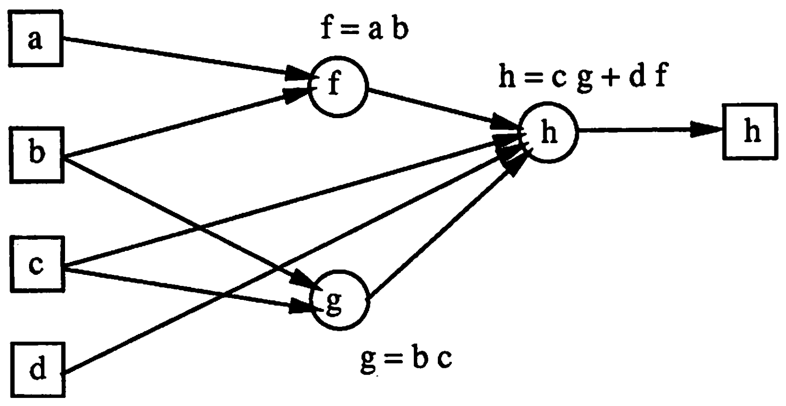


Figure 2.1: Example of a Boolean network

function is called a **cover** of the function. This representation is natural when the circuit is represented as a Programmable Logic Array (PLA) [43]. These circuits implement a **two-level** structure. The first level is devoted to computing each cube (a conjunction (AND) of the literals in the cube). The second level performs a disjunction (OR) of the cubes that describe an output. Due to their compact representation PLAs are widely used to implement control logic. However, for large functions, the PLA results in a slow circuit implementation. The PLA design is also very inefficient for some classes of functions (e.g. parity). In the quest for smaller representations and greater circuit speed logic designers turned to **multi-level** circuit implementations. Rather than be restricted to two levels of AND and OR functions, these circuits allow an acyclic interconnection of general functions of arbitrary depth.

The representation of Boolean functions as a multi-level circuit is called a **Boolean network**. A Boolean network is a directed acyclic graph (DAG) consisting of vertices (nodes) and edges. An example of a Boolean network is shown in Figure 2.1. The inputs and outputs of the Boolean function are represented by vertices labeled **primary inputs** and **primary outputs** and cannot be changed. The remaining vertices are **internal**. Each internal vertex g is associated with a completely specified Boolean function f_g and a Boolean variable y_g. The edges in the Boolean network represent the interconnection between nodes. An edge from node g to node h implies that the function f_h at node h uses as input the variable y_g associated with the node g. Node g is called a **fanin** of node h while h is a **fanout** of g. The set of all the fanins of h is represented as $FI(h)$ and the set of all the fanouts of g is represented as $FO(g)$. The distinction between the variable at a node and the node itself is important although frequently the same symbol is used for both. When f_h is specified in terms of its immediate fanins, it is called a **local function**. On the other hand, f_h is called the **global function** at node h when it is represented in terms of the primary inputs of the circuit. The variables that a function explicitly depends upon is called the **support** of the function. The maximum number of internal nodes on any input-output path is called the **depth** of the network.

A set of primitive functions that is supplied by the user is called a **cell library**. The elements of the cell library are called **gates** and each gate is characterized for its area and delay. A single function may be represented multiple times in the cell library. Gates that have the same functionality but different area/delay characteristics are called **versions** of each other. A Boolean network where each of the nodes is equivalent to a specific gate (functionally and in terms of its area and delay) is called a **logic circuit**. We will use the term **node** to mean both a vertex in a Boolean network and a gate in a logic circuit whenever there is no ambiguity. In that sense, a logic circuit is simply a Boolean network in which each node is annotated with the gate that is used to implement the function at the node. We will use the term circuit to refer to both logic circuits and Boolean networks expecting that the context will make the interpretation clear.

The process of optimizing a Boolean function to obtain a logic circuit is typically done in two phases. The first phase finds an alternative Boolean network that represents the same function (or a function that lies between f and $f \cup d$). This is the **technology-independent phase**. During the next phase, the optimized circuit is implemented using the gates in the cell-library. This operation is called **technology mapping**. As a result of technology-mapping, a logic circuit is obtained. Optimizations that exploit the delay properties of the gates in the cell-library and retain the mapped nature of the circuit are called **technology-dependent** operations.

Most circuits contain, in addition to combinational logic, elements that are loosely called "storage" or "synchronizing" elements. The presence of these elements transforms combinational circuits (in which the output is a Boolean function of the input) into **sequential circuits** (those in which the output is a function of the inputs and the past history). The values stored in the storage elements are referred to as the **state** of the system. Sequential circuits in which there is at least one storage element in every cyclic connection of gates are called **synchronous circuits** if all the storage elements are controlled by a specific set of signals called **clock** signals. The storage elements in synchronous designs can be classified into two types — edge-triggered or level-sensitive [43]. For convenience we will call edge-triggered devices **flip-flops** and the level-sensitive devices **latches**. Flip-flops output a value equal to the stored value until the next clock-edge which may cause the stored value to change depending on the inputs present at the time of clocking. For latches, the **active** period is defined as the time interval during which the output waveform responds to the input waveform. During the active period the latch is "transparent" (the output follows the input). At the instant they become inactive, they store the input value and output it till they become active again.

2.2 Determining circuit performance

Circuit simulation is the most accurate method for determining the timing characteristics of the signals in a circuit (short of fabricating the circuit and measuring its delay). Circuit simulators like SPICE [46] provide an accurate estimate of the circuit delay. However, for large combinational circuits circuit simulation is an expensive operation and cannot be invoked repeatedly during an optimization procedure. In addition, prior to physical design, only poor estimates of the capacitive load and wire resistances are available. Hence, using an accurate simulator at the logic level is not useful.

To address the speed of analysis and the unavailability of an accurate representation for the circuit at the logic level, *delay models* are used to predict the delay of each component in the circuit. The delay through a circuit is computed by combining the delays of the components during timing analysis (Section 2.2.3). This process is fast and can be used repeatedly during optimization.

2.2.1 Delay models

A **delay-model** is an equation or algorithm that predicts the delay between the inputs and output of a gate within the context of the circuit surrounding the gate. Delay models may be used at various levels of abstraction. As more information is available regarding the implementation, the delay models produce estimates of circuit delay that are closer to the delay value obtained from circuit simulation. After technology-mapping (when the gates used to implement the logic are known) we can predict the delay more accurately than with the technology-independent representation (acyclic connection of arbitrary functions). Some of the delay models used during synthesis are described next.

Library-based models

To estimate the delay through a gate in the cell-library, various delay models [81, 49] have been suggested to account for factors such as output loading, slow-changing input-waveforms and resistive effects of wiring. To generate such models the user conducts a number of simulations by varying the parameters of interest. Typical parameters used include the output load and the slope of the input waveform. The circuit delay is then known at a number of points in the parameter space. Curve fitting of the observed delays to the parametric equation yields the model equation.

For many cell-libraries, the model equation is a linear equation where the delay between an input i and the output o is a function of the output load.

$$d(i, o) = \alpha(i) + \beta(i) \times L(o)$$

α represents the intrinsic-delay of the gate (also called **block** delay) and β is a load-multiplier (also called **drive** or **output-resistance** of the gate) that represents the additional delay per unit output load. $L(o)$ is the cumulative capacitance that is driven by the output. The **library** model is simply the set of precharacterized values of α and β for each input pin of the gate. Due to differences in the transistor configurations that charge and discharge the load capacitance, the parameters used to compute the delay for the rising and falling transitions at the gate output are usually different.

The library-based models require that the logic have a mapped representation. Prior to the mapping process there is no information on what gates are used to implement the function. In this case technology-independent models are used. These fall into two categories — predictive models and structural models.

Predictive models

Predictive models assume that specific optimizations and mapping strategies are applied on the function at each node in the Boolean network. How these strategies affect the delay is then captured in terms of predictive models. A few of the predictive models that have been proposed are based on a mapping of each node in isolation, on a balanced two-level decomposition of the function at a node [73] and on a timing-driven-cofactoring of the node-function [23].

For Boolean networks that contain unmapped nodes the **mapped** delay model may be used to approximate the circuit delay. To apply this model, a mapping of each node in the network is carried out and the α and β values are computed by analyzing the delays in the mapped representation of the node. Note that this mapping is performed on a node-by-node basis and may not reflect the mapping that would be obtained if the entire network was to be mapped. Furthermore, the mapping of each node may be performed for minimum area or minimum delay. Based on the mapping criterion, we get **mapped-a** and **mapped-d** delay models respectively.

The model of [73] predicts the delay of a node as

$$d = a_0 + a_1 \ln(ts) + a_2 \ln f$$

where t is the number of cubes in the two-level cover of the function at the node, s is the maximum number of literals in any cube, f is the number of fanouts of the node and a_o, a_1, a_2 are constants derived from the cell library. A number of functions are generated randomly and mapped into the target cell-library. The delay through these functions is computed based on the *library* delay model. The parameters for the predictive models are chosen so that the predicted delay has the minimum mean-square error over the observed delays. This enables the model to predict the delay of the circuit after the Boolean network has been mapped into a specific cell-library. Since the model equation is based on a balanced, two-level decomposition of a node, it has some limitations. When the two-level representation at a node is large, e.g. for the parity function, a multi-level representation usually has smaller delay. Moreover, when the inputs arrive at widely different times, prediction based on a balanced decomposition is pessimistic. Hence, the model is accurate only when the functions in the Boolean network have been decomposed into smaller functions with similar arrival times at the node inputs. This assumption is very restrictive since one of the aims of technology-independent optimizations (using the predictive model) is to come up with a good decomposition of the Boolean network. Thus assuming a good decomposition to start with is unacceptable.

The timing-driven-cofactoring model (**tdc**) overcomes the restriction that all inputs have similar arrival times. The inputs are divided into groups such that the signals in

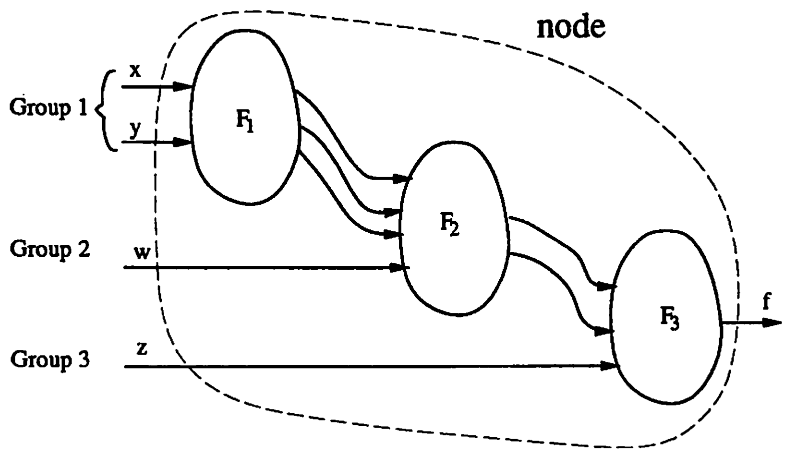


Figure 2.2: Grouping signals based on arrival time

each group have similar arrival times. This creates a staggered decomposition of a node like the one shown in Figure 2.2. The predicted delay between an input signal, say w belonging to group 2, and the node output f, is the sum of the delays through the functions between them (F_2 and F_3). Since inputs to each of the smaller functions arrive at similar times, it is easy to predict their delay. This enables the model to predict different delay values based on the the arrival times of the signals. However, the model has no consideration of fanout which reduces its accuracy. In addition, for large functions this model is computationally expensive.

The delay models of [73] and [23] have large errors when applied to logic functions that do not fit their domain of validity. This is a major drawback since it is not known a priori if the delay model is valid. In fact, it is clear that developing a good predictive model is a difficult task. The different ways in which the circuit representation affects the delay should be handled by a good model. A good model should be able to predict a delay close to the best attainable. This means that built into the predictor is a procedure to generate an implementation with small delay. Thus to build an "accurate" predictive algorithm one has to solve the timing optimization problem itself!

Structural models

In the absence of good predictive models, the only recourse is to use a delay model that is based on the mechanisms that contribute to circuit delay. A simple observation is that increasing the depth of the Boolean network increases the delay through the circuit. Thus the depth of the Boolean network is an important parameter that affects the circuit delay. The depth of the network is measured using bounded-fanin gates. Without this restriction the depth of the network is not a good predictor of delay (since all networks can be expressed in two-level). Results from circuit complexity [16] show that the bound t on the fanin of each node does not change the asymptotic complexity of the function. Typically a value $t = 2$ is used. The delay model that considers each gate to contribute 1 unit of delay is called the **unit** delay model.

In a circuit, there is additional delay since gates have to drive capacitive loads. Capacitive loads result in a slow switching of a transistor that can contribute significantly to the delay. Even though considerations of fanout are getting increasingly important (as device dimensions decrease so does the current sourcing capacity leading to larger charging times) it is difficult to model the delay contribution of the fanout. Using a linear delay model, like the **unit-fanout** model that computes the delay of each gate as $(1+0.2\times\text{fanout-count})$, the delay predicted for a node driving a large number of fanouts is very high. Clearly buffering the nodes is an optimization that will certainly be performed. So, the delay model should account for the existence of buffering algorithm and not predict a large delay. Buffering the gates in a circuit results in every node having a limited number of fanouts. Thus putting a bound on the number of fanouts that a node can drive, results in all bounded-fanin nodes having similar delays. In that case the network depth is a good measure of the circuit delay. In [26] it is shown that a network with arbitrary fanout can be replaced by a network with bounded fanout with only a constant factor increase in depth. The inaccuracy in modeling fanout and the result of [26] suggest that considerations of fanout can be relegated to technology dependent optimizations and consequently using the depth of the network (when every node has a bounded fanin) is reasonable.

2.2.2 Evaluating technology-independent delay models

To evaluate the technology-independent delay models, the true circuit delay is compared against the delay predicted by the different models. The technology-independent models are used to predict the delay prior to technology-mapping for all the examples in the benchmark set (see Section 3.4.1). The network is then mapped into the `lib2` cell-library provided with the MCNC benchmark suite and the circuit delay is measured using the *library* delay model. The cell-library, `lib2`, has different versions for some functions. In another experiment, in order to avoid the bias due to the presence of high-speed components, only the slow version of each cell is retained. This reduced library is named `lib2-subset`. Mapping for minimum delay is performed using the `map` command of SIS with options to reduce delay (`-m1`) and correct for large capacitive loads (`-A`).

Various statistics have been proposed to evaluate the delay models. The true network delay, d_i, and the predicted delay, e_i, represent the observation sets that we want to study. The correlation between the predicted and actual delays represents the degree by which one delay tracks the other. For a perfect predictor, whose correlation coefficient is 1, each value e_i is linearly related to d_i, i.e. $e_i = m \times d_i + c$.

One measure of "goodness" of a predictor, the Overall Relative Error (ORE), is proposed in [73]

$$ORE = \sqrt{\frac{\sum (e_i - d_i)^2}{\sum d_i^2}}$$

By re-expressing the observed value e_i as $e_i = (1 + r_i)d_i$ where r_i reprepresents the relative error, the ORE may be written as

$$ORE = \sqrt{\frac{\sum (r_i d_i)^2}{\sum d_i^2}}$$

The ORE weights the relative errors with the actual delays. This emphasizes the relative errors at the points with large d_i's, which is desirable, since the larger delay values are the ones that typically determine the network speed.

Another statistic that is useful in evaluating the delay models is the Maximum Relative Error (MRE). This represents the largest percentage that the prediction differs from the actual delay.

$$MRE = \max \left[\frac{|e_i - d_i|}{d_i} \right]$$

In evaluating the ORE and the MRE, we first normalize the predicted value, e_i, to the same scale as the actual value, d_i. This is done to avoid large errors that occur when different scales are used — actual delay may be in nano-seconds whereas the network depth (used to predict the actual delay) is an integer. Normalization is done by finding the linear curve ($e = \hat{m} \times d + \hat{c}$) that best fits the data. For each d_i value, a scaled value $\hat{e_i} = \hat{m} \times d_i + \hat{c}$ is generated. Thus $(e_i - \hat{e_i})$ is the error between an ideal (linear) predictor $\hat{e_i}$ and the actual values e_i predicted by the delay model. In the computation of the statistics the value $\hat{e_i}$ is used instead of d_i.

Figure 2.3 and Figure 2.4 show the actual circuit delay and the predicted delay corresponding to all the models, for all the 1583 outputs in the benchmark circuits, when `lib2-subset` and `lib2` are used as the target libraries respectively. Also shown as the solid line is the minimum-mean-square error linear curve through the data. In the scatter plot, a single point may represent more than one occurrence of the same datum and because of this, the best-fit curve sometimes appears to be different from what one might expect if there were no multiple occurrences of data values.

Table 2.1 provides a summary of the "goodness" of each of the models. Overall, the *unit* delay model using 2-input gates is the best predictor of the circuit delay at the technology-independent level. It is also a very simple estimator. For all the other predictors, one may attribute the poor correlation to the fact that they overestimate the delay at high-fanout gates. The mapping uses a buffering algorithm to reduce delay. This buffering is not accounted for in the delay models. Linear delay models are unrealistic when gates with large fanouts are present in the circuit. This effect may have been amplified in `lib2-subset` when the high power gates were removed from the library. To study the effect of a rich library on the predictors, a realistic library, `lib2`, is used. The order of the predictors remains unchanged. However, for all the predictors except the *unit* model, the prediction quality improves. This is consistent with the expectation that the presence of high-power gates should alleviate the discrepancy between a linear model and the fanout correction algorithms. A logarithmic model for fanout, like that of [73], results in a maximum ORE of 0.332 which is better than the ORE for any of the linear models.

The conclusion of this experiment is that the number of levels is the best **overall** predictor (based on correlation and ORE) for the true circuit delay at the technology-independent level. However factors such as fanout-loading and the richness of the library may result in drastic differences for specific outputs as evidenced by the large value for the MRE. The large value for the MRE for all the models suggests that these models are suitable only as gross estimators of delay and that they should not be used to differentiate path lengths in unmapped networks.

The relative merit of the *unit* delay model, along with its generality and simplic-

Delay	`lib2-subset`			`lib2`		
Predictor	Correlation	ORE	MRE	Correlation	ORE	MRE
mapped-a	0.769	0.516	2.96	0.794	0.486	3.34
mapped-d	0.815	0.454	2.79	0.835	0.437	3.92
tdc	0.615	0.728	3.70	0.642	0.698	2.67
unit-fanout	0.679	0.614	2.92	0.688	0.605	3.18
unit	0.928	0.269	1.34	0.893	0.327	3.92

For MRE computation only outputs with actual delay greater than 5.0 were used. There are 1292 (for `lib2-subset`) and 1208 (for `lib2`) such outputs.

Table 2.1: Evaluation of technology-independent delay predictors

ity, makes it the delay model of choice when optimizing the network at the technology-independent level.

2.2.3 Timing analysis

Once an appropriate delay-model has been chosen, the delay from the input of a gate to its output can be computed. The notation $d(g, f)$ is used to denote the delay from the input g of node f to the output of node f.

A **clocking scheme** is the temporal-ordering of the signals that control the registers in the circuit. Certain timing constraints must be satisfied [8, 70] for the circuit to function correctly. Based on the type of registers and the delays through combinational gates, timing analysis determines parts of the circuit that do not meet performance requirements. This process is called timing verification and several techniques have been proposed for it. Some techniques [77, 54, 29] are based on a path-by-path analysis which is accurate but slow due to the exponential number of paths. During the optimization stage, a fast analyzer is called for. To gain speed (at the expense of accuracy) many timing analyzers [63, 74, 76] adopt a "block-oriented" approach which is linear in circuit size. One such analyzer that is specifically designed for use in a logic-synthesis environment is `hummingbird` [76].

It should be emphasized that timing verification is concerned only with ensuring that the timing constraints required for correct circuit operation are satisfied. It does not address the problem of determining if the correct functionality has been implemented. Often, simulation is used to check if the circuit operates as desired. When the simulation results agree with the specification, timing constraints are satisfied as well. However, since the set of simulations may not include all input conditions, a circuit that passes a limited set of simulations can have timing violations that were not stimulated by the simulations. Thus timing verification is required independent of the technique used to check the functional correctness of the circuit.

In the "block-oriented" approach only the maximum and minimum delays are recorded. Starting at the registers, where the **arrival time** of the signal is known based on when the register generates a new value, the arrival time for every node in the circuit is computed. The arrival time of a node is interpreted as the latest time that a signal can arrive at the output of the node. The arrival time of a node f is denoted as $a(f)$ and computed as follows

$$a(f) = \max_{g \in FI(f)} \{a(g) + d(g, f)\}$$

This process terminates when circuit outputs or the inputs of registers are reached. At that stage the latest that the signal can arrive at the input of the register is known. This can be checked against the timing constraint described as the setup constraint (the input must arrive at least a certain time before the register is going to store the value). In the case of a violation it is clear that the signal arrives later than it is required. This leads to the concept of **required time** of a signal. For the inputs of registers (the output of combinational logic blocks) the required time is set as the latest by which the signal should have arrived at that terminal for correct circuit operation. The required time for every wire and node in the circuit can be computed. Associated with a connection between nodes g and f is the required time $r(g, f)$. It represents the time value before which the signal should arrive at the input of node f along the connection from g for the circuit to work.

$$r(g, f) = r(f) - d(g, f)$$

Since node g may have fanouts to several nodes, each one of which will impose a requirement on when the signal must be ready. The minimum of these is used as the required time for the signal produced by gate g. For a node g the required time is denoted as $r(g)$ and is

$$r(g) = \min_{f \in FO(g)} r(g,f)$$

The required time computation proceeds from the circuit outputs towards the circuit inputs. Once the two passes that compute the arrival time and required time have been computed we can determine the **slack time** at each node. The slack time at node g is $s(g) = r(g) - a(g)$ and at a connection from g to f is $s(g,f) = r(g,f) - a(g)$. Any wire or node that has a negative slack represents a violation in the timing constraint (the arrival time exceeds the constraint imposed by the required time) and is called a **critical** wire or node.

In the case of latches that are transparent for some duration, two passes are not sufficient. The coupling of latch-input and latch-output results in having to solve a set of simultaneous equations that relate the arrival and required times [65]. By solving the coupled simultaneous equations, a timing analysis computes, for every node and wire in the circuit, the arrival, required and slack times. A violation of the timing constraints results in paths with negative slacks. The circuit must be modified so that delay along the critical paths is reduced. Delay along critical paths may be reduced by selecting faster gates or by applying more drastic steps like changing the circuit structure. Opportunities may exist to reposition the registers and/or modify the clocking scheme for better timing characteristics. For designs containing latches, there may be additional violations when some paths have length less than a specified value. These combinational paths have to slowed down to ensure correct operation.

The next section discusses some of the transformations used to optimize combinational logic.

2.3 Logic optimization operations

The optimization of multi-level circuits has been a focus of considerable research. *Proceedings of the IEEE*, in its February 1990 and May 1991 issues, has excellent review papers on this subject. The focus of the optimizations has been to reduce the circuit area. A circuit with smaller area is a more compact representation of the function and leads to a reduction in the size of the implementation. The smaller size results in smaller wiring delays and this is helpful in reducing the delay as well. However, technology-independent optimizations like common sub-expression extraction and simplification may increase circuit depth. Also, sharing commonality may increase the fanout of the common nodes. If these nodes lie on the slow paths, the circuit delay increases. The following sections describe some of the operations that are used during optimization. The effect of these transformations on the circuit delay is also described. This provides clues as to how these operations may be adapted to help reduce circuit delay.

Simplification using don't cares

Simplification is used to find a compact representation for the Boolean function at every node. A compact representation of a function seems to be a natural starting point for any optimization. It is clear that by removing the redundancies from a representation of a function the size and depth can be reduced. In a multi-level network the simplification at a node considers the structure of the logic around it. This gives rise to don't care conditions that can be exploited during node simplification [75]. The use of don't cares for simplification affects the circuit delay as well.

The Satisfiability Don't Cares (SDC) represent the combinations of values that cannot occur in the network, e.g. the input and output of an inverter cannot take on the same value. Under these don't care combinations the local function can produce any value. This extra flexibility can be used to reduce area. Using don't cares during simplification often results in Boolean resubstitution [75] of one node into another. How the resubstitution operation affects delay is described later in this section.

The Observability Don't Care set (ODC) at a node represents the conditions under which the signal value at a node does not influence any circuit output. Under these conditions, the node output can be either a 1 or 0. Using the ODC's, the range of global functions at a node can be determined. Any function within this range is permissible and produces the correct circuit output. This flexibility leads to a number of equivalent networks that differ in their area and delay. By choosing a function that depends on signals with smaller arrival times, the circuit delay can be reduced [9].

Common sub-expression extraction

By **extracting** common-subexpressions from a number of functions, the circuit area is reduced. However, the better the area-saving, the more places the sub-expression fans out to. This could degrade circuit performance if there are critical paths that pass through the extracted common sub-expression. It is also possible that the common sub-expression selected for maximum area-saving depends on a late arriving signal. In this case, the late arriving signal passes through more levels of logic, resulting in greater delay. Figure 2.5 illustrates the process of extraction when one signal, namely c, arrives later than other signals. By extracting the sub-expression common to all the functions, c passes through more levels of logic. Extracting another sub-expression that does not contain c provides a better delay at the cost of increased area.

The extraction of common sub-expressions is based on a sum-of-products (SOP) representation of the functions. An alternate realization for a function is to implement its complement and add an explicit inverter between the complement and the destinations. The SOP representation of the complement may lead to a more compact decomposition. Determining for each node of the network whether to implement the node-function or its complement, so that the circuit area is reduced, is NP-complete [75]. Heuristic techniques to choose the phase of a function (complemented or uncomplemented) are also presented in [75]. Complementing the node-function may also lead to a representation with smaller delay through the node. An extra level of inversion has to be accounted for. When a signal is being used in both phases, no extra level of inversion is introduced. At the technology independent stage, the inversions are represented implicitly (rather than represent the complement using an inverter, the variable is complemented in all its occurrences) and do not affect the depth of the circuit.

Resubstitution and Elimination

Another optimization that is similar to common sub-expression extraction is the operation of **resubstitution**. The process of resubstitution involves expressing a node in terms of another, if possible. Both the function and its complement may be substituted. For example, given two functions F and G

$$F = abf + \bar{a}cd + \bar{a}de + cdf + ef$$
$$G = ab + cd + e$$

F may be re-expressed using G as

$$F = fG + \bar{a}cd + \bar{a}de$$

or as

$$F = \bar{a}dG + fG = (\bar{a}d + f)G$$

The first representation is called an **algebraic resubstitution** since no laws of Boolean algebra are used. In fact, F and G are treated as polynomials. The second representation uses Boolean identities ($\overline{a} \cdot a = 0$ and $d \cdot d = d$) to further simplify the expression and is called **Boolean resubstitution.** The inverse operation of resubstitution is called **elimination.** It involves removing from the expression of a node all occurrences of variables that represent the nodes that are eliminated. Thus if node G were to be eliminated from the compact representations of node F, we would get the initial expression for F. When all the internal nodes are eliminated the operation is called **collapsing.**

Resubstitution is often used to exploit commonality between sub-expressions that have been extracted from several functions. It appears that this transformation reduces the delay since it decreases the total number of wires in the circuit. However, the merging of two nodes with the same function, only one of which lies on a long path, causes excess loading along the long path. This leads to an increase in the delay along the long path, degrading the performance. Since the critical paths in the circuit are not determined until technology mapping, it is difficult to estimate the effect of this transformation at the technology-independent level. Also, technology-dependent transformations may decide to undo the resubstitution by duplicating some gate and driving most of the non-critical signals by one copy, thereby reducing the load and delay along the critical path.

Since resubstitution ignores arrival time data, it may make resubstitutions that lead to placing late arriving signals far from the outputs. Consider the previous example and assume that node F lies on the critical path while G does not. If signal b arrives late, then after resubstitution of G into F, b passes through more levels of logic. An alternate representation for F, in which b passes through a smaller number of levels, would be more favorable for delay even though it has larger area.

Repeated application of elimination, simplification, extraction and resubstitution is used as a typical strategy to reduce circuit area. The description of the these operations shows the complex interaction between the area and delay of the circuit. In addition the same transformation applied on the same network may be good or bad depending on the delay data (the arrival and required times). Since at the technology independent stage the delay data is imprecise, it is difficult to adapt the global strategies used for area optimization to address performance optimization. It is therefore not surprising that most techniques developed to reduce delay use local transformations to make incremental changes to the logic. In the following chapters reference will be made to previous work in the area of performance optimization to point out the important contributions of this research. The next section provides a brief description of the work in performance optimization at the logic level. For an extended survey of performance optimization techniques at different stages of design the reader is referred to [1].

2.4 State of performance optimization techniques

Initial attempts at improving the delay of a circuit focussed on making local changes only along the most critical path. [25, 11] reduce the delay by adding buffers and decomposing an *existing gate* into gates containing early and late arriving signals, with the latter being placed closer to the output. The optimizations are performed to reduce the network depth. As is pointed out in [11], the delay of the circuit is difficult to predict at the technology independent stage. To overcome this limitation a rule based system, SOCRATES [5], was developed that operates on a mapped circuit. Depending on the characteristics of gates in the cell-library, rules were defined to transform regions of the circuit into faster implementations. The strength of this approach is that it exploits the features of the library and technology being used. However, the generation of the rule-base is difficult and strategies to apply the rules may change with a change in technology.

In [12, 61] a more "global" view to the optimization problem is taken. A set of critical paths is identified and then a set of nodes is determined such that reducing the delay at those nodes results in better performance. The choice of the nodes at which to change the circuit structure locally is made so as to get the most improvement in delay for a small increase in area. Rather than using rules to generate a faster implementation, these methods restructure the logic on the fly based on the arrival time data. The optimizations are limited to be algebraic in nature and they do not exploit the ODC and SDC sets.

Techniques that try to reduce the depth of the circuit by exploiting don't care conditions in the circuit are described in [6, 9]. Using the ODC, it is possible to reduce the length of the long paths by re-expressing intermediate functions in terms of signals with smaller arrival times. A recent attempt at timing optimization [68] forms clusters of specified size to reduce the maximum depth of any cluster. The nodes in each cluster are collapsed to yield network of small depth. The network is then optimized to reduce the area without increasing the circuit depth. This constraint is enforced by restricting the extraction, resubstitution and simplification operations so that they do not add extra levels of logic.

The primary aim of all the restructuring approaches is to get a good multi-level structure of the circuit that will subsequently be mapped into a small delay implementation. They use simple, weak models to predict circuit delay. As a result, it is difficult to be confident that the savings observed at the technology-independent stage will be evident after technology mapping of the optimized circuit.

To alleviate this problem, researchers have extended the basic ideas on the technology-independent optimizations to work on mapped circuits. The works of [80] and [18] present heuristics to address the optimization of mapped circuits taking into account the characteristics of the cell-library. Chapter 4 will revisit these techniques to contrast them with the approach proposed therein. We will also show how the earlier techniques can easily be incorporated in the proposed optimization framework and results of doing so will be discussed.

The next chapter is devoted to understanding performance optimization at the technology-independent level. Even though the delay model used there is the *unit* delay model, the techniques described will form the cornerstone of a procedure to reduce circuit delay with the minimum increase in area. The procedure can exploit different local transformations, each of which reduces delay locally, to effect global decrease in delay.

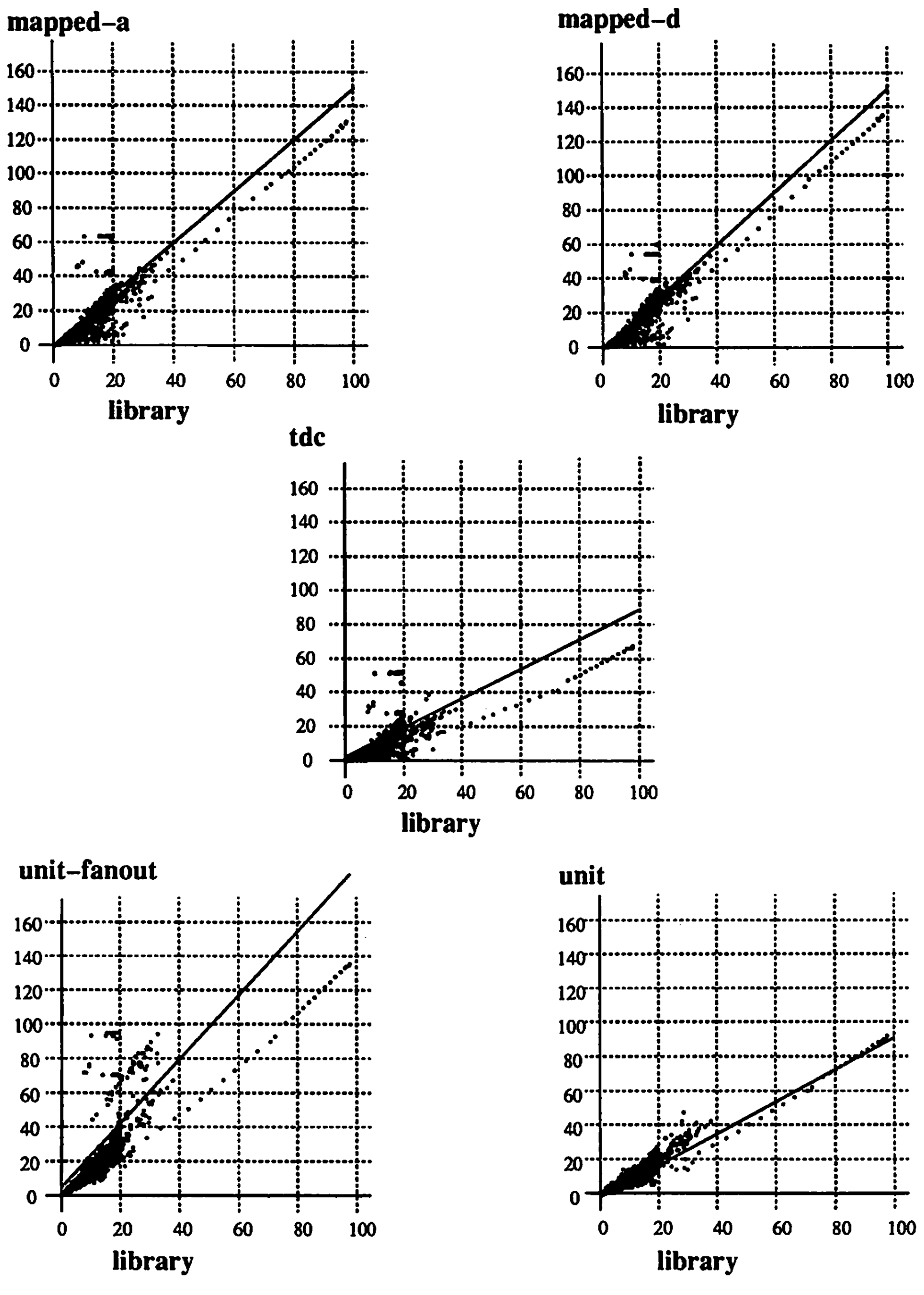


Figure 2.3: Delay prediction using a uniform cell-library, `lib2-subset`

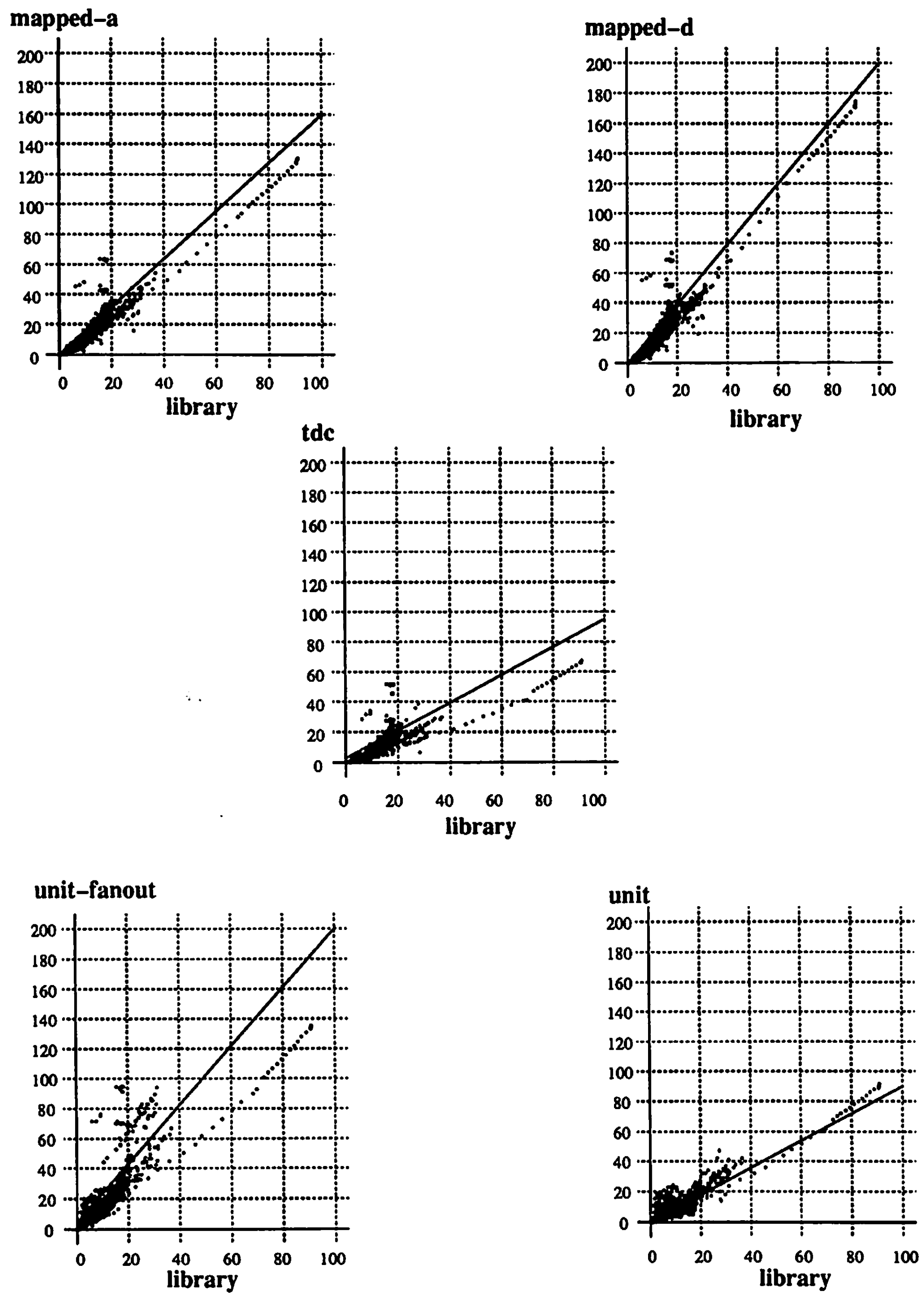


Figure 2.4: Delay prediction using the MCNC `lib2` cell-library

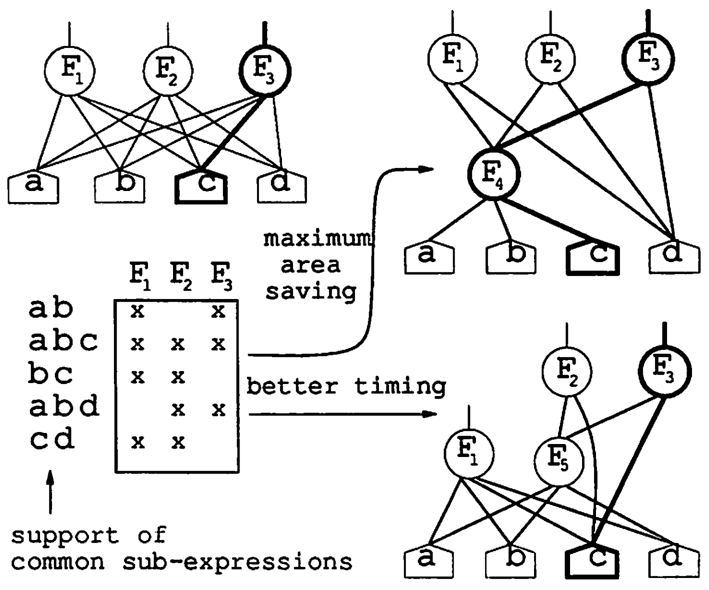


Figure 2.5: Effect of extracting common sub-expressions on delay

Chapter 3

Technology-independent optimizations

Experience gained from combinational logic synthesis has shown that the process of optimizing circuit area can be decomposed into two parts — a technology-independent optimization phase where the circuit function is represented in a multi-level structure, and a technology dependent optimization phase where the logic equations are transformed into an interconnection of gates chosen from the user-defined cell-library. At the technology-independent state the optimizations aim at reducing the size of the functions used to represent the circuit while the technology-dependent optimizations exploit the characteristics of the cell-library to reduce area. The division of the synthesis process into a technology-independent and technology-dependent part allows the designer to focus on an abstract representation of the circuit functionality at an early stage of the design process.

The success of technology-independent procedures to reduce area can, in part, be attributed to the availability of a good abstraction for the circuit area. The number of literals in the factored form of a function is a good estimate of the active area required to implement the function [36]. The literal count estimates the number of transistors required to implement the function. For delay, it is not immediately obvious what is a good abstraction at the technology-independent level. The representation that we use is a decomposition of the network into 2-input NAND gates. As was described in Section 2.2.1 the number of levels of logic is a good predictor of delay when gates have bounded-fanout and the library has gates of similar drive capabilities. This representation is also used widely in the circuit complexity literature [56] and is easy to compute. Another motivation for using 2-input nodes at the technology-independent phase is that the technology mapping procedure [30, 13] that follows it operates on a two-input representation of the Boolean network. The choice of a 2-input NAND representation allows the results of technology-independent timing optimization to be input directly to the mapping algorithms. This is important since any other representation of the logic at the technology-independent stage would require a decomposition step to transform the representation into 2-input nodes. If the user is not careful in choosing the decomposition step, this could invalidate some of the optimizations performed earlier.

The complex interaction between circuit area and delay (Section 2.3) and the difficulty in estimating the delay of large Boolean expressions at the technology-independent level (Section 2.2.1) suggests that the problem of targeting technology-independent optimizations towards reducing delay is a difficult one. So, rather than trying to optimize the circuit for delay (which is difficult to estimate and optimize), it is easier to mimic the traditional strategy used by designers to get fast circuits. The initial implementation is optimized for area, which is important for achieving a small layout, and then incremental changes are made to the circuit to reduce its delay. This is the opposite point of view to that of [68] where the delay of the circuit is reduced unconditionally (without regard to area) and then the area is reclaimed by restricting the technology-independent transformations such that the number of levels in the circuit is not increased. The proposed approach has a few features that distinguish it from the work of [68]. These are —

- The proposed method is incremental and can handle circuits with timing constraints on the inputs and outputs. The approach of [68] tries to reduce the maximum number of levels without regard to timing constraints. Often such drastic reduction is not needed.

- The proposed approach optimizes the actual delay of the circuit (its accuracy depends on the delay-model being used) rather than an abstraction like the circuit depth. This is an important consideration when the timing constraints are generated after a timing analysis. Translating the absolute delay values into number of levels is difficult.

To apply incremental changes to the area-optimized circuit, there is a need for operations that modify an initial implementation into another realization that improves the performance locally. Section 3.1 describes some of these and Section 3.2 develops a framework within which these different local optimizations can be evaluated and selected to improve overall circuit delay. The use of these transformations and optimization strategy is evaluated in experiments of Section 3.3 and Section 3.4.

3.1 Delay-reducing transformations

This section describes some transformations that lead to a local improvement in the circuit delay. The transformations are generated on the fly based on the circuit configuration to be modified. This is different from the local transformations used in rule-based systems like LSS [11] and SOCRATES [5] that used a predefined set of transformations based on the design style and target technology being used. The advantage of generating transformations algorithmically is that there is no need to change these procedures when a different target technology is considered. Each transformation restructures a part of the circuit into an alternate form that has a smaller delay (measured as a reduction in the arrival time at the output of the region). In describing the transformations, it is assumed that the region on which the transformation is applied has a single output. The function computed by this region is described in terms of the inputs to this region, not in terms of the primary inputs, and remains unchanged after the application of the transformation.

3.1.1 Timing driven simplification

The representation of the logic function at every node is made prime and irredundant to get a minimal representation. A minimal representation of the logic results in a small area implementation. The function is represented as a two-level structure, a sum-of-products representation, that consists of an OR gate fed by a number of AND gates representing individual cubes. By applying deMorgans laws, the OR and AND gates can be replaced by NAND gates. For this two-level NAND-NAND representation, the optimum decomposition into 2-input gates is known (the procedure AND_OR_DECOMP of [75][page 167]). It is based on another procedure NAND_DECOMP that is used to generate the optimum decomposition of a NAND function. For clarity both these procedures are described in Figure 3.1.

In [75] it is proved that the AND_OR_DECOMP procedure is optimum for delay when the cubes of the function do not share any inputs i.e. the cubes have disjoint support. For arbitrary two-level functions we would like to know how the delay after decomposition is

```
AND_OR_DECOMP($f$) {
    for each cube $c_i \in f$ {
        $s_i$ = $\overline{c}_i$
        NAND_DECOMP($s_i$)
    }
    $f = \overline{s_1 \cdot s_2 \cdots s_m}$
    NAND_DECOMP($f$)
}


NAND_DECOMP($f$) {
    if $|FI(f)| \leq 2$ return
    Let $u$ and $v$ be the two earliest arriving inputs
    $w$ = $u \cdot v$
    /* substitute $w$ into $f$ producing $g$ */
    $g$ = SUBSTITUTE($f$, $w$)
    compute arrival time of $w$
    NAND_DECOMP($g$)
}
```

Figure 3.1: Decomposition procedure for a NAND-NAND representation

affected by the simplification of the function. An interesting fact is that simplification also helps to reduce the delay of the two-level NAND-NAND decomposition when all the inputs have the same arrival time.

Theorem 3.1.1 *A minimum literal SOP representation of a function also provides a minimum depth solution when the implementation is a decomposition of the* NAND-NAND *structure into 2-input gates and all inputs arrive at the same time.*

Proof The proof is based on combinatorial merging [22] and the optimum decomposition procedure AND_OR_DECOMP. The combinatorial merging algorithm constructs a weighted r-ary tree that minimizes the weight of the root from a given set of integer weighted leaves when the weight of a node is 1 + the maximum weight of its fanins. In fact, if C is the set of leaves and $w(c)$ is the weight of a leaf $c \in C$, then the minimum weight that can be assigned to the root of any r-ary tree is

$$\lceil \log_r \sum_{c \in C} r^{w(c)} \rceil$$

A greedy, recursive procedure of combining the leaves with lowest weight provides the optimum r-ary tree. According to the AND_OR_DECOMP routine, the function f is re-expressed as a NAND-NAND function, and then optimum decomposition of each cube followed by the decomposition of the root NAND function yields the best decomposition.

Consider a function f with n inputs and described as m cubes, $\{c_1, \ldots, c_m\}$ of sizes $n_1, \ldots, n_m$ respectively. Let $N = \sum n_i$. The arrival time at the output of cube c_i (measured in terms of the number of levels of 2-input gates) is simply $\lceil \log(n_i) \rceil$ since the arrival times of the inputs are the same (w.l.o.g. assumed to be 0). The arrival time at the output of f can be computed using the combinatorial merging algorithm to be

$$A(f) = \lceil \log_2 (\sum_{j=1}^{m} 2^{\lceil \log(n_i) \rceil}) \rceil$$

Simple manipulation yields

$$\log(N) \leq A(f) < 1 + \log(N)$$

It is clear that the minimum value of N, the number of literals in the SOP representation of f, also results in the minimum depth decomposition of the NAND-NAND representation of f. ∎

The result of Theorem 3.1.1 is not widely applicable due to the following assumptions made in the proof,

1. the arrival times are equal for all inputs, and,
2. the function is implemented as a two-level NAND-NAND structure.

When the arrival times differ, the simplification problem for minimum delay (still assuming a two-level tree implementation) is not the same as minimizing the number of literals. The following function has 10 cubes containing 44 literals.

$$o = \overline{a}c\overline{d}e + a\overline{b}\overline{c}d\overline{g} + a\overline{b}\overline{d}eg + \overline{a}bcdg + \overline{a}b\overline{c}\overline{d}g + \overline{a}b\overline{d}f\overline{g} + \overline{a}\overline{b}\overline{c}df + \overline{a}c\overline{d}\overline{g} + \overline{b}c\overline{d} + \overline{b}cf$$

By choosing input c to be late arriving, there are 8 cubes that are affected. If the arrival time of c, $a(c)$, is chosen to be 10 and all other inputs arrive at 0 then $a(o) = 15$ under the assumption that a 2-input gate contributes 1 unit of delay . Another prime and irredundant representation for the same function is shown below. It contains 45 literals and differs from the previous one only in the first cube.

$$o = \overline{a}b\overline{d}eg + a\overline{b}\overline{c}d\overline{g} + a\overline{b}\overline{d}eg + \overline{a}bcdg + \overline{a}b\overline{c}\overline{d}g + \overline{a}b\overline{d}f\overline{g} + \overline{a}\overline{b}\overline{c}df + \overline{a}c\overline{d}\overline{g} + \overline{b}c\overline{d} + \overline{b}cf$$

In this representation only 7 cubes are affected by c. As a result $a(o) = 14$. Thus the representation with fewer literals may not yield a 2-input NAND-NAND decomposition with smaller delay.

Even though simplification for minimum delay is difficult, it is assumed that a small area solution will also provide a small delay implementation. This is the guiding principle used to simplify each node in the Boolean network. To achieve a small representation the minimizer is provided with the largest possible don't care set that results from the structure of the logic surrounding the node to be minimized. An appropriate choice of the don't care set can also result in a reduction in the circuit delay.

Using the Satisfiability Don't Care conditions may increase the levels in the circuit. This occurs when the simplification process causes a Boolean resubstitution. If the don't care set is appropriately "filtered" the simplification guarantees no increase in the number of levels. This is done by excluding from the computation of the don't care set the gates at the same level of the node being simplified. This will guarantee that the level of the node does not increase. This technique was proposed in [68].

Using the Observability Don't Care conditions can represent the function at a node in terms of signals with smaller arrival times, thereby reducing the circuit delay. Due to the presence of reconvergence, a function may be replaced by an alternate function that is "permissible". The choice of the permissible function is based on the one that will result in a reduction of delay. Earlier work of [9] lays the groundwork for the use of permissible functions to reduce delay. Recent work [57] makes these ideas practical on large circuits. For the specified region it is easy to generate all the supports that can be used to represent the function. A support containing few signals and early arriving signals is chosen to implement the function. The complexity of the function that results from the choice of a particular support is not easily determined. However, the heuristic works well in practice. As contrasted with other techniques, this transformation changes the local function computed by the selected region even though the overall circuit function is unchanged. In this work we use local transformations that do not modify the global function at a node and so are unable exploit the ODC to simplify the logic of the selected region.

Simplification is used to get a small representation for the function of the region selected for transformations. Since, decomposition based transformations principally work on a SOP representation of the function, and a small representation is useful in reducing both the area and delay, the simplification step is important.

3.1.2 Timing-driven decomposition

As was seen in Section 2.3 the choice of sub-expressions to extract affects the circuit delay. The heuristic used to generate a decomposition that has small delay is to place the late arriving signals closer to the output. By keeping the early arriving signals further from the output, the computation of part of the function can proceed before the late arriving signal is available. The procedure for doing so is described in [61]. The decomposition routine evaluates the divisors of the function according to the arrival time of its inputs and the area saving that would result from extracting that divisor. The divisor with the earliest arriving signals is extracted. The divisor is recursively decomposed following which the original function is decomposed. The bottom-up decomposition procedure is illustrated in Figure 3.2. Thick lines represent critical signals and dashed lines represent signals whose arrival time is not known. If at any stage of the recursion the function has no divisors, it is

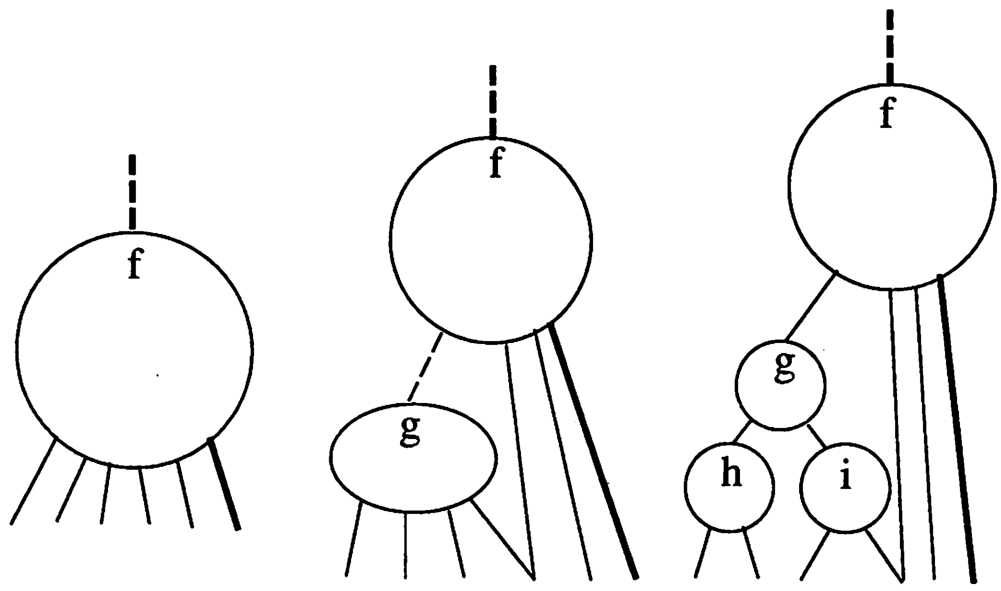


Figure 3.2: Timing-driven decomposition

decomposed using the procedure AND_OR_DECOMP described in Figure 3.2.

For timing driven decomposition (TDD), the choice of divisors that are evaluated affects the quality of the decomposition. In this work only algebraic divisors are used since they are easy to enumerate. Algebraic divisors such as kernels [50] or two-cube divisors [71] are possible candidates for extraction. The generation of divisors (kernels or 2-cube divisors) is based on the sum-of-products representation of the function. As the starting point, a minimum area representation is used. Based on whether kernels or 2-cube divisors are used, the local transformation based on timing driven decomposition is called **TDD-kernel** or **TDD-2cube**.

The use of 2-cube divisors overcomes the limitation that some functions have an exponential number of kernels. The number of 2-cube divisors, on the other hand, is quadratic in the number of cubes of the function. In our system both techniques may run into limitations — run-time limitation for kernels due to the possibility of combinatorial explosion and memory limitation for 2-cube kernel extraction due to the poor choice of data structures in the current implementation. A timeout feature prevents a single evaluation of

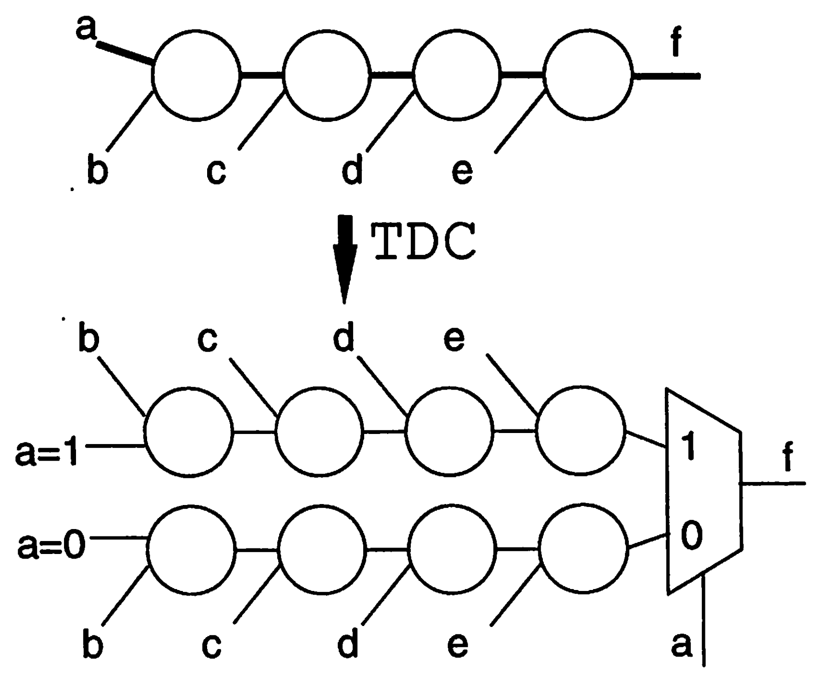


Figure 3.3: Timing-driven cofactoring

a local transformation from stalling the entire optimization.

3.1.3 Timing-driven cofactoring

A transformation frequently used by designers is to use the latest arriving signal as the control of a multiplexor as shown in Figure 3.3. In implementing this transformation there is a considerable overhead in area since the structure of the circuit is duplicated. However, it may be possible to share the logic between the two cofactored structures. Work on the use of the cofactoring transformation includes that of [6] and recent generalizations of the cofactoring transformations proposed in [23]. The generalized timing driven cofactoring, **TDC**, proposed in [23] clusters inputs with similar arrival times and uses them as the control of multiplexor logic. It overcomes the restriction that only one signal can be used as the select input of the multiplexor.

3.1.4 Generalized bypass transformation

Another local transformation is the **generalized bypass transformation** (GBX) described in [45]. The basic idea is to alter the circuit structure as shown in Figure 3.4 so that transitions do not propagate along the long paths in the circuit. To achieve this, the condition under which the function, f, depends on the critical input, a, is determined. This is the Boolean difference of f with respect to a.

$$\frac{\partial f}{\partial a} = f_a \oplus f_{\bar{a}}$$

Under this condition the output follows the input (possibly with an inversion). Hence by using the Boolean difference as the control input of a multiplexor, the function can be either the critical input (when the Boolean difference is a 1) or the precomputed function (precomputed for the critical signal value to be a 1 or 0 since the function is independent of the critical signal value). In order to set the critical input to a constant along the "0" path of the multiplexor, some gates may need to be duplicated if there are fanouts along the path.

3.1.5 Using the complement of the function

The techniques for extracting common-subexpressions are based on a sum-of-products representation of the function. Since it is not known a priori if the function or its complement has a decomposition with smaller delay, it may be beneficial to try the extraction techniques on the complement. As an example consider the function F and its complement $\bar{F}$ shown below. The subscript on an input variable indicates a non-zero arrival time for the input e.g. a_3 indicates that input a arrives at time 3.

$$F = a_3 b_3 c' d' e + a_3 f_1' c' d' e + b_3 d' e g_1 + d' e f_1' g_1';$$
$$F = (a_3' g_1' + c g_1 + b_3' f_1 + d + e')';$$

For F, the representation of the complement can be implemented in smaller delay (arrival time at output is 6) than the original function (arrival time is 8). The use of the complement can be exploited for any technique that depends on the subcircuit function like timing-driven decomposition. Techniques like GBX and TDCthat depend on the circuit structure do not make use of this flexibility.

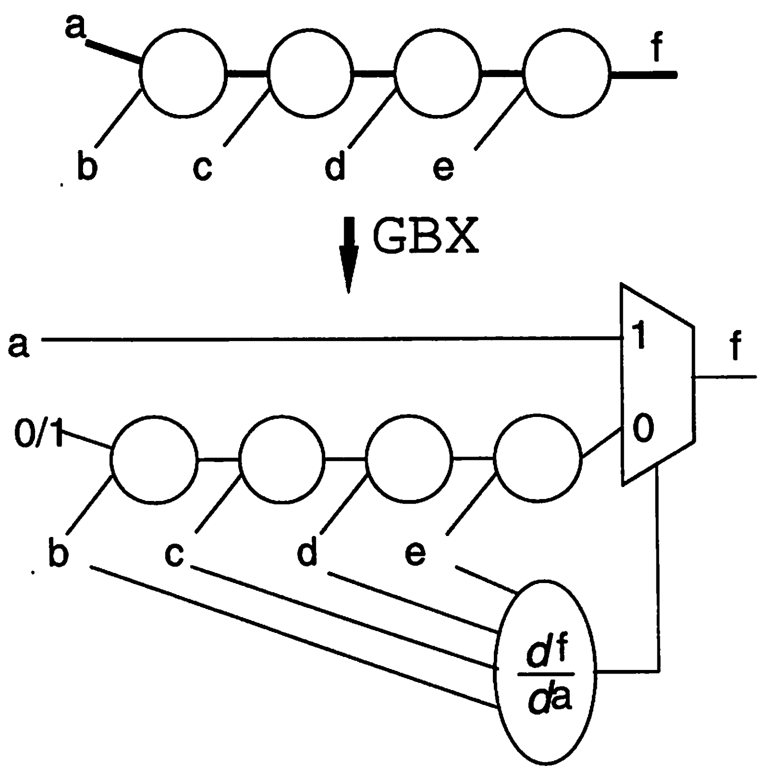


Figure 3.4: Generalized bypass transformation

3.2 Using local transformations to reduce delay

The previous sections describe some of the common technology-independent transformations that can be used to improve the delay locally. These transformations are local in their scope since they operate on a part of the network. However the "degree" of locality depends on the size of the region being transformed. For these techniques there is usually an increase in the area of the transformed region. The natural question to ask is **"How can local transformations be applied to ensure a reduction in the delay of a circuit at a minimal area increase ?"**. Before attempting to answer this, it is useful to note the fundamental difference between area and delay in terms of the type of objective function being optimized. Area provides an additive objective function. By that we mean that if two circuits are combined then the area of the resultant circuit is the sum of the individual areas. This property allows us to optimize part of a circuit and be sure that the area savings will be reflected as savings for the entire circuit. Delay, on the other hand, yields a non-linear objective function. For example, on combining two circuits in parallel, the delay is the maximum of the two. The implication is that we cannot focus on only some part of the circuit and improve the delay in that region. The improvement achieved there may or may not be reflected as a reduction in circuit delay.

Applying local transformations to reduce delay is a widely used technique among circuit designers. The earliest applications of local transformations was in rule-based systems like SOCRATES [5]. This approach had the advantage that the rules could be customized for a certain design style and technology. However finding the sequence of rules to apply was difficult and, in addition, the rules had to be modified for any change to the technology. An alternative to the rule-based approach was proposed in the Yorktown Silicon Compiler [12]. The notion of a *critical-network* was introduced. The critical-network represents the most critical portions of the logic. The observation was made that if delay reducing transformations are applied to a set of nodes that cut all input-output paths in this critical network, the circuit delay is guaranteed to be reduced. Following up on these ideas, researchers proposed heuristics to find separator sets at which specific delay reducing transformations were applied [61, 45]. The nodes in the critical network are assigned weights that represent the amount of improvement possible and a minimum-weight cutset procedure is used to determine the separator-set. However, since the improvement through a separator-set is related to the minimum weight on it and not on the cumulative weight, using a minimum-weight cutset procedure is not a meaningful way of identifying a good choice of nodes to transform. Furthermore, each method was limited to using a specific transformation.

Performance optimization procedures that are based on the concept of improving the delay at a set of nodes follow the outline of the the algorithm in Figure 3.5. This is a generic algorithm that can be customized to yield specific algorithms by changing steps 1 and 2 of the optimization loop, or by using different optimizations in step 3. In particular the algorithms of [12, 61, 17, 45] can be described using this generic procedure GENERIC_PERF_OPT.

The following sections are devoted to discussing in detail how a powerful optimization system can be built by careful customization of GENERIC_PERF_OPT. The optimization procedure will be flexible to allow the use of different local transformation in

```
GENERIC_PERF_OPT($\eta$) {
    repeat {
        1.  Determine critical region
        2.  Select separator-set of critical region
        3.  Apply selected local transformations
    } until (constraints violated AND delay decreases)
}
```

Figure 3.5: Generic optimization procedure

a unified manner. The local transformations will be used to derive a lower bound on the improvement possible during one iteration. The possible improvement determines the region of the circuit that should be improved. More importantly, the predicted improvement can be achieved at a small cost in circuit area by a selection procedure that uses similar arguments as the lower bounding technique. By choosing different lower-bounding techniques and corresponding selection strategies powerful heuristics can be built to reduce the circuit delay with a small increase in circuit-area.

3.2.1 Computing a lower bound on delay improvement

Associated with a network η are a set of primary outputs $\{O_1, \ldots, O_k\}$. The primary outputs are ordered in increasing values of their slacks, i.e. $s(O_1) \leq \ldots \leq s(O_k)$. The circuit fails to meet specification if $s(O_1) < 0$. O_1 is called the most critical output and its slack (which is negative) determines the amount by which the circuit needs to be sped up. It is clear that the critical paths to O_1 should be sped up. What about the other outputs? Imagine that the second output O_2 has almost the same slack as the most critical. If we were to concentrate on the most critical and improve it substantially, the circuit performance would be determined by O_2. Since it had a slack almost similar to O_1 the circuit performance would be virtually unchanged. This section examines procedures that determine the outputs that have to be sped to ensure a reduction in circuit delay.

Assume that there are p delay reducing transformations $T^1, \ldots, T^p$ available to the designer. At each node n in the network, the application of T^i results in a delay saving $D(n, T^i)$ and an area penalty $A(n, T^i)$. To evaluate the effects of a transformation applied at a node, a region of the circuit rooted at the node is transformed. The selection of this region may be specific to each transformation or may be determined by some simple method (all nodes in the transitive-fanin of the node up to a specified distance). Section 3.2.2 examines the issues involved in determining the region that is transformed. A transformation $T^*(n)$ is determined to be the best transformation at node n. To determine $T^*(n)$ the designer chooses the criterion to evaluate the transformations. It may be the transformation that provides maximum decrease in delay or one that provides the most saving per unit increase in area. The choice made is a greedy one, i.e. the transformation that best meets the chosen criterion is the representative one at the node.

For the *benefit* criterion (one that provides maximum improvement in delay), the best transformation $T^*(n)$ at node n has

$$\begin{array}{ll} D(n,T^*(n)) \geq D(n,T^i) & \forall i \in \{1,\ldots,p\} \\ A(n,T^*(n)) \leq A(n,T^i) & \forall i \text{ such that } D(n,T^*(n)) = D(n,T^i) \end{array}$$

The maximum reduction in delay possible at a node n is thus $D(n,T^*(n))$ and is denoted by $D(n)$.

Another alternative is to select the transformation that has the best *benefit/cost* measure. This too represents a greedy choice of the transformation that provides the maximum improvement in delay per unit increase in area. For this criterion the best transformation $T^*(n)$ is

$$\begin{array}{ll} D(n,T^*(n)) > 0 & \\ \frac{D(n,T^*(n))}{A(n,T^*(n))} \geq \frac{D(n,T^i)}{A(n,T^i)} & \forall\, i \in \{1,\ldots,p\} \end{array}$$

Whatever criterion is used to determine the best transformation at a node, the assumption is that this represents the only possibility at the node. By keeping the best transformation at the node as the representative one, the combinatorial explosion that occurs when considering all possibilities is avoided. By knowing the amount of saving that can be achieved at each node, it is possible to compute the guaranteed saving at the outputs. The computation is based on the assumption that the transformations selected for different nodes do not influence the savings that can be achieved at each of those nodes. For delay models that ignore fanout, such as the number of levels in a bounded-fanin circuit, this assumption is always valid. For delay-models where fanout is of importance, methods to overcome this assumption will be described in the next chapter.

Proposition 3.2.1 *The arrival time for node i can be reduced* at least *by $\delta(i)$, determined by the recursive formula*

$$\delta(i) = \begin{cases} 0 & \textit{if } i \textit{ is a primary input} \\ \max[\, D(i),\ \min\limits_{j \in FI(i)} \{s(j,i) + \delta(j)\} - \min\limits_{j \in FI(i)} s(j,i)] & \textit{otherwise} \end{cases} \tag{3.1}$$

where $D(i)$ is the delay saving at node i and $s(j,i)$ is the slack along the edge from node j to i.

Proof The proof is by induction on the level of nodes in the circuit in depth-first-traversal order from the output.

Base case: For a node i at level 0 there is no possible improvement in its delay since it is a primary input. Hence $\delta(i) = 0$.

Induction Step: We will show that if nodes with level $n-1$ or less have delay improvements of $\delta()$, then the possible improvement at any node at level n is given by the above equation.

Consider node i at level n with arrival time at its output equal to $a(i)$ and required time $r(i)$. Let j and k be the only two inputs of i (for nodes with more than two inputs a similar argument is applicable). Then $a(i) = \max[a(j) + d(j,i), a(k) + d(k,i)]$. Since j and k have level $n-1$ or less, the maximum improvement in their delay is $\delta(j)$ and $\delta(k)$ respectively (induction hypothesis). For i, the new value of the arrival time is $a'(i) = \max[a(j) - \delta(j) + d(j,i), a(k) - \delta(k) + d(k,i)]$. The saving in delay at node i is thus $a(i) - a'(i)$. This is equal to

$$\max[a(j) + d(j,i), a(k) + d(k,i)] - \max[a(j) - \delta(j) + d(j,i), a(k) - \delta(k) + d(k,i)]$$

Since $r(i) - d(j,i) - a(j) = s(j,i)$ (similar condition for input k), the improvement can be re-expressed as

$$\max[r(i) - s(j,i), r(i) - s(k,i)] - \max[r(i) - s(j,i) - \delta(j), r(i) - s(k,i) - \delta(k)]$$

which reduces to

$$\min[s(j,i) + \delta(j), s(k,i) + \delta(k)] - \min[s(j,i), s(k,i)]$$

This gives us the saving at i if no transformation at node i is selected. In case it is selected, the saving at the output of i is at least $D(i)$ (not accounting for additional savings at nodes

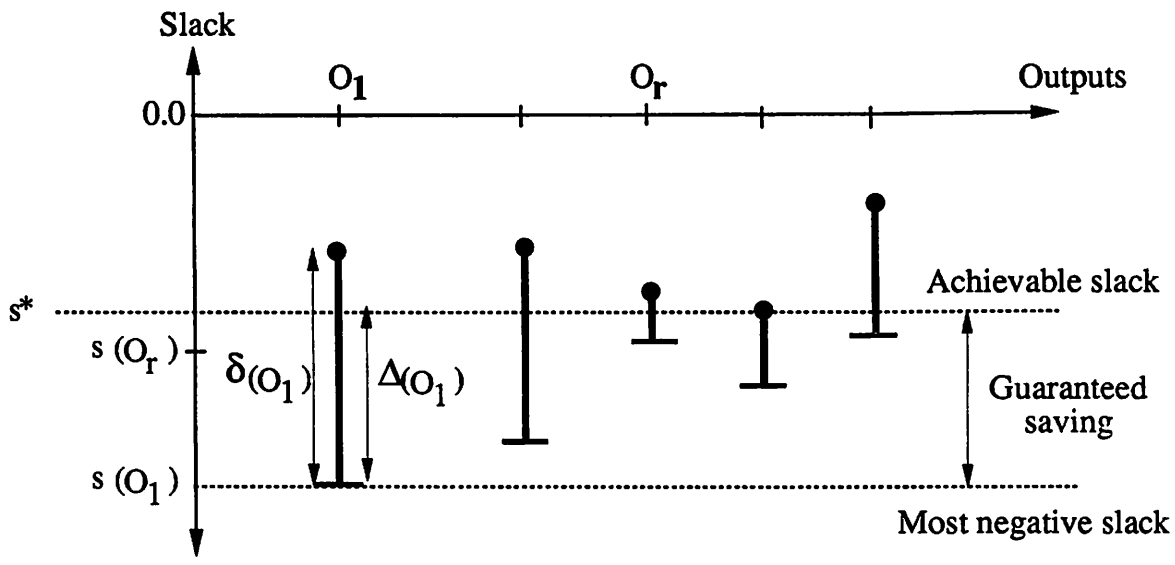


Figure 3.6: Determining the maximum delay improvement

in the transitive fanin). Since the reduction $\delta(i)$ is either due to the transformation or due to improvements at the inputs, the choice is made depending on which provides the greater saving. This results in computing $\delta(i)$ that is the same as in Equation 3.1. ■

Once the delay saving at the outputs is known, we can find out the slack s^* that can be achieved.

$$s^* = \min_{i=1,\dots,k}(s(O_i) + \delta(O_i)) \tag{3.2}$$

This is illustrated in Figure 3.6. To achieve this slack the arrival time at output O_i should be reduced by an amount $\Delta(O_i) = s^* - s(O_i)$. Note that the arrival time can be reduced by this amount since $\Delta(O_i) \leq \delta(O_i)$. To reiterate, Δ represents the ampont of saving that is desired at a node and δ represents the saving that can be achieved at a node.

A drawback of the above computation is that every node in the network needs to be visited to evaluate the savings in delay. For large circuits this can be very time-consuming. To overcome similar limitations, the concept of an ϵ-critical-network was introduced in [12]. The ϵ-critical-network consists of all the nodes and edges in the Boolean network that have a slack within ϵ of the minimum slack (ϵ is a user-specified constant) The nodes in the

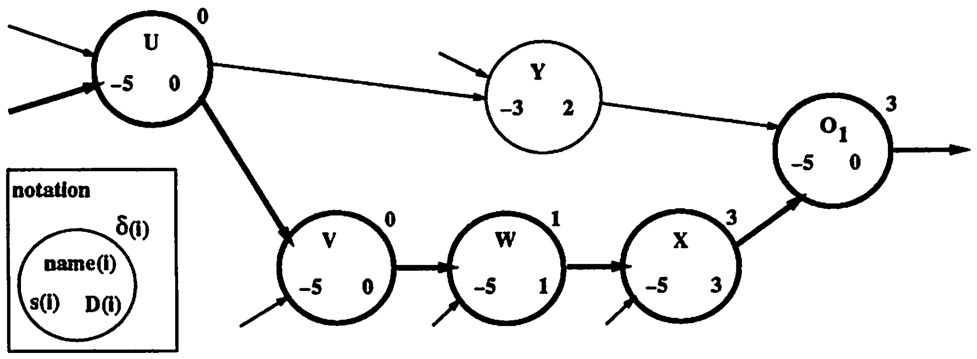


Figure 3.7: Using a fixed threshold to determine critical signals

ϵ-critical-network may be regarded as the part of the circuit that needs to be improved to reduce the circuit delay. The local transformations can be evaluated for all the nodes in the ϵ-critical-network to determine the value of s^*. Unfortunately, as illustrated by the example in Figure 3.7, a fixed value of ϵ, smaller than $-s(O_1)$, may yield a target value s^* that may not be achievable. The nodes in the network are shown along with their slack, the improvement due to local transformation, and the saving that can be achieved up to that node.

The ϵ-critical-network is shown in bold for $\epsilon = 1$. Since the most critical output, O_1, has a slack equal to -5, all nodes with a slack less than or equal to -4 are part of the ϵ-critical-network. Traversing the edges in the ϵ-critical-network, $\delta(O_1)$ is computed to be 3 (which exceeds the value of ϵ). The value of the achievable slack, s^*, is therefore $s^* = -5 + 3 = -2$ (from Equation 3.2). When the selected transformation is applied at node X, the new slack at O_1 is only -3 (determined by Y). The reason for this is that node Y was not ϵ-critical and so not evaluated for local transformations. The above situation arises only when the saving, $\delta(O_i)$, for some output O_i exceeds the value of ϵ chosen to determine the ϵ-critical-network. Rather than use a fixed value for ϵ, we can determine its value based on the delay information in the network. This overcomes the limitation of using a fixed value. For the example at hand, the value of ϵ can be modified to be $\delta(O_1)$ generated during the first pass. Now node Y is also included in the ϵ-critical-network. A second pass of computing the δ-values gives new values for the savings δ' at each output. The computed

```
DETERMINE_MAX_IMPROVEMENT($\eta$) {
    $\epsilon$ = user-defined small constant
    pass = 1
    $s^* = s(O_1) - \epsilon$
    do {
        for each primary output $O_i$ of $\eta$ with $s(O_i) \leq s^*$ {
            compute $\delta(O_i)$ using Equation 3.1
            $s^* = \min[s^*, s(O_i) - \delta(O_i)]$
        }
    } while ((pass++ < 2) OR ($s^* - s(O_1) > \epsilon$))
}
```

Figure 3.8: Procedure to determine the maximum achievable slack

saving may decrease. The larger value of ϵ results in consideration of paths leading to the output O_1 along which less improvement is possible. In the circuit of Figure 3.7, the value of $\delta'(O_1)$ reduces to 2 (being limited by the improvement possible at node Y) and the maximum achievable slack, s^*, is now -3. This example illustrates how a two-pass algorithm that uses an initial guess of ϵ can be used to determine the value of s^*. The procedure for computing the maximum achievable slack is called DETERMINE_MAX_IMPROVEMENT and is described in Figure 3.8. The advantage of this procedure to compute s^* is that only the nodes along the most critical paths need to be evaluated.

Proposition 3.2.2 *Procedure DETERMINE_MAX_IMPROVEMENT generates the same value of s^* as that achieved by application of Equation 3.1 and Equation 3.2 over the entire circuit.*

Proof The proof proceeds by analyzing the two passes of the algorithm along with the observation that nodes that are not considered are not relevant to the optimization. At the end of the first pass let the value of the achievable slack be s_1^*. This results in considering all the nodes, j in the second pass that have a slack $s(j) \leq s_1^*$. Computation of the new values for δ's at the outputs can only yield a smaller value (since more fanins of a node will be considered during the computation of the δ for a node, and the contribution of the inputs proceeds through a min function, this contribution can only decrease). As a result the final $s^* \leq s_1^*$. Any node i not considered in the second pass has a slack $s(i) > s_1^* \geq s^*$. For these nodes no improvement is required since they will not be the most critical nodes even after the transformations are carried out. Thus by not considering the nodes with $s(i) > s^*$, the same value of s^* will be computed as would be if they were considered. ■

The value of s^* provides a target that can be achieved by applying selected transformations. This value is also used to label nodes as being **relevant** or **non-relevant**. A node n is said to be relevant if $s(n) < s^*$. An edge from i to j is relevant if $s(i,j) < s^*$. A **relevant network** consists of all the relevant nodes and edges. It should be noted that the relevant network is defined with respect to some slack threshold, s^*. Computation of s^* assumes that the delay improvement due to local transformations is known. All the transformations described in Section 3.1 restructure a selected region in a manner so as to reduce its arrival time. The next section addresses the question, **"How to select the region to be transformed ?"**.

3.2.2 Selecting a region to transform

The motivation for applying local transformations along the critical paths is to reduce the length of the the critical paths. For transformations to result in local improvement they must be applied to an appropriate region. In the case of rule-based systems [5] this region is clearly defined as the one on which the rule is applicable. For systems that develop transformations on the fly there is a great deal of freedom in selecting the regions. We will call the region on which a transformation is applied the **scope** of the transformation. In [61], the scope was the set of nodes lying in the transitive fanin, and along the critical paths, within a distance d of the node under consideration. This characterization of the scope is called the *d-critical-fanin-section* of the node or simply the *critical* strategy. The parameter d provides control of the tradeoff between the run-time and quality. A larger region has more options to consider leading to possibly better decompositions at the expense of longer time required to evaluate the choices. The *critical* strategy is based on the idea that a portion along the critical path could perhaps be reduced to a smaller length, thereby providing some reduction in delay.

The example of Figure 3.9 illustrates the shortcoming of focusing only on the critical path. The example circuit computes the conjunction of four inputs that arrive at

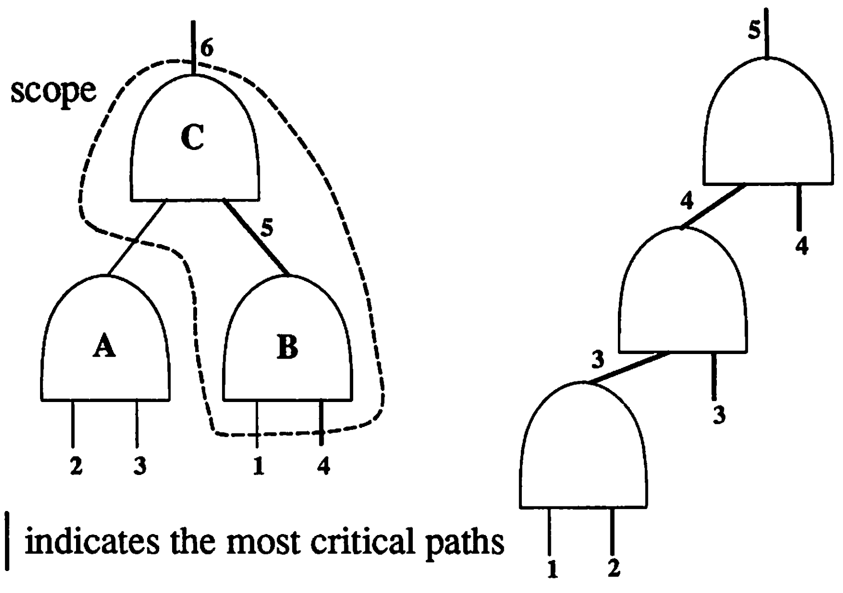


Figure 3.9: Example of the scope of a transformation

times 1, 2, 3 and 4. Each 2-input AND gate is assumed to have a delay of one. The arrival time at the output is therefore 6. The critical path consists of nodes B and C. Under the *critical* strategy to determine the scope we collapse these nodes. This results in a 3-input function that has inputs arriving at times 1, 4 and 4. No matter how this is decomposed into 2-input gates the output has an arrival time of 6. However, by also considering node A as part of the scope the output arrival time can be reduced to 5. This is achieved by the decomposition shown on the right.

One way to choose the scope of a transformation is to perform an exhaustive search among the possible regions. This is clearly impractical. The notion of depth appeals to intuition as a natural way of restricting the size of the region while simultaneously providing the transformation a reasonable portion of the critical path to reduce. One might choose to transform the entire *transitive* fanin region up to a specified distance d. This however results in large functions. Furthermore, when the non-critical region consists of good Boolean factors that are unlikely to influence the delay, this strategy may lead to poor decompositions when only algebraic techniques are used.

The following strategy, *compromise*, is suggested as an intermediate strategy that generates a scope whose size may be greater than the *d-critical-fanin-section* and may be as large as the *transitive* scope. It is based on the distribution of arrival times and in extreme cases may be reduced to either the *critical* or the *transitive* strategy. The idea is to include non-critical nodes that may lead to a better decomposition while neglecting nodes that appear to have been combined well. Record the minimum arrival time $\underline{a}$ for any input to the critical path up to a depth d from the root node n. Then include any non-critical nodes that are in the transitive fanin of n and within a distance d of n and have inputs that arrive at or later than $\underline{a}$. Nodes in the transitive region that have arrival time smaller than $\underline{a}$ are already combined with early arriving inputs. This is in accordance with the guiding heuristic for performance optimization (keep late arriving signals closer to the output) and so there is no benefit from including these in the region to be transformed (we may in fact lose if we are unable to recover the good Boolean factors).

If an algorithm exists for the optimum decomposition of a function then, by using the *transitive* strategy with unlimited depth, a minimum delay implementation can be obtained. As an example of an optimum decomposition technique, [75] provides an algorithm to decompose an AND function with arbitrary arrival times at its inputs into the optimum tree of 2-input gates. Thus to optimize an arbitrarily structured AND tree, one needs to simply apply the decomposition technique at the primary outputs with unlimited scope. Rather than consider the entire tree, is there a strategy that yields the optimum solution looking only at an appropriate sub-tree? For the example in Figure 3.9 the *compromise* strategy indeed yields the optimum solution. However, the *compromise* strategy is not adequate in all cases. Figure 3.10 shows a family of circuits realizing an $(\frac{3n}{4}+1)$ input AND function with one input arriving at m, $\frac{n}{4}$ inputs arriving at 1 and the remaining $\frac{n}{2}$ inputs arriving at 0, for different $m > 3$ and $n = 2^m$. For this family of circuits, the *compromise* strategy applied to the primary output does not yield an optimum scope. The scope generated using the *compromise* strategy with unlimited depth consists of the shaded nodes. For this example, the *critical* strategy generates the same scope. Redecomposition of this scope leads to an overall delay of $(m+2)$. On the other hand, the optimum decomposition has a delay of $(m+1)$.

Figure 3.9 and Figure 3.10 are circuits with tree topologies where the optimum decomposition can be found only if the entire tree is collapsed. This is rather disheartening since it suggests that local transformations may need to be at the scale of the entire circuit

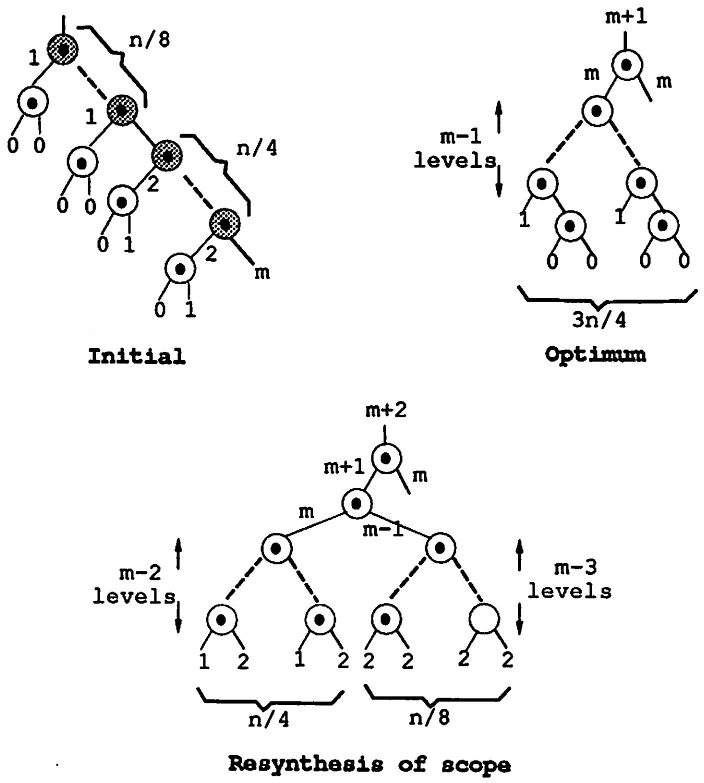


Figure 3.10: Example illustrating the limitation of the selection strategy

to guarantee optimum results, even for simple topologies like trees. On the other hand, we do not wish to make all the improvement in a single pass. We can reduce the delay in small steps by iteratively applying transformations. In the previous section we computed the slack s^* that can be achieved via appropriate transformation of regions selected according to some criterion. The question remains "**How to choose, from among the many places where transformations can be applied to yield s^*, sections to transform such that the area increase is minimized ?**".

3.2.3 Selecting local transformations

Once the relevant region of the circuit has been identified, appropriate transformations need to be selected which will yield the desired improvement in circuit performance.

The choice of transformations must be made carefully taking into account factors such as the area increase and the interaction between the transformations. The area increase must be kept small. A larger area leads to a larger layout with longer wires that cause an increase in delay. Even though the interaction between layout and the logic is not considered explicitly, it is important to look ahead so as not to put the layout tools in a position where they have to improve a bad design. Another important consideration is the interaction between the various transformations. In an ideal situation we assume that the transformations do not affect each other. At the technology-independent stage, the unit-delay model ignores fanout and so there is no interaction between transformations. However, in mapped circuits this is not the case and we e will revisit the selection procedure in Section 4.3 to consider the interaction between the transformations.

The amount of saving, δ, that can be achieved at an output differs for different outputs. The amount of saving desired at output O_i is given by $\Delta(O_i)$

$$\Delta(O_i) = (s^* - s(O_i)) \leq \delta(O_i) \tag{3.3}$$

It is thus clear that for each output there exists a choice of transformations that will produce the required saving at that output. There may be many possible alternatives and we are interested in choosing one with the minimum area increase. The problem of selecting local transformations to meet a target delay is defined below.

XFORM_SELECT:

> Given a Boolean network η, a constant M, and for each node $n \in \eta$ a delay saving value $D(n)$ and an area weight $A(n)$. Also given are k primary outputs $O_1, \ldots, O_k$, each with a value $\Delta(O_i)$. Find a set of nodes $\mathcal{S}$, with $\sum\limits_{n \in \mathcal{S}} A(n) \leq M$, such that $\forall i = 1, \ldots, k$, the delay at output O_i is reduced by $\Delta(O_i)$.

The set $\mathcal{S}$ refers to the vertices where local transformations are applied to yield the desired savings and is called the **selection-set**. A selection-set is *minimal* if no element of the set can be removed while still being a selection-set. The constant M is the area increase that results from the selected transformations and the optimization problem is interested in minimizing the value of M. The decision problem XFORM_SELECT is NP-hard. This is readily proved by observing that a special case of the problem XFORM_SELECT is the *Basic Circuit Implementation* (BCI) problem [35] which is shown to be NP-hard even for a chain of inverters. It is therefore unlikely that an exact algorithm will be found.

In an attempt to make the problem tractable, researchers imposed the restriction that only the ϵ-critical-network should be improved by finding a set of nodes that would improve all paths in the ϵ-critical-network by at least ϵ. For a directed acyclic graph, a **separator-set** is a collection of nodes such that removing these nodes disconnects every input-output path. A separator-set is a *minimal separator-set* if no subset is a separator-set. The selection procedure in [12, 61] was based on finding the minimum-weight separator-set of the ϵ-critical-network. Nodes in the ϵ-critical-network are given finite weights (that reflect the ease of reducing the arrival time at the node) and the edges are given infinite weights. Since a separator set of the ϵ-critical-network corresponds to a finite weight vertex cut of the ϵ-critical-network, a minimum-weight vertex cut can be used as the selection-set. However, there is no guarantee that the proposed improvements will be attained or that the area constraint will be met. Nevertheless, the use of a separator-set was an important contribution for heuristics that followed.

As the next step, to guarantee an improvement of ϵ with the minimum area increase, the node weights were based on the area increase and delay reduction at the node [39]. If the saving required at an output is ϵ, then all nodes in the ϵ-critical-network with a delay improvement greater than ϵ are assigned finite weights equal to the area penalty.

$$w(n) = \begin{cases} \infty & \text{if } s(n) < \epsilon \text{ and } D(n) < \epsilon \\ A(n) & \text{if } s(n) < \epsilon \text{ and } D(n) \geq \epsilon \end{cases} \tag{3.4}$$

Every finite-weight separator-set corresponds to a choice of nodes (each with improvement greater than ϵ) that improve all the input-output paths by at least ϵ. By using the minimum-weight separator-set the area increase is minimized. The drawback of this polynomial-time approach is that the value of ϵ is user-defined and arbitrary. Furthermore, in the case that the ϵ-critical-network does not have a finite-weight cut, this procedure is not useful.

Rather than be limited to a fixed ϵ, we introduce the restriction that the desired savings, Δ's, be computed in accordance with Equation 3.3. This restriction of the general problem is called **RESTRICT_XFORM_SELECT**. By doing this we are assured of finding a selection-set that achieves the predicted delay improvement. The problem is to find a selection-set that does so with the minimum area increase. The selection-set may not correspond to a separator-set of the relevant network (the relevant network is simply the ϵ-critical-network with $\epsilon = s(O_1) - s^*$).

There is a relation between the selection-sets and the separator-sets if the circuit has a tree structure. As was defined earlier, a node or edge is **relevant** if its slack is less than s^*. A path from the input I to an output O is said to be relevant if all nodes and edges along it are relevant. This definition distinguishes between the paths that have to be sped up and those that are not relevant to improving the delay by Δ. To relate the selection-sets and separator-sets, we make use of the fact that in a tree the slack is non-decreasing along any path from the output to an input . This implies that for node y that is a fanin of node x, $s(x) \leq s(y)$. The arrival time for node x can be expressed as

$$a(x) = \max[a(y) + d(y, z), a(z) + d(z, x)]$$

where z is some other input of node x. Since the circuit structure is a tree, every node has one fanout. The equation relating the required times is simply $r(y) = r(x) - d(y, x)$. Combining the two equations we get

$$\begin{aligned} s(x) &= r(x) - a(x) \\ &= r(y) + d(y, x) - \max[a(y) + d(y, z), a(z) + d(z, x)] \\ &= r(y) + d(y, x) + \min[-(a(y) + d(y, z)), -(a(z) + d(z, x))] \\ &= \min[(r(y) - a(y)), (r(y) - a(z) + d(y, z) - d(z, x))] \\ s(x) &\leq s(y) \end{aligned}$$

The monotonicity of slacks is hinged on the fact that each node has a single output which allows us to relate the required times by the simple equation $r(y) = r(x) - d(y, x)$. For a general circuit structure (a directed acyclic graph) the required time relation involves more gates and the monotonicity of slacks cannot be guaranteed.

Theorem 3.2.1 *Given a circuit that has a tree structure and a saving Δ desired at the output O computed according to equation 3.3, for every minimal selection-set S and every relevant path p (composed of nodes with slack $s(O) + \Delta$ or less), $|p \cap S| = 1$.*

Proof

We will show that if $|p \cap S| > 1$, then the selection-set is not minimal. Consider two nodes n_1 and n_2 along any relevant input-output path such that n_1 and n_2 belong to S and n_1 is closer to the output. Since the circuit has a tree structure, $s(n_1) \leq s(n_2)$. Since this path is relevant $s(n_2) \leq s(O) + \Delta$. Also assume that $D(n_1) > 0$ and $D(n_2) > 0$. If this is not the case, then there is no improvement at these nodes and they can be deleted from the selection-set.

There are two possibilities at node n_1. The first possibility is that $s(n_1)+D(n_1) \geq s(O) + \Delta$, in which case the transformation at n_1 is adequate to decrease the delay along p and n_2 can be removed from the selection-set, and $\mathcal{S}$ is not minimal. After removing n_2, $|p \cap \mathcal{S}| = 1$. The other case is when the transformation at n_1 does not provide adequate improvement in delay, that is $D(n_1) < s(O) + \Delta - s(n_1)$. Since the improvement predicted at the output O is Δ, it must be the case that $\delta(n_1) \geq s(O) + \Delta - s(n_1)$ (n_1 is closest to the output and the computation of δ proceeds via a min function). Since $D(n_1) < \delta(n_1)$, it follows that for all inputs k to n_1, $\delta(k) \geq s(O) + \Delta - s(n_1)$. Hence all relevant paths from the inputs to n_1 contain at least one element which provides the adequate improvement. Since, all paths from the input up to n_1 must have this property, the transformation at n_1 can be removed from the selection-set and output O will still be improved by Δ. In this case as well $\mathcal{S}$ is not minimal and after removing n_1 from $\mathcal{S}$, $|p \cap \mathcal{S}| = 1$. ■

Applying the above theorem it can be shown that a minimal selection-set is a minimal separator-set of the relevant-subnetwork. Since $|p \cap \mathcal{S}| = 1$, removing the nodes in the selection-set will disconnect every relevant path at exactly one point. Thus the selection-set is also a separator-set.

When the circuit has multiple outputs each of which terminates a tree, and the amount of saving desired at each output is different, we conjecture that the problem of finding the selection-set with the minimum area increase is NP-hard. Intuitively, this is to be expected since the choice of selection that is best for a particular output may be incompatible with the best choices for other outputs.

When all outputs terminate trees, if all $\Delta(O_i)$ are equal and determined by Equation 3.3, the problem can be solved exactly in polynomial-time using a flow algorithm. This algorithm is illustrated in the procedure EQUAL_SAVING_PROC of Figure3.11.

Proposition 3.2.3 *The procedure EQUAL_SAVING_PROC finds the set of transformations $\mathcal{S}$ which achieves a delay saving Δ with the minimum area increase when all relevant outputs terminate trees and have to be improved by Δ.*

Proof Finite weights (equal to the area saving) are assigned to the nodes of the relevant network that can potentially achieve a saving of Δ. The nodes that are not relevant have a weight 0 and do not affect the size of the minimum-weight separator-set. The irrelevant nodes that are part of the separator-set are removed at the end to generate the selection-set.

```
EQUAL_SAVING_PROC($\eta$, $\Delta$) {
    for each vertex $n$ of $\eta$ {
        $weight(n) = \begin{cases} 0 & \text{if } s(n) \geq s(O_1) + \Delta \\ \infty & \text{if } s(n) < s(O_1) + \Delta \text{ and } D(n) + s(n) < s(O_1) + \Delta \\ A(n) & \text{if } s(n) < s(O_1) + \Delta \text{ and } D(n) + s(n) \geq s(O_1) + \Delta \end{cases}$
    }
    $\mathcal{S}'$ = min-weight separator-set($\eta$)
    $\mathcal{S} = \mathcal{S}' - \{x | s(x) \geq s(O_1) + \Delta\}$
}
```

Figure 3.11: Selecting transformations for the special case

Since each output terminates a tree, according to Theorem 3.2.1 every relevant path must have on it at least one node with finite weight. If this is not the case, then a delay saving of Δ cannot be achieved at the corresponding output (the propagation of δ proceeds along a min function). Hence, a finite-weight separator-set exists for the multi-output network. All the nodes of a minimum-weight separator-set have the property that $D(n) + s(n) \geq s(O_1) + \Delta$, which results in the desired saving of Δ at all outputs. Since, any finite-weight separator-set is adequate in reducing delay by Δ, and the weight of the separator-set is simply the area increase, a minimum-weight separator-set provides the desired saving with minimum area increase. ∎

Since the general problem is conjectured to be NP_hard and the special case is polynomial we look for ways to transform the general case into the special case. To apply the procedure EQUAL_SAVING_PROC to solve SELECT_XFORM, it is necessary to determine the value of Δ that should be used. Obvious ways of selecting Δ are presented as HEURISTICS 1, 2 and 3 and cases in which these are sub-optimum will be presented.

HEURISTIC-1: Consider the minimum improvement possible.

For the relevant outputs, set the smallest delay improvement as the target improvement. Then use the algorithm of Figure3.11 to decide where to apply local transformations. The shortcoming of this approach is that the delay improvement is not guaranteed to be the maximum possible at this stage. This is clarified by Figure 3.6. The most critical output is O_1 and the minimum improvement is predicted for output O_r. If we use a value of

$\Delta = \delta(O_r)$ for the flow algorithm described earlier, a delay reduction of Δ is guaranteed at each output. The smallest slack would then be $s(O_1) - \Delta$ which is less than the maximum value of the minimum achievable slack (s^*). Due to the small amount of saving guaranteed at each step, several iterations may be required to meet the achievable target s^*. As an example, consider the circuit shown in Figure 3.12(a) where O_1 has a slack of -2 and O_2 has a slack of -1. The delay improvement and area penalty of each node is indicated on the figure. Selecting the smallest possible delta will result in the selection-set $\{Y, U\}$. Now output O_2 meets the timing constraint but O_1 does not. So another pass is required. Since the area penalty at V is less than that at W and both will result in meeting the timing constraint, node V is chosen. Thus the area cost is that of nodes Y, U and V namely 30. However, a choice of optimizing node Y and W on the first iteration, reduces the circuit delay by 2 at an area penalty of only 21. There may be some advantage of the heuristic as well. The small choice of step-size results in examining only the most critical paths of the circuit and thus reduces the time spent in evaluating the transformations. In addition, since the step size is small, a smooth tradeoff may be possible.

HEURISTIC-2: Select the delta based on output slacks.

As in the case of HEURISTIC-1, the focus here is on improving the most critical path of the circuit. The most critical output has a slack $s(O_1)$. Determine the output O_i such that $s(O_i) > s(O_1)$ and $s(O_i) - s(O_1)$ is minimum ($s(O_i)$ is the next smallest slack value). Set the improvement to be $\Delta = \min[s(O_1) - s(O_i), \delta(O_1)]$. The idea here is to improve the most critical outputs by just enough till some other output becomes more critical. In this case too, we may use a large number of iterations, each of which decreases the delay by a small amount. The cumulative cost of these may be larger than a single application of a more powerful transformation that reduces the delay substantially at a reasonable cost in area. The circuit in Figure 3.12(b) demonstrates a case when this heuristic performs badly. At each step the delay of the critical path is reduced by 2 which makes the other more critical. Two iterations, the first using $\{U, V\}$ and the second $\{W, X\}$, result in meeting the timing constraints at a cost of 40. The same could have been achieved by an application of transformations at Y and Z with an area penalty of 22.

HEURISTIC-3: Consider the outputs one-at-a-time.

Since outputs need to be improved by different amounts the procedure tries to first satisfy the requirement that each output be improved by the desired amount, and as a secondary objective, will try to keep the area increase small. This is accomplished by using

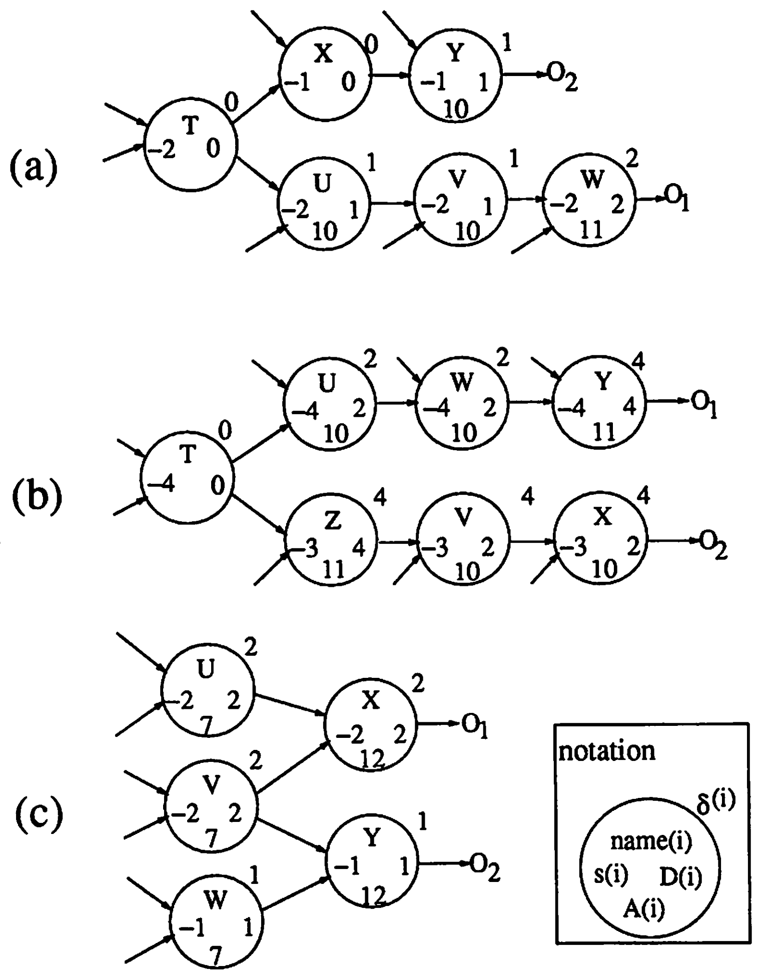


Figure 3.12: Counter-examples to heuristic procedures.

the algorithm of Figure 3.11 on a single critical output, starting from the most critical. For each output only the transitive-fanin needs to be considered as the relevant network. This procedure is guaranteed to produce the maximum value for the minimum slack. However the area penalty may be excessive in this procedure since the choice of the separator-set for a single output may be disjoint from those for the other outputs. The example circuit in Figure 3.12(c) describes a situation for which the area penalty is excessive. The heuristic chooses nodes X for output O_1 and node Y for output O_2 for a cost of 24. However the choice of U, V and W yields the same delay decrease for an area increase of 21.

From the polynomial-time heuristics described above, for the case when the circuit has a tree topology, it is clear that finding a satisfactory solution to the problem RESTRICT_XFORM_SELECT (the problem XFORM_SELECT under the assumption that the selection procedure considers at most one transformation along a path and that the savings desired at the outputs are compatible with Equations 3.3 and 3.2) will require an algorithm that considers different savings at each output and works on the entire network. To develop an exact algorithm we borrow ideas from HEURISTIC-3 where we consider the best selection for a particular output. The natural extension is to consider all selection possibilities for each output and then choose a set of vertices that "covers" at least one selection-set for each output. This procedure is outlined in the algorithm EXACT_SELECT_TREE shown in Figure 3.13.

The procedure EXACT_SELECT_TREE generates all the selection-sets for each relevant output. This is accomplished by generating all the finite weight separator-sets of an appropriate flow network. Then a **selection function** is constructed. A satisfying assignment of this function is an assignment of 1's and 0's to the inputs of this function that result in the function evaluating to be 1. The selection function is defined over a set of variables $x_1, \ldots, x_p$, where variable x_i is associated with node i in the relevant network. A variable x_i equal to 1 in a satisfying assignment implies that the node i has been selected as a candidate for application of a local transformation.

A separator-set that results in the desired saving $\Delta(O_i)$ at output O_i can be represented as a conjunction of the variables corresponding to the nodes in the separator-set. A valid choice of $\mathcal{S}$ must select at least one of the separator-sets for each output. So, first form the functions $F(i)$ as the disjunction of all the separator-sets that result in the desired saving at output O_i. A satisfying assignment of $F(i)$ will ensure that the desired saving is achieved at O_i. Since $\mathcal{S}$ must achieve the desired saving at all the outputs, all the

```
EXACT_SELECT_TREE($\eta$, $s^*$) {
    $\mathcal{F} = 1$
    for each output $O_i$ of $\eta$ with $s(O_i) < s^*$ {
        $flow(i)$ = flow-network for the output $O_i$
        $C_i = GENERATE\_FEASIBLE\_CUTS(flow(i))$
        $F(i) = 0$
        for each separator-set $c_k \in C_i$ {
            $g(i,k) = \prod_{j \in c_k} x_j$
            $F(i) = F(i) + g(i,k)$
        }
        $\mathcal{F} = \mathcal{F} \wedge F(i)$
    }
    $P$ = min-weight-satisfying-assignment($\mathcal{F}$)
    $\mathcal{S} = \{x \mid (x = 1) \in P\}$
}
```

Figure 3.13: Exact procedure for selecting transformations

F's need to be satisfied. This can be done finding a satisfying assignment to the selection function $\mathcal{F}$.

$$\mathcal{F} = \prod_{s(O_i)<s^*} F(O_i)$$

A satisfying assignment of $\mathcal{F}$ is a valid selection-set. However, in order to find the selection-set with the smallest area, a minimum weight satisfying assignment of $\mathcal{F}$ is required. It is clear from the construction of $\mathcal{F}$ that it is a unate function. For the selection function $\mathcal{F}$, satisfying assignments correspond to paths from the root to the "1" vertex in the BDD representation [7]. At a vertex labeled with variable x_i in the BDD, the "1" edge is given a weight equal to the cost of applying the transformation at node i in the network and the "0" edge is given a weight of 0. As was shown in [38], the minimum cost solution to the covering problem defined by $\mathcal{F}$ is given by the shortest path from the root to the "1" vertex. When traversing the shortest path starting from the "1" vertex, we record the variables x_i's that are reached along a "1" edge in the BDD. These variables correspond to the nodes that are selected for transformation.

```
GENERATE_FEASIBLE_CUTS($flow(O_i)$) {
    while (TRUE) {
        $C$ = maxflow($flow(O_i)$)
        if ($C$ has finite weight)
            suppress any node in $C$ /* Set weight = $\infty$ */
        else {
            unsuppress all nodes /* restore original weights */
            suppress nodes that cover all separator-sets found
            $C$ = maxflow($flow(O_i)$)
            if ($C$ has finite weight)
                suppress any node in $C$ /* Set weight = $\infty$ */
            else
                return ; /* no more separator-sets */
        }
    }
}
```

Figure 3.14: Procedure to generate all finite-weight cuts of a flow network

An important step in the procedure EXACT_SELECT_TREE is to find all the relevant separator-sets in the graph. The nodes on the separator-sets of interest must have a delay value $D(n) \geq 0$, as well as $s(n) \leq s^*$ and $s(n) + D(n) \geq \Delta(O_i)$. Enumeration techniques that generate all the cutsets of a graph [69, 2] are not suitable to generate the limited number of acceptable separator-sets. In addition, the procedure should be able to generate the valid separator-sets with small area as early as possible. Such a procedure has the advantage that the generation of separator-sets can be stopped early to reduce computation time.

The procedure GENERATE_FEASIBLE_CUTS generates the separator-sets of interest and is described in Figure 3.14. It takes as input a flow-network, $flow(O_i)$, corresponding to the relevant network for output O_i where a saving of $\Delta(O_i)$ is desired. The flow-network is constructed as follows. For node n in the transitive-fanin of O_i at which $s(n) \leq s^*$ two vertices n^f and n^t are introduced in the flow network with a directed edge from n^f to n^t. The edge is given a weight equal to $A(n)$ if $s(n)+D(n) \geq \Delta(O_i)$ otherwise the edge weight is set to ∞. For all edges with $s(i,j) < s^*$ an edge with weight ∞ is introduced from vertex i^t to j^f. All the vertices that have no outgoing edge are connected to the "sink" vertex t and all vertices without any incoming edge are connected to the "source" vertex s by edges of infinite weight. This construction guarantees that all the valid separator-sets of the relevant network rooted at O_i are present as finite weight edge cuts in the flow network $flow(O_i)$. The finite weight cuts are generated by repeated application of a maxflow-mincut algorithm on the flow-network.

In order to generate a new separator-set on each application of the `maxflow()` procedurte, some nodes must be disallowed from being part of a cut. This is accomplished by temporarily setting the node weight to ∞. This is called "suppressing a node". Having generated a separator-set one can use a backtracking mechanism to choose the node to be suppressed next. This strategy is inefficient since it does not preclude the generation of the same separator-set a large number of times. One way to avoid the repeated generation of a separator-set is to find a minimum set of nodes that when suppressed will avoid the generation of any of the separator-sets generated thus far. The nodes to be suppressed are found by solving a covering problem. The separator-sets correspond to the rows of the covering matrix and the nodes are the columns. The matrix has an entry (u, v) if node v is a member of the separator-set u. By suppressing the nodes in a minimum column cover of the matrix, we ensure that the separator-sets generated thus far have an infinite weight. Thus every valid separator-set is generated only once. The other feature of GENERATE_FEASIBLE_CUTS is that the minimum-weight separator-set is the first one generated.

The procedure EXACT_SELECT_TREE, uses a number of sub-problems that are known to be computationally expensive. However, these techniques can be used in a reasonable amount of time on a large variety of examples. The reason for this is the small number of separator-sets that exist for each output. When the number of separator-sets is large we can put a limit on the number of separator-sets generated. This allows us to trade-off the computation time and the quality of the selection procedure.

It should be noted that the procedure EXACT_SELECT_TREE is based on a sufficient condition for improving circuit delay. The sufficient condition states that in order to improve the delay, it is sufficient to apply transformations that reduce the delay at a separator-set by the desired amount. This condition is clearly not necessary as there may be many transformations along a path that, when applied together, may yield the desired saving. One can, in principle, apply these transformations one at a time. If we revisit the circuit in Figure 3.12(a) but this time set the value of area penalty at nodes U and V to be 2 each, then the iterative procedure of HEURISTIC-1 (which would be akin to choosing both U and V together), does indeed provide the required saving at a smaller area penalty. However, the simultaneous choice of transformations requires a careful consideration of the interactions between the transformations that is computationally infeasible (the combinations of the transformations is exponential).

3.2.4 Generalization to DAG's

The previous section described heuristic and exact procedures for selecting a set of local transformations that achieve a reduction in delay guaranteed by the lower-bound technique with a small increase in area for the case when the relevant network for each output is a tree. This is a very restrictive assumption since most circuits that have been optimized for area have reconvergent paths. When the circuit has a directed acyclic graph (DAG) topology, the separator-sets in the circuit do not result in the saving computed according to the lower-bounding technique of Section 3.2.1. The circuit in Figure 3.15 illustrates this. The lower-bound computation results in a saving of 3 at the output. However, it can easily be seen that this can be achieved only by applying transformations at $\{T, X\}$ which does not form a separator-set of the graph.

The use of separator-set can be justified, even for DAG topologies, by using a weaker lower-bounding technique. If the difference in slack between the inputs of a gate is ignored while computing the saving, *i.e.*

$$\delta'(i) = \begin{cases} 0 & \text{if } i \text{ is a primary input} \\ \max[D(i), \min_{j \in FI(i)} \delta'(j)] & \text{otherwise} \end{cases} \tag{3.5}$$

then a weaker lower-bound on the improvement can be computed. Since the slacks difference between the inputs of a gate is ignored and the lower bound is propagated via a min procedure, it is guaranteed that if the improvement at output O_i is $\delta'(O_i)$ then for each path p from the inputs to output O_I there is a node n_p along the path with local improvement $D(n_p) \geq \delta'(O_i)$. Removing all the n_p nodes disconnects the output from the inputs. Thus $\delta'()$, the weaker lower bound on the delay improvement, is met by a selection-set that is a separator-set of the relevant network. For the circuit in Figure 3.15, the weak lower bound predicts $\delta'(Y) = 2$. By ignoring the difference in slacks, the fact that the delay along

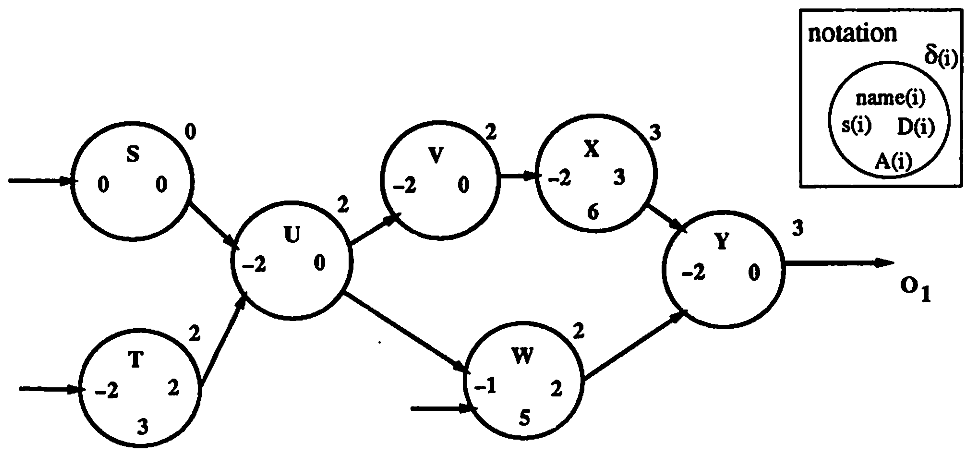


Figure 3.15: Example of a relevant network with a DAG structure

the specific path $\{S, U, W, Y\}$ is smaller than along the path $\{S, U, V, X, Y\}$ is ignored. A separator-set $\{X, W\}$ achieves a saving of 2. If there is no saving in delay at W, i.e. $D(W) = 0$, the predicted value of $\delta'(Y) = 0$ whereas the saving computed according to Equation 3.1 is $\delta(Y) = 3$. This shows that the weaker lower-bound, that ignores slacks, may be made arbitrarily poorer compared to the lower-bound that accounts for the slack difference.

At this stage, it is appropriate to summarize the facts relating the structure of the relevant network and the use of separator-sets for selecting transformations.

1. Using a fixed value of ϵ and weights according to Equation 3.4, a minimum-weight separator-set is guaranteed to achieve a saving of ϵ at a minimum increase in area for any circuit topology.

2. A saving Δ, computed according to Equation 3.3, can be achieved by a minimum weight separator-set for a tree topology (Theorem 3.2.1).

3. For a DAG topology, a saving δ', computed according to Equation 3.5 is achieved by some separator-set.

4. For a DAG topology, a separator-set is not adequate to guarantee a delay saving Δ, computed according to Equation 3.3.

```
   EXACT_SELECT_DAG($\eta$, $\{\Delta(O_i)\}$) {
       $\mathcal{F} = 1$
       for each output $O_i$ of $\eta$ with $s(O_i) < s^*${
1.             propagate-saving-backward($O_i$, $\Delta(O_i)$);
2.             $F(O_k)$ = build-selection-function($O_i$, $\Delta(O_i)$);
3.             $\mathcal{F} = \mathcal{F} \wedge F(O_k)$
       }
4.     $P$ = min-weight-satisfying-assignment($\mathcal{F}$)
5.     $\mathcal{S} = \{x \mid (x = 1) \in P\}$
   }
```

Figure 3.16: Selection procedure for DAG topology

When the relevant network is a DAG we need a selection procedure that follows the lower-bound computation of Equation 3.1. The procedure EXACT_SELECT_DAG of Figure 3.16 outlines the basic idea of imitating the lower-bounding technique to determine the set of nodes to transform.

In **Step 1.** of the algorithm, the saving required at each primary output is propagated backwards through the network according to the following equation.

$$\begin{aligned} \tau(j,i) &= \Delta(i) - s(i,j) - \min_{k \in FI(i)} s(i,k) \qquad \forall j \in FI(i) \\ \Delta(j,i) &= \begin{cases} 0 & \text{if } \tau(j,i) > \delta(j) \\ \tau(j,i) & \text{otherwise} \end{cases} \\ \Delta(j) &= \max_{i \in FO(j)} \Delta(j,i) \end{aligned}$$

The quantity $\tau(j,i)$ computes the value of saving that is required at the connection (j,i) if no transformation is applied at node i. When the achievable saving $\delta(j)$ is less than the expected saving $\tau(j,i)$, it must be the case that a transformation is selected at node i. Since the larger value of $\tau(j,i)$ cannot be met in the transitive fanin of j, the saving at node j is set to zero. A value $\Delta(j,*)$ is computed along each fanout of j. The maximum value of saving desired for any fanout is used as $\Delta(j)$, the amount of saving required up to the output of node j. For the circuit in Figure 3.15, the savings when propagated backwards result in $\Delta(X) = 3$, $\Delta(W) = 2$, $\Delta(V) = 0$, $\Delta(U) = 2$, $\Delta(T) = 2$ and $\Delta(S) = 0$.

After the saving that is required at each node has been computed, then during **Step 2**, all possible choices of nodes that lead to the desired saving are generated. This is accomplished by building a selection function at each node. *Satisfying assignments of the selection function, $F(i)$, represent the choices for transformations that result in a saving of $\Delta(i)$ at node i.* $F(i)$ is computed in topological order from the inputs to the outputs.

$$F(i) = \begin{cases} 1 & \text{if } \delta(i) = 0 \\ 0 & \text{if } \Delta(i) \leq 0 \text{ and } \delta(i) > 0 \\ i + \prod_{j \in FI(i)} F(j) & \text{if } D(i) \geq \Delta(i) \\ \prod_{j \in FI(i)} F(j) & \text{otherwise} \end{cases}$$

At a node i, if the local saving $D(i)$ is greater than the desired saving $\Delta(i)$ then the selection function includes i as an implicant. The desired saving may also be achieved at each of the inputs and that provides the other contribution to the selection function. It should be noted that since $\Delta(O_k) \leq \delta(O_k)$, the selection function at any relevant output cannot be identically 0.

To illustrate the construction of the selection function, the selection function for each node in Figure 3.15 is listed.

$$\begin{array}{lcll} F(S) & = & 1 & \\ F(T) & = & T + 0 & = \mathrm{T} \\ F(U) & = & F(S) \cdot F(T) & = \mathrm{T} \\ F(V) & = & 0 & \\ F(W) & = & W + F(U) & = \mathrm{W} + \mathrm{T} \\ F(X) & = & X + F(V) & = \mathrm{X} \\ F(Y) & = & F(W) \cdot F(X) & = \mathrm{X} \cdot (\mathrm{W} + \mathrm{T}) \end{array}$$

As was stated earlier, each relevant output needs to be improved by a specific amount to ensure that the circuit delay is reduced. Thus the overall selection function is the conjunction of the selection functions for each relevant output. After iterating over steps 1, 2 and 3, the overall selection function $\mathcal{F}$ is generated.

$$\mathcal{F} = \prod_{s(O_k) < s^*} F(O_k)$$

The choice of transformations that achieves the lower bound at a minimum area increase is the minimum weight satisfying assignment of $\mathcal{F}$. As described earlier this can be found efficiently using a shortest-path computation on the BDD representing $\mathcal{F}$. From the minimum-weight assignment the selection-set S is determined. For the circuit in Figure 3.15, the minimum-weight satisfying assignment of the selection function is $\{X, T\}$ that has a cost of 9. The other satisfying assignment, $\{X, W\}$ has a cost of 11.

It is interesting to note that when the circuit has a tree structure, the procedure EXACT_SELECT_DAG produces the same solution as the procedure EXACT_SELECT_TREE that is based on generating separator-sets. The DAG-based procedure builds the selection function as a multi-level function. In contrast, a separator-set represents a cube in the representation of the selection function for an output. The enumeration of separator-sets may be viewed as generating a two-level cover for the selection function. In that respect, the general method is more efficient since the two-level representation may have a size that is exponential in the size of the of multi-level representaion.

3.3 How effective is iterative improvement?

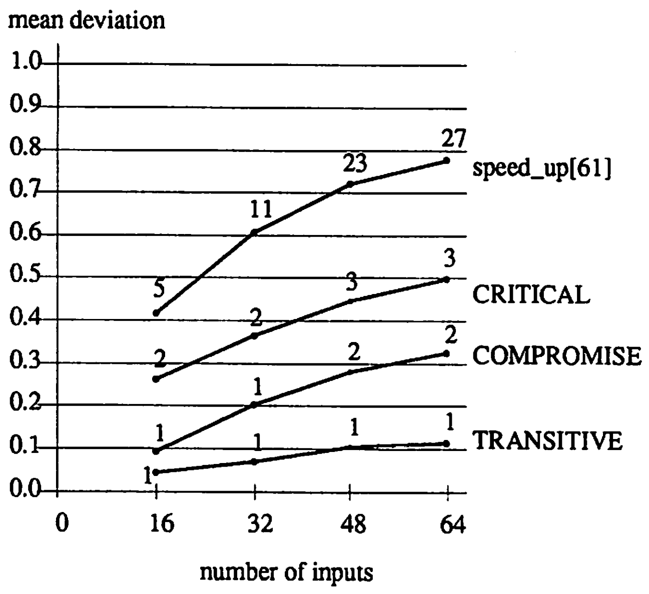


Figure 3.17: Comparison of strategies used to determine transformation scope

The use of iterative application of local transformations to reduce the circuit delay is a heuristic procedure. To determine the power of the heuristic a comparison with an exact algorithm is required. In the case of an AND function the optimum algorithm to decompose it into 2-input gates is known [75]. We will use this to explore the power of local transformations in optimizing networks computing the conjunction of n inputs with arbitrary arrival times. The initial implementation may be a serial chain of 2-input AND gates, a parallel tree or a random tree structure. For each of these circuit structures, the inputs are assigned random arrival times. The range of arrival times is 0 to $2n$. The optimum arrival time at the output is compared with the arrival time of the circuit optimized by iterated application of local resynthesis. The only transformation used is the timing-driven decomposition which is optimum for an AND function. By using the best possible local transformation the impact of the strategy used to determine transformation scope can be studied.

Trees with different circuit structures and random arrival times aere optimized using local transformation and the output arrival time was compared to the optimum for the given random arrival times. Different strategies for selecting the scope of the transformion — *critical*, *transitive* and *compromise* — are evaluated and the depth of the scope is limited to 3 nodes. Figure 3.17 shows the average deviation from the optimum (plotted value) as well as the maximum deviation from the optimum (printed value) for circuits with different number of inputs. It is clear that the selection strategy affects the quality of the result. As is to be expected, the *transitive* strategy is the most effective while the most restrictive strategy, *critical*, which looks only along the critical-path, is the weakest. With any of the selection strategies, iterative improvement based on the lower bounding technique performs better than the heuristic weighting technique of [61]. For all the experimental circuits that were generated, iterative improvement using the *transitive* strategy provides a decomposition that is no slower than 1 compared to the optimum decomposition. For other methods of determining the transformation scope (*critical* and *compromise*), the maximum deviation increases as the circuit size increases.

The depth of the selection region is 3 levels. As the size of the circuit increases the transformation works on a smaller percentage of the circuit. This reduces its ability to produce global changes through iterated application. This observation is supported by the experimental data of Figure 3.17. When the circuit is a parallel tree with sixteen inputs and the depth of the scope is 3, the local transformation at the root encompasses the entire

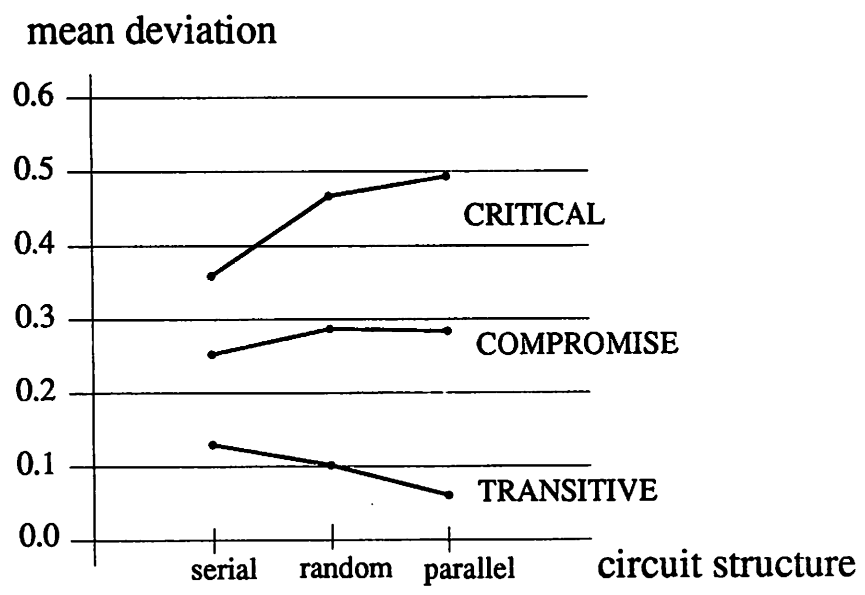


Figure 3.18: Interaction of circuit structure and scope strategy

tree and so optimum circuits are obtained for any set of random arrival time values at the inputs.

Another interesting observation is that the strategy used to determine transformation scope is influenced by the structure of the initial circuit. The choice of looking along the critical paths works well when the circuit has a serial structure. Due to the chain like circuit structure, there is not much interaction between the different paths and so looking along the critical path is a viable method. In the case of a balanced-tree structure, several paths may have to be modified simultaneously. This is not possible if the scope is chosen to be lie along the most critical path. Thus for parallel structures, strategies that have a wider scope, e.g. *transitive*, perform better. Figure 3.18 shows the mean deviation from the optimum plotted as a function of the circuit structure for the different scope-determining methods. The critical-path based strategy performs better on serial structures than on parallel structures while the *transitive* strategy performs better on parallel circuit structures than on serial structures. The *compromise* strategy is quite immune to the structure of the initial structure.

The above experiment avoids the possible dependence between the selection of the scope and the quality of the local transformation. This independence is achieved since the decomposition transformation is optimum for AND gates. When the logic function generated by collapsing the nodes in the scope is a complex sum-of-products expression, the transformations of Section 3.1 do not produce optimum decompositions. Furthermore, as the size of the function increases the quality of the decomposition may decrease. Thus increasing the depth of the scope is not guaranteed to produce better optimization in the case of general circuits.

This simple experiment demonstrates that application of local transformations (even locally optimum transformations) under the proposed optimization procedure does not always generate the optimum solution. It is however heartening to note that the mean deviation from the optimum is small and that the proposed procedure produces much better results than the heuristic strategy of [61]. It would also be interesting to evaluate a rule-based system with the same input. This would provide a fair comparison between the power of an algorithmic selection strategy such as the above versus the application of a sequence of rules. Unfortunately this comparison could not be made due to lack of access to a rule based optimization system.

3.4 Results of technology-independent optimization

The techniques for circuit restructuring based on the application of local transformations has been implemented as part of SIS (Sequential Interactive System) that is an environment for performing various logic synthesis operations [58]. The system provides support for representing and manipulating Boolean functions and networks. This section describes the results obtained by the application of the techniques described in this chapter and compares them with other existing techniques.

3.4.1 Description of example circuits

For the purpose of validating the various ideas presented in this chapter a benchmark set is used. Our set of benchmark circuits comes from the the 1991 MCNC logic synthesis benchmark set [79] and some industrial designs (from AT&T and Intel). The suggested subset of the two-level PLA examples and multi-level examples is chosen from the MCNC benchmark examples. The industrial examples have no details of their functionality but provide a valuable source of large design examples. The initial descriptions of the examples were optimized for area using the script `script.rugged` provided with SIS. For the two-level examples the SIS commands (`resub -a; simplify -m nocomp -d`) were run before the script to avoid runtime problems. Table 3.1 shows some information about the optimized examples including the number of inputs and outputs, the circuit size as measured by the literals in factored form and also the area after a minimum area mapping using the MCNC cell-library `lib2`. The source for the example and its function is also described. In the examples where the functionality is not known the word "Logic" is used.

3.4.2 Choice of initial decomposition

The application of local transformations to restructure a circuit is carried out on a network composed of 2-input gates. Since the transformations try to reduce the depth of the network, it is important to start the optimization on a network structure that has a small depth. The operations used to decompose the network allow an area/delay tradeoff. Some decomposition procedures used to generate the 2-input gate network are described here in terms of the commands used in SIS .

good_decomp; tech_decomp -a2 Routines to generate 2-input representations are based on the sum-of-products representation of the function. A node may have a small factored form but a large SOP representation. **good_decomp** [75] is a procedure to represent the function as an interconnection of smaller functions. Each of the smaller functions has a small SOP representation as well. The decomposition is chosen to find a set of functions whose interconnection requires a small area. After the decomposition, each node is represented in terms of 2-input NAND gates and inverters. This decomposition into 2-input gates is done without regard to the arrival times of the inputs, i.e. a balanced decomposition of each cube followed by a balanced decomposition of the sum term is carried out. This process is called **technology-decomposition** [50] since it is a precursor to the technology-mapping algorithms. This strategy is geared towards generating a 2-input decomposition with small area.

speed_up -i At the other end of the spectrum of techniques, a decomposition procedure that generates the smallest depth 2-input decomposition is possible. The nodes in the network are sorted in topological order (a node appears after all its fanins). Now timing-driven decomposition (Section 3.1.2) can be applied to each node in topological

Example	**gd; td -a2**		**gd; el -1; sp -i**		**sp -i**	
	depth	**lits**	**depth**	**lits**	**depth**	**lits**
C1355	20	812	19	820	19	820
C1908	31	784	30	800	30	810
C2670	32	1188	24	1253	24	1259
C3540	42	2053	41	2140	41	2154
C432	31	312	27	314	27	312
C6288	92	4443	90	4829	90	4829
C7552	41	3569	34	3571	34	3827
ampbpreg	19	1368	19	1497	19	1397
ampbsm	20	1105	15	1116	15	1116
amppint2	32	893	22	913	20	927
ampxhdl	18	527	16	533	16	529
b12	7	131	7	136	8	136
b9	10	188	10	200	10	228
cordic	14	126	11	130	11	128
cps	22	1902	17	1894	17	1894
dalu	23	1570	19	1618	19	1778
des	26	6051	20	6057	19	6079
dflgrcb1	13	475	12	505	12	498
duke2	23	676	15	662	15	662
ex1010	19	3942	19	3998	18	3930
ex4	14	839	11	944	10	1000
fconrcb1	14	346	12	361	12	361
k2	21	1687	20	1703	20	1703
kcctlcb3	10	341	9	358	9	358
misex2	10	160	7	163	7	163
misex3c	47	796	26	834	26	832
pdc	16	585	14	601	14	603
rd84	15	209	11	213	11	209
rot	25	1050	21	1093	21	1089
sbiucb1	20	383	16	395	16	395
spla	21	980	16	978	16	981
t481	23	1183	18	1159	18	1183
tfaultcb1	10	261	9	280	9	280

gd; td	Decomposition targeting small area
gd; el -1; sp -i	Compromise between area and delay based decompositions
sp -i	Initial decomposition for small delay
depth	Maximum depth of the decomposed circuit
lits	Number of literals in the decomposed network

Table 3.2: Comparison of techniques for decomposition into 2-input NAND gates

3.4.3 Experiments with the proposed procedure

The optimization procedure developed in this chapter is a recipe that can be tailored according to the users requirements. The parameters that guide this choice are the cpu-time and the quality of the solution desired. The "quality" of the solution may be measured solely on the improvement in circuit performance or it may also account for the area increase. The following sections investigate how some of the choices affect the quality of optimization.

Selection of the scope

This aspect was dealt with in Section 3.3 with the observation that the *transitive* selection strategy was the most powerful and looking along the critical path provided the least improvement. For the AND -trees considered therein, the area of the optimized circuits was the same as that of the initial circuit. In addition, the local transformation was optimum. Here we will evaluate the effect of the selection strategy on real circuits where the local transformations are not optimum and where the DAG-structure results in logic duplication. The larger the scope, the larger is the duplicated area. The local transformation used here is timing-driven-decomposition of Section 3.1.2. Both the function and its complement are evaluated to determine the best decomposition based on extracting kernels. The depth of the scope is equal to 3. Table 3.3 describes the results of varying the selection strategy.

Due to the wide variation in the size and depth of the circuits the arithmetic mean is dominated by the bigger circuits. To avoid this, the geometric mean is used as the comparison statistic. The geometric mean of data values $a_1, \ldots, a_n$ is simply

$$\textbf{G.MEAN} = \left(\prod_{i=1}^{n} (a_i) \right)^{1/n}$$

The difference between the selection strategies is not very spectacular. On the average, looking along the critical path, results in circuits where the decrease in levels may be smaller than with either the *transitive* or the *compromise* strategy. For 24 of the 33 circuits, all three strategies led to the same delay decrease. In addition, for these circuits, the area of the optimized circuits were very similar. This leads us to believe that the strategy used to determine transformation scope is not a significant parameter in the optimization.

Example	critical		transitive		compromise	
	lits	depth	lits	depth	lits	depth
C1355	1284	15	1284	15	1284	15
C1908	1005	22	991	22	1007	22
C2670	1360	16	1358	16	1360	16
C3540	2188	31	2246	31	2198	31
C432	586	20	528	21	592	20
C6288	5391	68	5423	69	5444	67
C7552	4240	19	4585	19	3833	21
ampbpreg	1593	14	1575	14	1571	14
ampbsm	1152	10	1148	10	1164	10
amppint2	960	14	990	14	986	14
ampxhdl	589	11	801	10	824	10
b12	148	6	148	6	148	6
b9	204	7	200	7	200	7
cordic	172	9	172	9	172	9
cps	1918	12	1914	12	1916	12
dalu	1727	13	1698	14	1687	13
des	6115	17	6161	17	TO	TO
dflgrcb1	502	9	508	9	502	9
duke2	698	12	702	12	714	12
ex1010	3936	14	3936	14	3936	14
ex4	944	10	944	10	944	10
fconrcb1	382	9	382	9	382	9
k2	1731	15	1735	15	1733	15
kcctlcb3	401	7	385	7	403	7
misex2	182	6	182	6	182	6
misex3c	932	21	886	22	879	22
pdc	612	11	617	11	614	11
rd84	209	11	209	11	209	11
rot	1182	15	1174	14	1179	14
sbiucb1	397	13	397	13	397	13
spla	1003	12	999	11	1002	12
t481	1189	16	1187	16	1187	16
tfaultcb1	280	8	280	8	280	8
G. MEAN	808	12.91	813	12.88	813	12.89

critical	Only critical path is part of the scope
transitive	Entire depth bounded transitive fanin is transformed
compromise	Based on delay data, some nodes may be added to **critical**
depth	Maximum depth of the optimized circuit
lits	Number of literals in the optimized network

Table 3.3: Comparison of strategies for selecting regions to transform

Choice of the local transformation

This experiment evaluates a number of local transformations, all in the same environment, to judge which transformation technique is superior. The local transformations that are evaluated are the two decomposition methods (based on kernels and 2-cube divisors), timing-driven cofactoring and the generalized bypass transformation. For each local transformation, the *critical* strategy is used to determine the scope. This is due to the fact that the bypass and cofactoring techniques work best when there is a small number of critical inputs. The depth of the circuit to transform is restricted to 3 (to allow the kernel based technique to execute in reasonable time). It should be pointed out that using a fixed depth circuit for the bypass does not give the best results. Table 3.4 summarizes the results of this experiment. Entries marked "TO" do not complete within a reasonable time.

Transformations based on the logic function (timing-driven decomposition) perform better than techniques that are based on circuit structure (bypass and cofactor transformations). Also, the area increase is more for transformations that are able to reduce delay by a larger amount. Regarding, the timing-driven decomposition techniques, kernelling explores a larger set of divisors, compared to 2-cube divisors, and is able to reduce delay by a larger amount.

One unforeseen result is that the timing-driven-decomposition based on 2-cubes (TDD-2cube) runs out of memory on more circuits than the timing-driven-decomposition based on kernels (TDD-kernel). This was unexpected since there are far fewer 2-cube divisors than there are kernels. Closer inspection reveals that the data-structures used in the implementation are the cause of this. Kernelling is based on memory-efficient data structures whereas the 2-cube divisors have overheads associated with each divisor, leading to memory limitations.

Depth of the scope

The local transformations affect a small part of the circuit. It seems natural that the larger the region being transformed, the larger would the improvement in a single iteration. However, it would be at the cost of an increase in the amount of logic being duplicated. Due to the problems of evaluating the kernel-based extraction on large functions, the transformation chosen for this experiment is the decomposition based on 2-cube kernels. The strategy used to select the region to transform is the compromise strategy based on the

Example	Initial		TDD-kernel		TDD-2cube		TDC		GBX	
	lits	depth	lits	depth	lits	depth	lits	depth	lits	depth
C1355	820	19	1284	15	964	15	820	19	1188	18
C1908	800	30	1005	22	973	22	868	28	806	28
C2670	1253	24	1360	16	1342	17	1267	22	1258	18
C3540	2140	41	2188	31	2204	29	2149	33	2147	37
C432	314	27	586	20	548	21	314	27	320	26
C6288	4829	90	5391	68	5465	69	4945	88	4973	74
C7552	3571	34	4240	19	3929	20	3685	28	3771	26
ampbpreg	1497	19	1593	14	1595	14	1525	15	1503	15
ampbsm	1116	15	1152	10	1152	10	1138	12	1141	11
amppint2	913	22	960	14	974	15	943	18	941	15
ampxhdl	533	16	589	11	589	11	545	14	557	12
b12	136	7	148	6	148	6	148	6	148	6
b9	200	10	204	7	207	7	196	8	199	7
cordic	130	11	172	9	172	9	137	10	140	9
cps	1894	17	1918	12	1918	12	1902	13	1900	11
dalu	1618	19	1727	13	1786	13	1664	15	1609	14
des	6057	20	6115	17	TO	TO	6203	17	6069	17
dflgrcb1	505	12	502	9	502	9	495	12	498	11
duke2	662	15	698	12	698	12	676	13	672	13
ex1010	3998	19	TO	TO	TO	TO	3936	14	3936	14
ex4	944	11	944	10	958	10	918	11	915	10
fconrcb1	361	12	382	9	376	9	360	12	370	11
k2	1703	20	1731	15	1731	15	1705	16	1710	15
kcctlcb3	358	9	401	7	381	8	349	9	353	8
misex2	163	7	182	6	182	6	165	7	165	7
misex3c	834	26	932	21	869	22	820	25	831	23
pdc	601	14	612	11	621	12	611	12	612	11
rd84	213	11	209	11	209	11	209	11	209	11
rot	1093	21	1182	15	1134	16	1136	18	1113	16
sbiucb1	395	16	397	13	433	13	390	16	389	15
spla	978	16	1003	12	1005	12	976	15	986	13
t481	1159	18	1189	16	1189	16	1189	16	1186	16
tfaultcb1	280	9	280	8	280	8	278	9	285	8
G. MEAN	697	16.9	767	12.9	756	13.1	706	15.2	715	14.1

Initial	Area-optimized circuit decomposed into 2-input gates
TDD-kernel	Timing-driven decompositon based on kernels
TDD-2cube	Timing-driven decompositon based on 2-cube divisors
TDC	Timing-driven cofactoring technique
GBX	Generalized bypass transformation
depth	Maximum depth of the optimized circuit
lits	Number of literals in the optimized network

Table 3.4: Comparison of local transformation techniques

distribution of delay data. Different depths of the transformed region are evaluated and the results are shown in Table 3.5. For a depth of 3, the absolute area and depth are shown. To allow easier comparison, for depths of 2, 4 and 5 only the difference (from a depth of 3) is shown.

As expected, increasing the scope of the transformation results in a greater reduction in the number of levels accompanied with an increase in circuit area. The disconcerting observation is that this trend is only true on the average — for some examples increasing the scope of the transformation results in a smaller improvement. This can be attributed to the heuristic nature of the local transformations which often degrade in quality as the functions become large. Increasing the scope of the transformations also leads to larger memory use and in a few cases this is prohibitive. A small scope of local transformation brings to light the shortcoming of using local transformations to effect global change. The examples *pdc* and *kcctlcb3* are interesting since the delay reduction with a small (d=2) and large (d=4) depth exceeds the reduction for a depth of 3. A possible explanation may be the heuristic nature of the decomposition. From this experiment we see that a depth of 3 (in terms of 2-input gates) provides a good compromise to determine the scope of timing-driven decomposition.

Area considerations

In the previous experiments, the sole consideration was to get a circuit that was as fast as possible. The circuit area was a secondary criterion, used only to break ties between selection-sets that provide the same amount of improvement. This section describes ways in which the area of the optimized circuit can be kept small while still getting significant improvement in depth.

One way of reducing the area overhead is by not using aggressive optimization while generating a local transformation. Rather than generate the best solution, it is often possible to generate decomposition that provide a smaller improvement in delay and are area efficient. As an example, consider timing-driven-decomposition. While evaluating different divisors, the area saving that results from extracting the divisor is taken into account. The weight of a divisor, W, is a combination of the area saving, A, and a timing weight, D, based on the distribution of the arrival times of its inputs.

$$W = \alpha A + (1 - \alpha) \times D$$

Example	d=2		d=3		d=4		d=5	
	Δ l	Δ d	lits	depth	Δ l	Δ d	Δ l	Δ d
C1355	256	1	964	15	TO	TO	TO	TO
C1908	22	1	973	22	151	0	TO	TO
C2670	-88	2	1342	17	-65	0	TO	TO
C3540	-24	3	2204	29	148	1	131	1
C432	-206	2	548	21	TO	TO	TO	TO
C6288	-617	1	5465	69	164	1	TO	TO
C7552	-122	2	3929	20	1	1	191	1
ampbpreg	-98	1	1595	14	TO	TO	54	0
ampbsm	-10	1	1152	10	46	0	110	0
amppint2	-25	0	974	15	89	-1	146	-2
ampxhdl	-36	1	589	11	149	-1	297	-1
b12	0	0	148	6	-5	-1	-5	-1
b9	-13	1	207	7	6	0	12	0
cordic	-17	0	172	9	-42	0	-42	0
cps	4	0	1918	12	14	0	14	0
dalu	-152	1	1786	13	-23	0	-58	0
des	6205*	15*	TO	TO	TO	TO	TO	TO
dflgrcb1	8	0	502	9	-1	0	34	-1
duke2	-12	0	698	12	30	0	22	0
ex1010	3936*	14*	TO	TO	TO	TO	TO	TO
ex4	11	0	958	10	-14	0	6	0
fconrcb1	11	0	376	9	13	0	29	0
k2	-30	1	1731	15	0	0	58	0
kcctlcb3	8	-1	381	8	62	-1	-26	0
misex2	-1	0	182	6	0	0	7	0
misex3c	-19	0	869	22	62	0	62	0
pdc	8	-2	621	12	23	-2	45	-2
rd84	0	0	209	11	0	0	0	0
rot	-12	0	1134	16	12	-1	174	-2
sbiucb1	-44	2	433	13	-33	1	26	0
spla	-16	2	1005	12	33	-1	89	-1
t481	0	0	1189	16	0	0	0	0
tfaultcb1	0	0	280	8	6	0	11	0
Overall	-1214	19			826	-4	1387	-8

d=2,3,4,5	Maximum depth of the region being transformed
depth, (Δ d)	Maximum depth (difference from d = 3) of the optimized circuit
lits, (Δ l)	Number of literals (difference from d = 3) in the optimized network

Items marked with (*) represent absolute values in a difference column

Overall	Sum of all differences.

Table 3.5: Effect of transformation depth on optimization

where α is a fraction that controls the relative importance of area. Setting $\alpha = 0$, results in an aggressive optimization where decreasing the delay is the only criterion. At the other extreme, setting $\alpha = 1$ ignores the arrival time data and the decomposition has a small area.

For an aggressive optimization strategy, the best transformation is chosen at a node regardless of the area penalty. It is only later, during the choice of the selection-set, that the area increase is minimized. However, by using a *benefit/cost* metric (see Section 3.2.1) to select the local transformation at a node, the area spent to achieve a unit improvement in delay can be reduced.

To isolate the effect of each of these considerations a multiple part experiment is performed. A set of local transformations based on timing-driven-decomposition are used — TDD-kernel, TDD-2cube, TDD-comp-kernel, TDD-comp-2cube. The latter two perform timing-driven decomposition on the complement of the function. A "compromise" selection strategy with depth 3 is used to determine the scope of each transformation. First the aggressive local optimization with *benefit* metric is used (AGG-B). Next the *benefit/cost* metric is introduced (AGG-BC). Then, the non-aggressive local transformation is evaluated using the *benefit* metric (NOAGG-B). Finally, as the most area-efficient technique, the non-aggressive optimization is used along with the *benefit/cost* metric (NOAGG-BC). Table 3.6 shows the results of this experiment.

The experimental observations agree with the predictions — the aggressive optimization and the use of the *benefit* based evaluation of transformations produce the best optimization at a correspondingly larger circuit area. Using the *benefit/cost* evaluation improves the area by a very little amount. When non-aggressive optimization is used, the increase in circuit area is smaller but the amount of delay reduction decreases. Combining the non-aggressive optimization with the *benefit/cost* evaluation does not result in any further change. It should also be pointed out that the fewest iterations are required when the *benefit* evaluation is used along with aggressive optimization.

In this experiment, there is not much difference between using the *benefit* and the *benefit/cost* evaluation schemes. The reason for this may be the coarse granularity of delay, measured as the number of levels. Due to the discrete nature of delays, often the different transformations lead to the same improvement in delay. In that case the *benefit* evaluation method is equivalent to the *benefit/cost* method.

Example	AGG-B		AGG-BC		NOAGG-B		NOAGG-BC	
	lits	depth	lits	depth	lits	depth	lits	depth
C1355	1232	15	1232	15	836	15	836	15
C1908	1002	21	931	22	842	22	842	22
C2670	1398	16	1398	16	1322	16	1320	16
C3540	2174	30	2174	30	2216	29	2216	29
C432	480	18	480	18	466	17	466	17
C6288	5343	67	5369	67	5010	67	5010	67
C7552	4006	19	3820	20	3861	19	3861	19
ampbpreg	1588	14	1588	14	1606	14	1606	14
ampbsm	1163	10	1165	10	1154	10	1154	10
amppint2	1006	13	1006	13	999	12	999	12
ampxhdl	657	10	657	10	573	11	573	11
b12	148	6	148	6	148	6	148	6
b9	209	6	209	6	194	8	194	8
cordic	160	8	160	8	154	10	154	10
cps	1987	11	1985	11	1920	12	1920	12
dalu	1652	13	1649	13	1668	13	1668	13
des	6113	16	TO	TO	6231	16	6231	16
dflgrcb1	502	9	499	8	495	8	495	8
duke2	684	12	690	12	688	12	688	12
ex1010	TO	TO	TO	TO	3936	14	3936	14
ex4	996	9	996	9	918	11	918	11
fconrcb1	375	9	375	9	375	9	375	9
k2	1735	15	1739	15	1727	15	1727	15
kcctlcb3	405	7	405	7	395	7	395	7
misex2	182	6	182	6	182	6	182	6
misex3c	878	21	878	21	910	20	910	20
pdc	626	10	635	10	634	10	634	10
rd84	212	10	212	10	208	10	208	10
rot	1184	13	1185	13	1142	14	1142	14
sbiucb1	421	12	412	13	401	13	401	13
spla	1012	11	1011	11	1023	11	1023	11
t481	1187	16	1187	16	1187	16	1187	16
tfaultcb1	284	8	284	8	280	8	280	8
G. MEAN	765	12.33	762	12.35	736	12.63	736	12.63

AGG-B	Aggressive optimization using the Benefit metric
AGG-BC	Aggressive optimization using the Benefit/Cost metric
NOAGG-B	Non-aggressive optimization using the Benefit metric
NOAGG-BC	Non-aggressive optimization using the Benefit/Cost metric
depth	Maximum depth of the optimized circuit
lits	Number of literals in the optimized network

Table 3.6: Effect of area-saving parameters

3.4.4 Comparison with other techniques

The restructuring techniques used in the experiments do not use the don't care information in the circuit. Previous techniques like the one in [68] rely on the use of don't cares and on redundancy removal to reduce the area after partial collapsing of the clusters of nodes determined by a partitioning for delay. Table 3.7 compares the results of the two techniques in terms of the levels of logic (measured using 2-input gates) just before mapping the two circuits. The script used for implementing the clustering techniques was `script.delay` that is part of the SIS distribution and the decomposition into 2-input gates was done using the `tech-decomp` command of SIS. The critical-path-restructuring was done using only non-aggressive TDD-2cube (favoring small area implementations) with a scope of depth 3.

```
do
    reduce_depth -S 8 -r
    red_removal
    eliminate -l 100 -1
    simplify -l
    full_simplify -l
    sweep
    decomp -q
    fx -l
    speed_up -m unit -i
while (delay decreases)
```

Figure 3.19: Iterative clustering script

The result of this experiment shows that restructuring is more effective in reducing the depth of circuits. This is not surprising if the nature of the two techniques is studied. Partial collapsing is a single pass operation that clusters nodes and tries to simplify them without increasing the circuit depth. Restructuring, on the other hand, is a multi-pass operation. For a fair comparison, multiple passes are made to allow the clustering to operate on networks of smaller depth. The decomposition of complex nodes into 2-input gates is done using timing-driven decomposition rather than by simple AND-OR decomposition

Example	cpr		clustering		multi-clust	
	lits	**depth**	**lits**	**depth**	**lits**	**depth**
C1355	836	15	1064	17	1048	17
C1908	842	22	1044	30	1047	27
C2670	1320	16	TO	TO		
C3540	2216	29	2271	38		
C432	466	17	425	32		
C6288	5010	67	5329	90	4659	88
C7552	3861	19	4252	33	4313	30
ampbpreg	1606	14	1531	11		
ampbsm	1154	10	1214	16	1257	12
amppint2	999	12	927	17	1031	11
ampxhdl	573	11	564	11		
b12	148	6	134	7		
b9	194	8	222	9	192	9
cordic	154	10	132	12		
cps	1920	12	2129	17	2034	14
dalu	1668	13	1694	16	1854	14
des	6231	16	6674	24	6685	19
dflgrcb1	495	8	511	10		
duke2	688	12	704	16	739	10
ex1010	3936	14	TO	TO		
ex4	918	11	939	17	911	13
fconrcb1	375	9	395	13	376	12
k2	1727	15	2273	17	2073	14
kcctlcb3	395	7	487	9	462	8
misex2	182	6	187	10	173	6
misex3c	910	20	782	32	842	21
pdc	634	10	574	12	529	10
rd84	208	10	261	14	247	14
rot	1142	14	1224	22	1228	16
sbiucb1	401	13	436	19	420	13
spla	1023	11	1055	15	1000	13
t481	1187	16	TO	TO		
tfaultcb1	280	8	318	9	273	8

cpr	critical-path restructuring using 2-cube divisors
clustering	Clustering performed using `script.delay`
multi-clust	Multiple passes using modified `script.delay` of Figure 3.19
depth	Maximum depth of the optimized circuit
lits	Number of literals in the 2-input NAND representation

Table 3.7: Comparison of clustering and critical-path restructuring

(technology-decomposition). This change, by itself, results in fewer levels for some of the examples in which repetition did not help. The blank entries in Table 3.7 indicate that there was no gain from multiple passes. Figure 3.19 shows the script used to apply the clustering process repeatedly. In only six examples (indicated in italics) the multiple-pass clustering results in circuits with smaller area and with the same or better delay than critical-path restructuring. This points out that the local transformations lack the optimizations that result from exploiting the don't care conditions arising from the network structure. The clustering technique also uses redundancy removal while the restructuring techniques do not. The success of local restructuring in reducing the number of levels in the circuit, even when techniques that use don't care information are not used, illustrates the power of the selection heuristic.

3.4.5 Optimization of a 32-bit adder

The above experiments show the effectiveness of local restructuring techniques in reducing the depth of circuits. Since, a number of local transformations are derived from optimizations performed on arithmetic adder circuits, an interesting example circuit to study is a 32-bit adder. Manual designs exist for small area (ripple-carry adder) and small delay (carry-look-ahead adder). A validation of the local transformation approach would be to generate the look-ahead adder from the slow ripple carry adder. Other types of adders like carry-select and carry-select also use specific local transformations (cofactoring and generalized bypass respectively) and the comparison between manual and automatically generated structures is meaningful. The bypass adder depends on making long paths false [45] and requires viability analysis to predict the true delay. Since the local transformation uses static-delay analysis it is difficult to compare the results of the bypass transformation with that of critical-path restructuring. The redundancies in the bypass adder are removed [31] so that the static delay will also reflect the true delay.

The initial description of the 32-bit adder is a cascade of the following two-bit adder. The carry generated in stage $i-1$ is used in stage i.

$$\begin{aligned} c_i &= c_{i-1} \cdot (a_i + b_i) + a_i \cdot b_i \\ o_i &= a_i \oplus b_i \oplus c_{i-1} \end{aligned}$$

Figure 3.20 shows the result of applying the different transformations and some of the other adder structures (the carry-look-ahead and the carry-bypass adders). In these experiments we also plot the intermediate circuits, produced after every iteration of applying the selected local transformations. A part of the graph has been magnified to emphasize the region of greatest interest.

This experiment shows the effects of choosing different initial starting points, strategies to determine the scope of transformations, use of the area saving considerations on the series of circuit generated. At any given value of delay, the optimal solution is the one with the least area. A good optimization strategy generates optimal structures during its entire range. Unfortunately, there is no single method that provides the minimum area solution over the entire range of achievable delay values.

Aggressive optimization using all the transformations and using the *critical* strategy with a depth of 3 to determine the scope of transformations provides the maximum decrease in delay. This optimization scenario is called **all:agg:trans**. Typically, for a given value of delay, the circuits with small area were obtained by using two-cube divisor based decompositions that were not aggressive. Also, focusing along the critical path led to retaining good boolean factors that were lost if broader scope was used for evaluating the transformations. This area-efficient strategy is called **2c:nagg:crit**. The strategy of looking along the critical path is effective since the circuit has a single longest path. This underscores the importance of providing a framework where different choices for the scope, the severity and type of optimization can be easily experimented with to provide the designer a tool to optimize the circuit.

In the case of the adder circuit it is possible to generate, automatically, circuit structures that match manually-generated structures for the fast **carry-look-ahead (CLA)** adder. The two configurations of look-ahead adders correspond to using 2-bit and 4-bit lookahead blocks to generate the 32 bit adder. We are not as successful in obtaining circuit structures that match manually generated **carry-bypass (CBA)** adder configurations. The reason for this is under investigation. Possible explanations may be the choice of scope (there is a good strategy to determine the scope [45] whereas we use a fixed depth scope) and that the transformation is not as area efficient as it can be (adding redundancy removal might alleviate this).

Another benefit of the proposed procedure over the heuristic restructuring of [61] is that it is more robust to changes in the initial decomposition of the circuit into 2-input gates. Figure 3.20 shows two different starting points that correspond to decompositions of the ripple carry adder using the following SIS commands —

(1) `good_decomp; tech_decomp -a 2`

(2) `good_decomp; eliminate -1; speed_up -i`

Application of iterative improvement using non-aggressive TDD-2cube applied along the critical path provides similar results when started from the different starting points.

An interesting observation is that the locus of optimal solutions is not a monotone function. This seems to suggest that the circuit structures obtained are not the best and that there is scope for improvement.

3.5 Conclusions

This chapter has addressed the problem of reducing the depth of circuits used to represent logic functions. How different operations affect the circuit depth has been described. In large circuits, that cannot be collapsed, these operations are viewed as local transformations. Since the delay of the circuit is determined by the longest path the local transformations have to be applied at appropriate parts of the circuit to ensure a reduction in delay. A paradigm to apply local transformations to reduce the delay with a small area penalty has been presented.

The framework for applying local transformations can benefit tremendously from the proper directions of an experienced designer. In particular, insight that the designer possesses can be used to make appropriate decisions regarding the following aspects.

1. Determine a good local transformation at a node. Currently, only the size and topology of the scope can be varied to find good local transformations.

2. Provide a set of powerful transformations based on knowledge of the circuit functionality. For specific arithmetic circuit functions, optimizations related to the mathematical properties may be exploited.

3. Determine the relative importance of keeping the area increase small, based on the constraints that the circuit must satisfy.

Experimental results are provided to discuss the tradeoffs that arise when different choices are made for the various components of the optimization procedure. This allows a designer to explore various alternate circuit structures for their designs. By using the iterative techniques it is possible to stop the optimization when timing constraints are satisfied. This is useful in synthesis systems which generate performance constraints for components in the circuit. Section 5.1 will describe a procedure for generating these performance constraints. In the absence of such a capability, the designer has to accept the area penalty of using manually designed fast circuits even in situations where the timing constraints do not necessitate such a fast circuit.

There are several questions that still remain unanswered. The problem of expressing an incompletely specified Boolean function such that the resulting function can be represented with a circuit of small depth has not received much attention. Current techniques on the decomposition of Boolean functions all work with a sum-of-products representation of the function. Some variant of simplification that considers the complexity of the function along with the arrival time distribution of the inputs would be an important contribution.

Another problem that remains is to determine a schedule for reducing the circuit delay. During an iteration the proposed lower-bounding technique seeks to make the largest possible reduction in delay that it can guarantee. It may be the case that a slower rate of reducing delay may produce better results. There is a loose analogy with the cooling schedule used during simulated annealing.

To avoid interaction between different regions, the selection procedure seeks to provide improvement only at one segment along critical paths. By choosing to apply transformations at a lot of non-overlapping regions, fewer iterations may be required. In particular, recent work on the optimum clustering for delay under a pin-constraint [24] for each cluster, suggests that by using dynamic programming, the choice of transformations that lead to maximum delay reduction in a single pass can be made. The area overhead in such a technique could be large. Nevertheless, the approach deserves further investigation.

Finally, the focus of this chapter was to develop techniques that restructure the logic so that it has a small depth in terms of 2-input gates. Factors like fanout consideration and the restricted set of gates that are available in a cell-library were ignored. The next chapter addresses these issues and extends the ideas on the application of local transformations to handle mapped circuits.

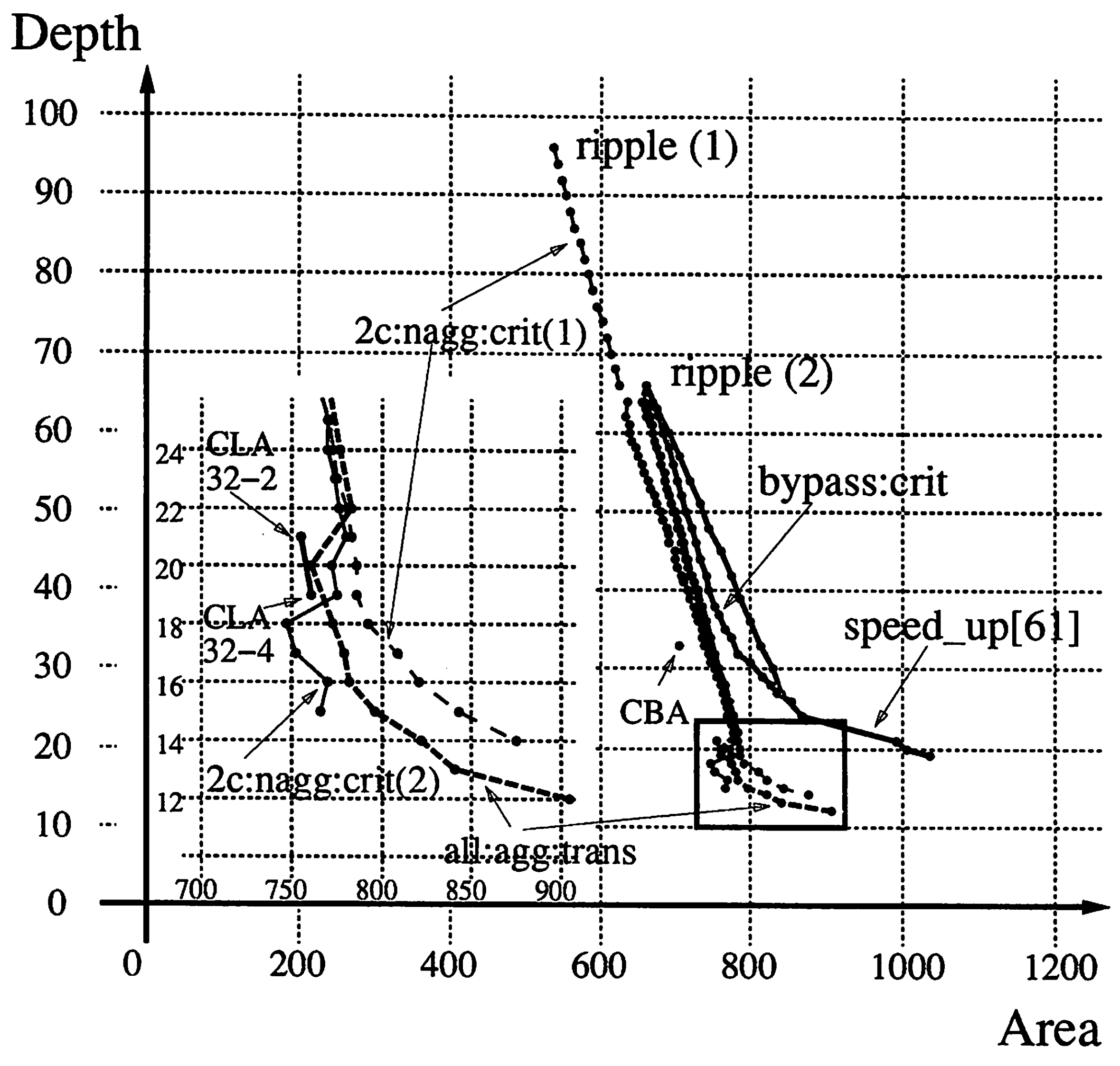


Figure 3.20: Optimization a 32-bit serial adder

Chapter 4

Technology-dependent optimizations

An abstract representation of the circuit function in terms of an optimized Boolean network has to be mapped onto a given set of gates (each with an associated function) to be realized as a circuit. At this stage there are electrical considerations that affect the circuit performance. The electrical characteristics of the devices (transistors), the size of the devices and the resistive and capacitive interactions have to be modeled.

Optimization to reduce circuit delay has had success in the area of transistor sizing [19]. The parameters to be optimized are the sizes of individual devices once the choice of the basic functions to represent the circuit has been made. The freedom to vary the size of each device makes the problems of placement and routing difficult and designs that use these algorithms are very specialized, full-custom circuits. For a large percentage of ASIC designs the preferred methodology is to use a set of precharacterized gates. Design styles based on standard-cells, gate-arrays and sea-of-gates fall into this category. In these styles, the freedom to size individual transistors is absent. However, since the delays of the gates is known, it is easier to develop a good structure for the underlying transistor-netlist. As an example, consider the implementation of a function AND-4 that computes the conjunction of 4 signals. Figure 4.1 shows two possible circuits that realize the same function in CMOS technology. The circuit on the left has three gates ($X = \overline{AB}$, $Y = \overline{CD}$ and $O = \overline{X+Y}$) while the one on the right has two gates ($Z = \overline{ABCD}$ and $O = \overline{Z}$). In a full-custom design style either alternative can be improved by the application of transistor sizing. It is

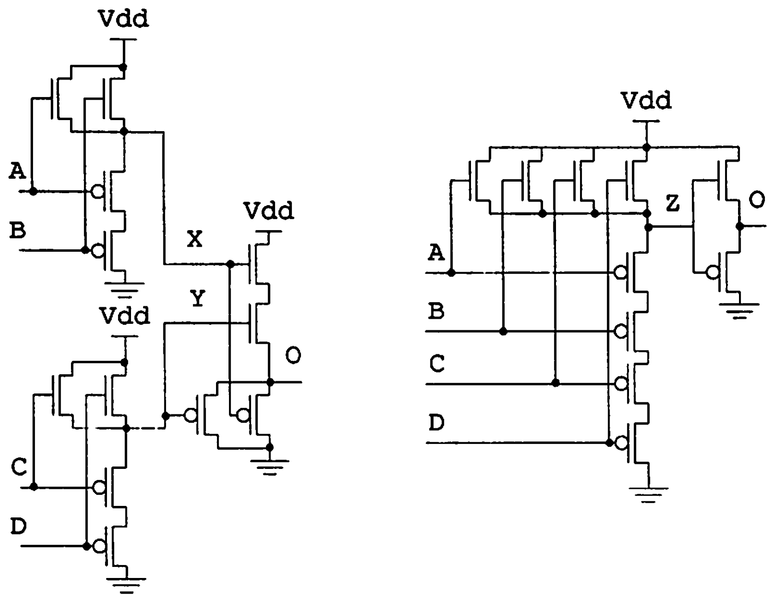


Figure 4.1: Alternative realizations of a 4-input AND function

difficult to select between the two alternatives a priori. In the case of standard-cell design style where a precharacterized library is available, it is easy to select the faster alternative. However, the user is restricted to using a fixed set of gates. It should be mentioned that most industrial cell-libraries provide different versions of gates. Thus there could be two versions of a gate — a large cell that has the capacity to drive large capacitive loads and a minimum size gate that can be used when the capacitive load is small or the function is not on any critical path. In the case of inverters it is typical to find several different versions.

The procedure to map a Boolean network onto a set of gates has been studied extensively. Many techniques have been proposed to minimize the area of mapped circuits. These include direct-mapping of the function onto a gate [32], pattern-matching techniques where a subject graph corresponding to the Boolean network is "covered" by a set of patterns representing the gates [30, 13] and Boolean-matching methods [40] that try to find gates in the library with the same Boolean function as nodes in the Boolean network. Among these techniques, tree-matching methods have been applied most successfully to performance oriented technology mapping [30, 67, 66]. Some of the extensions possible to that of [66] will be discussed in Section 4.1.

In circuits that have first been optimized for area it is not uncommon to find nodes that have a large fanout. The large fanout leads to increased capacitive loading of gates thereby reducing the circuit speed. Large fanouts may also lead to reliability problems. The large current required to charge large capacitive loads may lead to *metal migration* which results in failure of the circuit. Finally, in some technologies like bipolar circuits, there is an intrinsic limit on the number of gates that a given gate can drive. Section 4.2 addresses optimizations that improve delay in the presence of large fanout.

Previous work that uses restructuring to optimize mapped circuits includes that of [80, 66, 18]. In [80] the emphasis was on restructuring the trees in the network to reduce delay. The choice of trees to restructure was made based on a cutset procedure that weighted the trees. Tree-mapping and fanout-optimization were combined to produce a mapping procedure to minimize circuit delay in [66]. In the LATTIS system described in [18] the emphasis is to evaluate a set of transformations along the most critical path of the circuit. After evaluating a set of transformations, which retain the mapped nature of the circuit, the one with the greatest decrease in delay per unit increase in area is selected. This strategy is called "maximum-bang-for-the-buck". The amount of improvement is scaled by the number of inputs and outputs that the transformation affects.

The techniques for gate-selection and fanout-optimization are based on delays in the circuit which get modified during the course of the algorithms. This poses difficulties when trying to integrate the techniques. Section 4.3 combines the fanout-optimization and tree-covering with the machinery built in Section 3.2.3 to select local transformations. This approach integrates the two techniques more closely than that in any of the previous approaches.

Before proceeding to an evaluation of the techniques proposed for optimizing mapped circuits, it is important to be aware of the additional considerations that are present when dealing with mapped circuits. Recall that the technology-independent optimization used a representation consisting of 2-input gates with implicit inversions (inversions were not explicitly represented by inverters, just by changing the phase of the input to the node) and the circuit delay was simply the number of nodes on the longest path. For mapped circuits, the additional considerations are —

Complexity of the delay model For mapped circuits the delay model used is the *library* delay model (see Section 2.2.1). This model computes the pin-to-pin delay based on the load that is being driven by the gate. During optimization it is not adequate to simply count the levels of logic. The dependence of delay on the load (which may be affected by other transformations) makes the problem of predicting the performance improvement a difficult one. Furthermore, the delay for the rising and falling transition of a signal may be different. The asymmetric gate delays necessitate keeping track of a duple of values rather than a single value. The delay model also specifies the signal dependencies — whether the output has an inverting, non-inverting or unknown relation to a transition at the input — so that the computation of delay is not overly pessimistic.

Prescribed set of primitives Mapped circuits use gates from a cell-library to implement a Boolean function. While optimizing a mapped circuit it is essential to retain the mapped nature of the circuit to get accurate delay values. This is accomplished by performing a local mapping of the region that has been transformed. Appropriate timing constraints must be used during mapping to accurately reflect the environment around the function. The fine granularity and uniformity of the basic blocks during technology-independent optimization allows a fine control on the size of region being transformed. In mapped circuits, gates have different numbers of inputs. This makes it difficult to have a uniform size for the region being transformed. The approach we have used is to still use the depth as a guide for the region to transform. However, there is a user-specified limit on the size of the function being transformed. This allows us to avoid large run-times when large functions are generated for the application of a local transformation.

Explicit inversions Since a mapped circuit represents an implementable circuit it cannot have any implicit gates. The use of implicit inverters during technology-independent optimizations bypasses the problem of determining which polarity of function should be implemented. This problem of determining the phase (complemented or uncomplemented) of the function to implement is an additional aspect that adds to the complexity of optimizing mapped circuits.

With this background of additional problems that arise when mapped circuits are considered, the remainder of this chapter is devoted to techniques for reducing the delay of mapped circuits.

4.1 Optimization based on gate selection

Tree-based technology-mapping [13] has been used for both area and performance optimization. The procedure decomposes a Boolean network into a network of 2-input NANDgates to generate a subject graph. The subject graph is covered using a set of pattern graphs that are generated for the gates in the cell-library. The selection of a minimum-cost cover is based on dynamic-programming and produces the optimum solution when the subject graph is a tree and the objective is to minimize the area. The dynamic programming technique is extended [51] into a two-pass technique that produces the optimal delay implementation for a library with symmetric values for rise and fall delays.

When applied to directed-acyclic graphs, the heuristic of optimally mapping individual trees for the fastest implementation is not a good one. This is due to the fact that fast implementations contain high-power gates which result in increased capacitive loading. A good heuristic that combines tree-mapping and fanout-optimization to solve the performance-directed mapping of circuits is described in [66]. The heuristic makes multiple passes over the network alternating between mapping trees and buffering multiple-fanout gates. A limitation of the heuristic is its strong dependence on the set of trees that have been generated before the covering step. In addition, the problem of choosing the phase in which to generate each tree is not addressed. In this section techniques that attempt to improve performance by addressing phase-selection and tree-decomposition will be investigated.

Statistics gathered from a large number of circuits shows that the average size of a tree in an optimized circuit is small. From data presented in [32], the average tree has a depth of 2.44 two-input gates and is mapped into 1.93 gates after technology-mapping. A small tree has relatively little chance of being mapped into a faster structure. As the size of the tree increases, the difference between a good and a bad implementation increases. Furthermore, small trees result in relatively large number of multiple-fanout gates along critical paths. This leads to larger loads and hence a slower circuit. The creation of multi-fanout points also creates a decision point — whether to implement the signal or its complement. In tree-based mapping, the choice of the phase of a function is rather arbitrary. By having larger trees along the critical path fewer decisions have to be made, reducing the chance of generating a poor circuit.

Larger trees, generated by duplication of logic, can be remapped for smaller delay. This idea was explored during the technology-mapping phase itself (Section 4.5.2 of [66]) by allowing overlap between trees. Due to the changes in fanout that result from overlap, the prediction of delays is difficult. On the average, by using heuristic prediction methods a delay reduction of 9% with an area increase of 44% was obtained. To reduce the area overhead, overlap was allowed only when a node had fanout less than 5. This heuristic reduced the area overhead to 28% and provided 8% reduction in delay. It should be noted that allowing partial overlap of trees results in the creation of new fanout points. Furthermore, when tree-overlaps are implemented as part of technology-mapping, the delay data is inaccurate since the gate selections have not been finalized.

With these observations, and using the basic paradigm of applying local transformations from the previous chapter, it is possible to improve the result of technology-mapping. After mapping the critical path is known precisely. Trees along the critical-path may be combined into larger trees and remapped to reduce delay. The problem is to identify what trees to collapse into one another so that a delay improvement is achieved at a small cost in area. This problem is similar to the one addressed when technology-independent optimization was described.

The local transformation for tree-restructuring consists of re-mapping the selected sub-circuit for minimum delay. The scope of the transformation may be chosen based on depth of the region or by an alternative technique. The use of tree-based mapping naturally suggests that the scope be restricted to be a tree. The scope is determined by expanding the tree to include nodes from trees in its fanins. The entire fanin tree is duplicated to avoid the creation of new fanout points. This may have the disadvantage of large area-overhead but reduces the number of multi-fanout points along the critical path, alleviating the problem of selecting the phases of the trees. An area/delay tradeoff can be accomplished by controlling the number of trees, along the critical paths, that are combined to generate the scope for re-mapping.

As the number of trees duplicated along the critical path increases, there is greater duplication of logic. This results in a larger area penalty. This is evident from the results presented in Table 4.1. The delay improvement is 5% when the scope is 1 more tree or 2 more trees. For some examples there is a large improvement in delay. These examples are emphasized. Even though the delay improvement is not as much as the tree-overlap heuristic of [66] the area overhead is very small. The best improvement is 19% for tree-1

Example	delay-map		tree-1		tree-2	
	Area	**Delay**	**Area**	**Delay**	**Area**	**Delay**
C1355	1333	19.66	1.00	1.00	1.00	1.00
C1908	1485	31.15	1.06	0.92	1.03	0.95
C2670	2043	21.91	1.02	0.97	1.01	0.98
C3540	4003	36.76	1.01	0.99	1.05	1.00
C432	599	26.60	1.10	0.98	1.28	0.96
C6288	8132	91.08	1.00	1.00	TO	TO
C7552	6243	30.60	1.00	0.98	1.01	0.97
ampbpreg	2355	18.07	1.01	0.97	TO	TO
ampbsm	2060	16.27	1.02	0.90	TO	TO
amppint2	1506	20.80	1.11	0.87	1.03	0.91
ampxhdl	890	13.19	1.05	0.95	1.10	0.97
b12	220	5.66	1.00	1.00	1.00	1.00
b9	343	8.17	1.03	0.88	1.01	0.90
cordic	218	9.70	1.00	1.00	1.00	1.00
cps	3562	17.92	1.01	0.81	1.02	0.78
dalu	2624	21.19	1.02	0.91	1.03	0.90
des	9866	18.41	1.00	1.00	1.00	0.99
dflgrcbl	858	10.50	1.00	1.00	1.00	1.00
duke2	1294	17.10	1.11	0.84	1.06	0.87
ex1010	6223	16.50	TO	TO	TO	TO
ex4	1460	9.94	1.01	0.99	1.01	0.99
fconrcbl	657	12.24	1.02	0.87	1.13	0.78
k2	3373	19.82	TO	TO	TO	TO
kcctlcb3	644	9.07	1.04	0.94	1.03	0.95
misex2	340	7.30	1.00	1.00	1.00	1.00
misex3c	1591	27.79	1.00	1.00	1.13	1.00
pdc	1204	11.98	1.02	0.96	1.02	0.97
rd84	402	10.78	1.00	1.00	1.00	1.00
rot	2155	18.22	1.01	0.93	1.09	0.92
sbiucbl	754	15.29	1.02	1.00	1.04	0.99
spla	1952	15.89	1.01	0.86	1.08	0.85
t481	2165	17.27	1.02	0.95	1.02	0.97
tfaultcbl	506	6.58	1.02	0.95	1.02	0.96
G.MEAN			1.02	0.95	1.04	0.95

delay-map	Circuit mapped for minimum delay (`map -m 1 -A`)
tree-1	Combining 1 tree along the critical paths and remapping
tree-2	Combining 2 trees along the critical paths and remapping
Area	area of the circuit (MCNC `lib2` data divided by common divisor 464)
Delay	delay of the circuit (MCNC `lib2` data in nanoseconds)

Table 4.1: Effect of tree duplication on circuit delay

and 22% for tree-2. Surprisingly, duplication of more trees leads to smaller improvement for some examples. An observation that may explain this anomaly is that the duplication of logic leads to increased fanout in the circuit. So it seems reasonable to follow up the remapping of larger trees with a round of fanout-optimization. It should be mentioned that fanout-optimization was used during the minimum-delay mapping [66] to drive large fanout loads. For application in local transformations a technique is required to modify the fanout-trees in a mapped circuit to account for the changes that have been made, e.g. duplication of gates. The next section addresses the fanout correction problem.

4.2 Optimization based on fanout buffering

In addition to the choice of gates used to implement functions, the delay of the circuit depends on the configuration of gates chosen to distribute a signal to its destinations. In optimized circuits it is not uncommon to have nodes that have a large fanout. After mapping, these nodes drive large capacitive loads which leads to long charging times. This reduces the circuit speed and may even result in erroneous circuit operation. The introduction of buffers not only reduces the circuit delay, it also makes the delay values more accurate (since the loads in the circuit after buffering would be within the range of load values used to characterize the library models).

A gate may be required to drive different capacitive loads at several destinations in either complemented or uncomplemented form. The time at which a signal is required at each destination is known. The problem is to derive a distribution tree that is able to provide the signal at its destinations before the required times by using inverters and buffers from the cell-library. There are two problems that need to be considered — " **What order should the gates be visited?**" and "**How is a good fanout-tree constructed?**". These are explored in following sections.

4.2.1 Buffering strategies

This section addresses the question "**What order should gates be visited for buffering?**". Section 3.7 of [66] also investigates this question. It is shown that optimizing the fanout-trees in topological order from outputs to inputs is the best strategy to achieve the smallest delay without consideration of area. As reported in Section 3.8.3 of [66] the area penalty of building the best fanout-trees everywhere is large (on the average the area

```
  area-based_buffering($\eta$) {
      order nodes in topological order, outputs to inputs
loop: for each unvisited node $v$, visited in order {
          mark $v$ as visited
          if ($v$ is critical AND buffering improves delay at $v$) {
              delay_trace($\eta$);
              goto loop;
          }
      }
  }
```

Figure 4.2: Critical-path based buffering strategy

increase is 51% compared to the minimum-area mapping). To recover area after fanout-optimization, the non-critical sections are buffered to reduce the area while ensuring that the delay does not increase. The area-recovery phase is not guaranteed to be optimal even though the area is reduced substantially (the average area increase after applying area-recovery is only 9% over the minimum-area mapping). To summarize, the approach of [66] builds the best trees at all places and then recovers area by down-sizing non-critical gates. We call this method **delay-based** buffering.

An alternative strategy called **area-based** buffering is described in Figure 4.2. It builds fanout-trees only along the critical paths (critical paths may change during the process). The rationale behind this strategy is to incur an area penalty only where it is needed to reduce delay. It starts with a minimum-area mapping and successively reduces the delay at the expense of increased area. This multi-pass strategy proceeds from the output towards the inputs and builds the fanout-tree for a gate along the critical path that has not yet been considered. Once a gate has been considered, the delay data is updated and another gate along the critical path (possibly a different path from the previous one) is chosen. At each stage the gate that has not been visited and is closest to the output is chosen. By proceeding from the outputs to inputs, the required-time data at internal nodes is updated to account for the trees generated earlier. This strategy reduces to the optimum strategy for minimum delay if the required time constraints on the outputs are very strict.

Then the buffering proceeds from the outputs to inputs in topological order under both strategies.

To compare *delay-based* buffering with *area-based* buffering proposed here, we remove all timing constraints from the outputs so that both procedures attempt to reduce the delay of the circuit. Also, the same local fanout-optimization (described in Section 4.2.2) is used in both procedures.

The circuit is mapped for minimum-delay without any fanout-optimization. This is achieved by disabling the fanout-optimizations `fanout_alg noalg` and then mapping the circuit for minimum delay using the SIS commands `map -m1 -A`. The option `-m1` minimizes delay and the option `-A` keeps the area small. *area-based* buffering is used to improve the circuit at the expense of increasing the circuit area. This is compared with the result of applying *delay-based* buffering using the SIS commands (`fanout_alg top_down; map -m1 -A`). Table 4.2 shows the results of this experiment

The results of this experiment are surprising. *area-based* buffering produces smaller delays than *delay-based* buffering, while the *delay-based* technique outperforms the more selective *area-based* buffering in terms of circuit area. On closer examination the shortcomings of each of the strategies in achieving the desired objective can easily be explained.

Fast circuits are constructed at fanout points during *delay-based* buffering. By doing so high-power gates are used increasing the capacitive loads that the fanin signals have to drive. Subsequent area recovery may reduce the load driven by the critical signals by down-sizing some non-critical gates. Thus area-recovery often improves the circuit delay. However, the area recovery phase is not specifically targeted to improving circuit performance by down-sizing non-critical gates. In the case of the critical path based approach, the initial circuit is optimized for area so that all gates have lower capacitive loads. Buffering along the critical path may increase the gate sizes for some gates. The non-critical signals that do not affect circuit performance are not sized, and offer a smaller load compared to the other strategy, leading to superior delay reduction.

The area recovery phase of *delay-based* buffering has the capability of down-sizing non-critical gates to reduce circuit area. This capability is not present in *area-based* buffering which only builds fanout-trees along non-critical paths. Since the fanout trees built on the initial critical path may not lie on the final critical path they may be down-sized to reduce circuit area. The obvious solution is to use the area-recovery procedure after the critical-paths have been buffered. The machinery to carry out this experiment is not in place.

Example	Delay-based		Area-based		Ratio	
	Area	**Delay**	**Area**	**Delay**	**Area**	**Delay**
C1355	1176	20.35	1220	20.61	1.04	1.01
C1908	1128	32.58	1213	31.93	1.08	0.98
C2670	1629	22.48	1678	22.52	1.03	1.00
C3540	2806	37.69	2937	37.32	1.05	0.99
C432	477	27.90	532	26.99	1.12	0.97
C6288	6925	92.05	6847	92.64	0.99	1.01
C7552	4846	32.96	5053	32.37	1.04	0.98
ampbpreg	1814	20.04	1955	19.91	1.08	0.99
ampbsm	1589	16.93	1667	16.61	1.05	0.98
amppint2	1132	21.64	1188	21.10	1.05	0.98
ampxhdl	730	13.88	764	13.59	1.05	0.98
b12	179	5.74	190	5.69	1.06	0.99
b9	288	8.35	289	8.35	1.00	1.00
cordic	183	9.70	191	9.70	1.04	1.00
cps	2583	18.84	2646	18.95	1.02	1.01
dalu	2012	22.42	2117	21.90	1.05	0.98
des	7478	20.40	7982	19.94	1.07	0.98
dflgrcb1	696	10.55	722	10.50	1.04	1.00
duke2	912	17.59	962	17.51	1.05	1.00
ex1010	5081	17.28	5602	17.11	1.10	0.99
ex4	1122	10.12	1163	10.22	1.04	1.01
fconrcb1	537	12.96	560	12.54	1.04	0.97
k2	2677	21.03	2790	20.18	1.04	0.96
kcctlcb3	531	9.89	576	9.71	1.08	0.98
misex2	271	7.60	300	7.43	1.11	0.98
misex3c	1060	29.39	1108	28.81	1.05	0.98
pdc	866	12.16	924	12.33	1.07	1.01
rd84	294	11.73	313	11.62	1.06	0.99
rot	1551	18.44	1596	19.09	1.03	1.04
sbiucb1	550	15.57	578	15.99	1.05	1.03
spla	1467	17.03	1500	16.38	1.02	0.96
t481	1690	18.12	1806	17.34	1.07	0.96
tfaultcb1	440	6.63	452	6.58	1.03	0.99
G.MEAN	-	-	-	-	1.05	0.99

Delay-based	Building best fanout-trees, followed by area recovery
Area-based	Building fanout-trees along critical paths only
Ratio	Using the *delay-based* buffering as the base value
Area	Area of the circuit (MCNC `lib2` data divided by common divisor 464)
Delay	Delay of the circuit (MCNC `lib2` data in nanoseconds)

Table 4.2: Comparison of different buffering strategies

However, using area-recovery along with *area-based* buffering should, in principle, lead to smaller area.

4.2.2 Top-down buffering algorithm

In the above experiment, a heuristic technique is used to build the buffering trees. In this section we describe the algorithm **top_down** used to build fanout trees during the above experiment. The algorithm follows a divide-and-conquer strategy during which the set of fanout signals is partitioned into subsets, and the process is recursively applied on the smaller problems. This technique bears similarity to the technique used by Paulin *et. al.* to decompose gates with high fanin [48].

Assume that the signal generated by gate g is distributed to several destinations. The **destination-set** of gate g is FO and it consists of two parts — FO^+ and FO^-. FO^+ consists of gates to which the signal g is distributed. The elements in FO^- receive the complemented signal $\overline{g}$. The inverted signal is generated by an inverting buffer denoted by g_I. Let the cardinality of the fanout sets be $|FO^-| = m$ and $|FO^+| = n$. For each destination h in the destination-set FO, we are given the values c_h and r_h. c_h is the capacitive load of pin h and r_h is the required time for pin h. We use the superscripts + and - to denote values associated with the sets FO^+ and FO^- respectively. The fanouts in both partitions are sorted by their required times i.e. $r_1^+ \leq r_2^+ \leq \cdots \leq r_n^+$ and $r_1^- \leq r_2^- \leq \cdots \leq r_m^-$. Assume that there are M inverting buffers in the target technology to be used for buffering. Non-inverting buffers can be made up as cascades of inverting buffers. The problem is to build a fanout-tree at the output of gate g such that the required time at the input of gate g exceeds a user-specified target value TR.

The algorithm **top_down** builds a fanout-tree by selecting a delay reducing transformation at the current gate depending on the distribution of the loads and required times. The transformations aim to reduce the required time at the input of the gate. The three transformations used are illustrated in Figure 4.3. They are **repower**, **trans1** and **trans2** and we denote the required time at the input of gate g after the application of each transformation by $R0$, $R1$ and $R2$ respectively.

The algorithm top_down is described in Figure 4.4. We first evaluate the required time $R0$ resulting from choosing the best available version of the gates g and g_I (the *repower* transformation). Having done the obvious powering-up of gate g, the distribution of

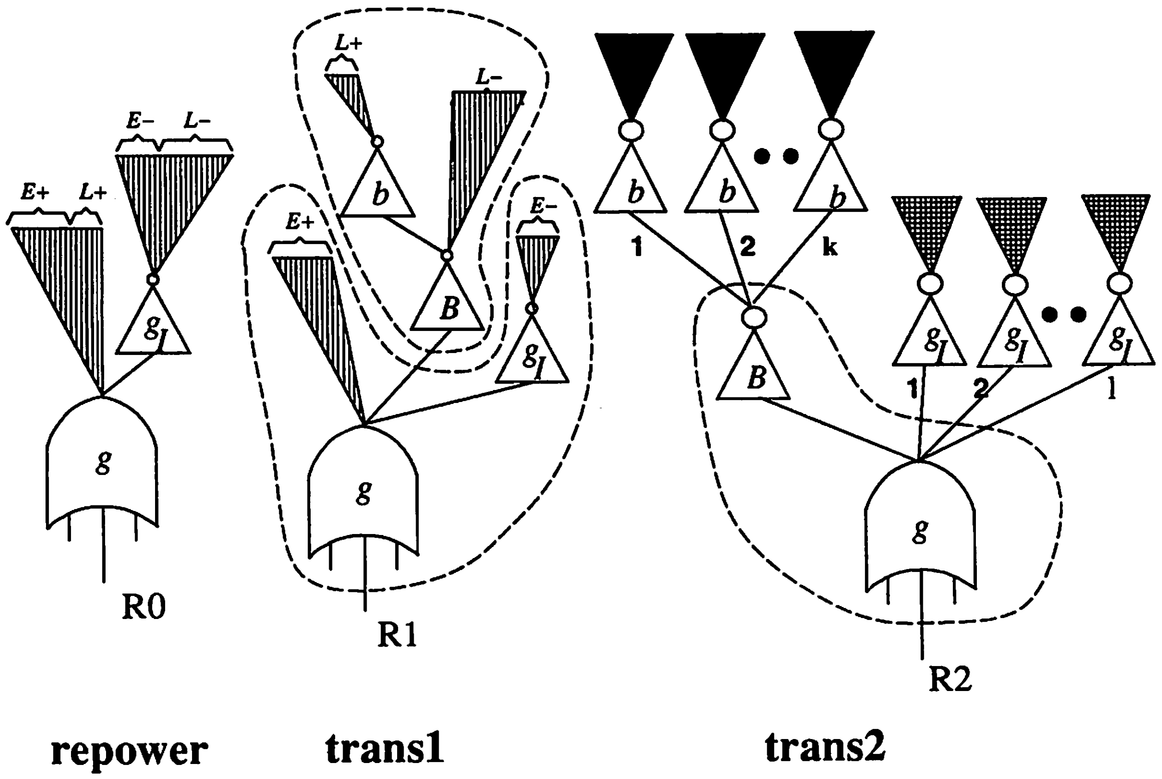


Figure 4.3: The basic buffering transformations

destinations into two sets of early and late required destinations is evaluated according to *trans1*. The root gate and inverter g_I are used to drive the destinations where the signals are required early. The load driven by these gates is reduced. $R1$ is the maximum required time that is achieved by using this transformation for any partition into early and late destinations and choice of buffers. If there is no improvement as a result of *trans1* ($R1 < R0$) we conclude that the required times have a small spread. In that case a balanced redistribution according to *trans2* is evaluated. The best configuration of added buffers is found and the maximum value for the required time $R2$ is computed. Transformation *trans2* is accepted only if the required time improves, $R2 > R0$. Having selected either *trans1* or *trans2* , the algorithm will recur on the new nodes created and then again on the root node. The routine *recursion_will_help*() is used to prune recursive calls that will not help improve the required time at gate g.

```
top_down( $g$, $g_I$, $FO^+$, $FO^-$, $TR$) {
    $R0$ = evaluate_repower($g$, $g_I$, $FO^+$, $FO^-$)
    if ($R0 > TR$) return
    ($R1, E^+, E^-$) = evaluate_trans1($g$, $g_I$, $FO^+$, $FO^-$)
    if ( $R1 > R0$) {
        create_trans1_configuration($g$, $g_I$, $E^+$, $E^-$)
        if ($R1 < TR$ AND recursion_will_help()) {
            top_down($B$, $b$, $L^-$, $L^+$, $TR$)
            top_down($g$, $g_I$, $E^+ \cup \{B\}$, $E^-$, $TR$)
        }
    } else {
        ($R2, k, l$) = evaluate_trans2($g$, $g_I$, $FO^+$, $FO^-$)
        if ($R2 > R0$) {
            create_trans2_configuration($g$, $g_I$, $k$, $l$)
            if ($R2 > TR$)
                top_down($g$, $B$, $\bigcup_l g_I$, $\bigcup_k b$, $TR$)
        }
    }
}
```

Figure 4.4: Alogoritm **top_down** to create fanout-trees

Whenever the transformation *trans1* is applied two sub-problems result. Application of *trans2* results in the creation of a smaller problem. The sub-problems created are shown in Figure 4.3 as regions enclosed in dashed lines. The algorithm can be visualized as a recursive tree. This recursive tree is useful in analyzing the complexity of the algorithm.

The gate g that is being buffered is part of a network and consequently the point where the delay value is computed must be clearly defined. While improving the required time at the input of gate g it is possible to increase in the delay through a gate f that is a fanin of g. This may happen if g presents a larger input capacitance to f. To compute the actual benefit of a transformation we must account for the non-zero drive ($\beta(f)$) of gate f. This captures the relevant environment for gate g and allows accurate evaluation of the improvement in required time. Different values for $\beta(f)$ may lead to different transformations being accepted. We will now describe how the transformations are evaluated and what is the criterion used for accepting a transformation.

We first evaluate the saving resulting from choosing the best available version of the gates (the *repower* transformation). This is the obvious way to get a faster circuit when different versions of the gate being buffered are available in the target technology. This transformation does not change the structure of the fanout-tree (all signals in FO^+ are still driven by gate g and those in FO^- by g_I). The attempt here is to reduce the delay by choosing replacement gates for g and g_I that have the capability of driving large loads. The *repower* transformation results in a required time $R0$ being achieved at the input of gate g. If this is adequate ($R0 > TR$) the algorithm terminates. If gate g does not have a replacement we may decompose the gate into smaller gates which usually have better drive capabilities (due to fewer series transistors driving the output). We may also consider a sub-network η_s as the root of the fanout-tree (rather than a single gate g) and can apply techniques of technology-mapping to find the best replacement sub-network for η_s driving the specified load. Since we want to isolate the performance improvement that can be achieved by buffering alone we restrict the root of the fanout-tree to be a single gate.

Applying *trans1* involves partitioning the sorted set of destinations into two sets, E and L. The set E consists of destinations where the signals is required early and the set L contains the late required (less critical) destinations. Denote by E^+ and E^- the subsets of FO^+ and FO^- that comprise the early destinations. These are driven by the gates g and g_I respectively as shown in Figure 4.3. The late required subsets are therefore $L^+ = FO^+ \backslash E^+$ and $L^- = FO^- \backslash E^-$ which are driven by inverting buffers b and B respectively. For each partition we have different choices for the buffers g_I, b and B. For a specific configuration the required time at the input of gate g is denoted by $R1(E^+, E^-, b, B, g_I)$. We denote by r_{g_I}, r_b and r_B the required times at the inputs of inverting buffers g_I, b and B respectively.

The choice of the added buffers b and B, the gate g_I and the sets E^+ and E^- is made so as to maximize the required time at the input of gate g. The best required time that can be achieved by an application of *trans1* is denoted by $R1$. If an application of *trans1* enables us to meet the target required time TR, we implement the transformation and exit the recursion. If the target required time is not met and $R1 > R0$ (there is a saving compared to the *repower* transformation) we accept the transformation and recur on the problems of buffering gate B and then gate g. Recursive calls that do not help to improve circuit performance are eliminated by the routine *recursion_will_help()*. An example of pruning recursive branches occurs in transformation *trans1*. Whenever the required time r_B at the input of buffer B exceeds $\min[r_1^+, r_{g_I}]$ we do not need to recur on B. Buffering node B in an effort to increase r_B beyond its current value does not influence the required time at the input of gate g. At each stage of the recursion we know the maximum amount that we need to save (derived from the target required time TR) and recursion terminates when this saving is achieved. These pruning strategies result in a significant reduction in the run-time of the fanout correction process.

By pre-computing the cumulative capacitance for partitions of FO^+ and FO^-, the required time at the input of gate g can be computed in constant time for a given configuration. The precomputation can be performed in $O(m+n)$. Since there are M^3mn configurations corresponding to different choices of buffers and partitions, $O(M^3mn)$ computations are needed to find the maximum value of $R1$. Let the time to recursively apply the unbalanced decomposition (the transformation *trans1*) for a problem with m signals in negative phase and n signals in the positive phase be denoted by $T1(m,n)$. If $|E^+| = s$ and $|E^-| = t$, the recursive application of *trans1* can be captured using the equation —

$$T1(m,n) = O(M^3mn) + T1(n-s, m-t) + T1(t,s)$$

The solution to this equation that results in the largest run-time is $T1(m,n) = O(M^3m^2n^2)$ for $s = t = 1$. If $n = O(N)$ and $m = O(N)$ then $T1(N) = O(M^3N^4)$. We can interpret N as the size of the destination-set.

When there is no saving by using the transformation *trans1*, one can infer that the required times in the sets FO^+ and FO^- are clustered together. A balanced tree as per transformation *trans2* is attempted to improve the required time at the input of g. The transformation evaluates a multi-way split of the positive and negative destinations to generate groups of destinations with similar cumulative capacitance. The required time $R2$ at the input of gate g is computed for a k-way balanced partitioning of FO^+ and a l-way partitioning of the negative destination-set FO^-. Each k-way division of the positive destinations is driven by an inverting buffer b. The inverter B is used to drive these k added inverters. The l-way partition of FO^- requires that l copies of the inverting buffer g_I be used to drive the partitions. For a given (k,l)−way decomposition of the destination-set FO (shown in Figure 4.3) we evaluate the required time $R2(l,k,g_I,b,B)$.

With appropriate precomputation of cumulative capacitance for partitions we need

$O(M^3mn)$ computations to find the maximum value of $R2$. Let the time to recursively apply the transformation *trans2* for a problem with m signals in negative phase and n signals in the positive phase be denoted by $T2(m,n)$. $T2(m,n)$ can be computed by solving the equation —

$$T2(m,n) = O(M^3mn) + T2(k,l)$$

where $2 \leq l \leq \frac{m}{2}$ and $2 \leq k \leq \frac{n}{2}$. The solution to this equation which yeilds the largest value for $T2$ is $T2(m,n) = O(M^3mn\log(mn))$.

The more expensive recursion occurs when *trans1* is applied. This dominates the computational complexity. The overall complexity of the buffering procedure based on applying delay improving transforms is therefore $O(M^3m^2n^2)$. The algorithm **top_down** is a polynomial-time heuristic. In practice, due to the pruning strategies that avoid needless recursions, it is fast. At each stage in the recursion it preserves a valid buffer tree which is better than the one at the previous stage. This allows the buffering to be done on a *need-to* basis. The amount of improvement desired can be specified as TR and the recursion stops as soon as the desired saving has been obtained. This leads to improved run times and prevents the unwarranted area increase that could result from building the best fanout-tree. If the best fanout-tree is desired the target required time TR is set to be a large number.

4.2.3 Comparison of buffering algorithms

Chapter 3 of [66] describes a range of buffering techniques. These techniques are heuristics that sort the required times of the destinations and build *ordered trees*, i.e. a buffer, x, cannot drive another buffer, y that has a destination which has a smaller required time that the destinations driven by x. The main difference is that the techniques of [66] are bottom-up approaches. They determine a set of destinations to be driven by a buffer, treat the buffer input as a new destination, and proceed by dynamic programming to generate the fanout trees. By restricting the type and number of trees, different heuristic algorithms are obtained. For convenience these are briefly described.

balanced The required times are ignored and a balanced tree is generated to drive the destinations. The balanced tree has a depth of 2.

bottom_up The destinations that are required the lastest (least critical) are combined based on the combinatorial merging algorithm [22]. The number of destinations to be driven by a new buffer is determined heuristically based on the buffer being evaluated.

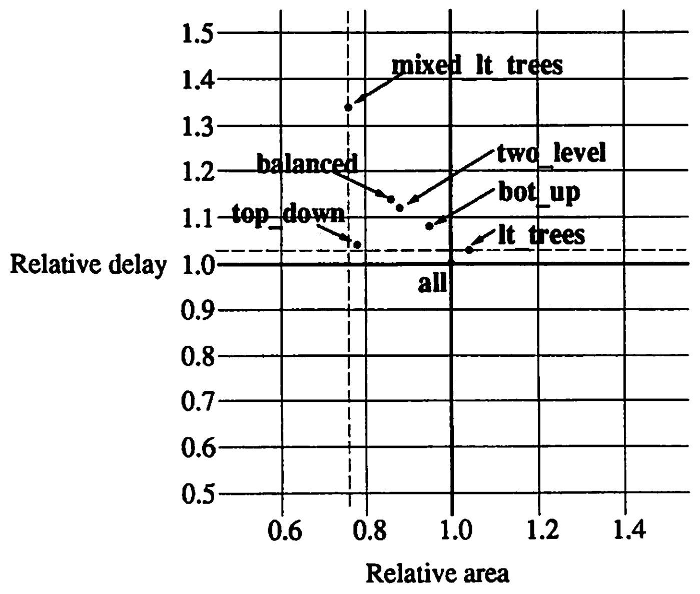
lt_trees LT-trees are a restricted class of fanout-trees. They are useful since a polynomial-time, dynamic programming algorithm can be used to determine the optimum fanout-tree in this class. They use non-inverting buffers. Thus a general problem is decomposed into two sub-problems, one for the complemented destinations and another one for thw uncomplemented destination sets, that are solved independently.

mixed_lt_trees This is an extension to the LT-trees that handle sinks of different polarities directly and use both inverting and non-inverting buffers.

two_level This is similar to the **balanced** algorithm. However, there is consideration of loads and required times when the destinations are assigned to the single level of buffers added between the root and the destinations.

For the technology-mapping command in SIS, it is possible to specify the type of fanout-optimization to be used. The best results are obtained when all the heuristic techniques are enabled. In that case, the technology-mapper can apply all the heuristics and then pick the one that best fits the area/delay requirement at hand. By specifying a single algorithm, the algorithms can be compared in the same setting. Figure 4.5 shows the mean area and delay gains (geometric means) when the circuit obtained using each algorithm in isolation is compared against the circuit where all algorithms were available.

For all the example circuits, the area and delay are normalized to the area and delay when all the algorithms are used. By using all available algorithms, the solution with the smallest delay is obtained. When a single algorithm is used, the delay is always greater. Dashed lines indicate the minimum-area and the minimum-delay solution obtained by applying a single algorithm. Among individual algorithms, the algorithm *lt-trees* that uses only non-inverting buffers and builds separate fanout-trees for the inverted and non-inverted destinations provides the best delay solution. However, the area penalty is large. In fact it is the only algorithm that results in a larger area than the minimum-delay solution. This is to be expected since the non-inverting buffers in this library are composed of cascade of inverters. Also, the isolation of positive and negative polarity destinations leads to large area overhead. One would therefore expect that the algorithm *mixed_lt_trees* that uses inverters and interleaves the complemented and uncomplemented destinations should reduce the area overhead. This is indeed the case (area overhead reduces from +4% to



	balanced	bot_up	lt_trees	mixed_lt_trees	top_down	two_level
Area	0.86	0.95	1.04	0.76	0.78	0.88
Delay	1.14	1.08	1.03	1.34	1.04	1.12

Figure 4.5: Comparison of fanout algorithms

-24%), however the performance of the circuit degrades considerably (from 3% worse to 34% worse). Algorithm *top_down* proposed in this section provides a solution that has very low delay penalty (4% worse) as well as small area (-22% less area than the minimum-delay solution). It is the only algorithm that performs well with regard to both and delay considerations. The other simple algorithms have characteristics inferior to the top_down algorithm, both in terms of area and delay.

At the start of this section, tree-duplication was listed as a source for increase in fanout load. The delay-optimized circuits described in Table 4.1 were subjected to fanout-correction using the top_down algorithm. For single tree duplication followed by fanout correction, the area increase was 3% for a delay decrease of 6% compared to the minimum-delay mapping without duplication. When up to two trees can be duplicated, the area increase after fanout correction is 5% for a delay reduction of 7%. Thus by using fanout correction to overcome the additional fanouts created by tree-duplication further improvement in delay can be obtained.

Till now we have performed optimizations that have been limited to trees. This restriction is removed when we consider the application of general restructuring techniques in the next section.

4.3 Applying local transformations to mapped circuits

The traditional approach towards logic optimization, both for area and delay, is to perform appropriate technology-mapping on Boolean networks that have been optimized at the abstract, technology-independent level. This approach is justified when the metric used during technology-independent optimization is a good predictor for the mapped circuit as in the case of area optimization. The number of literals in the factored-form of a function is a good estimate of the circuit area. As was alluded to in Table 2.1, the predictors for delay perform poorly. This poses a big problem as evidenced by the following experiment. Optimized circuits from Table 3.7 are mapped for minimum delay. One would expect that the substantial difference in depth at the technology-independent level would be reflected in similar differences in delay for the mapped circuits. However, as Table 4.3 shows, this is not the case. It appears that the advantage based on smaller depth has been eroded during technology-mapping. In fact, for 19 circuits the clustering approach produces smaller delays, while iterative improvement provides smaller delays in 11 circuits. Both techniques have some clear winners (20 % better than the other).

Several factors could have contributed to this unpredictability. Considerations of fanout were ignored at the technology-independent stage. The structure and interconnection of trees (that affects the technology-mapping) was uncontrolled and not considered during optimization. The application of transformations cause logic duplication and the effect of this on delay was also not considered. One approach to overcome the unpredictability of delay between the Boolean network and the mapped circuit is to apply local transformations on mapped circuits.

Previous sections described methods that can be used to augment the performance of technology-mapping — tree-duplication and fanout-optimization. Both involve repeated application of a specific local transformations on the circuit to improves the circuit performance. Each of these techniques addresses a different aspect of circuit performance.

Tree-duplication can be viewed as a transformation that tries to generate a signal as early as possible, i.e. to reduce its arrival time. It is an example of an **arrival-time-based** transformation. Fanout-optimization aims at maximizing the required time at the input of a gate and is classified as a **required-time-based** transformation.

The use of different delay information by restructuring and fanout techniques makes it difficult to apply them simultaneously. Tree duplication is based on arrival time data that may change when some gate in the circuit is buffered. Similarly, buffering is based on required times and loads that vary with different choices of gates. Applying one type of technique on part of the circuit and the other elsewhere results in interactions that are hard to predict and evaluate. An attempt to combine fanout-optimization and gate selection was made in the context of technology-mapping [66]. Alternate passes of tree-covering and fanout-optimization were used. During tree-covering, the load at fanout points was estimated using fanout heuristics. Experimental results showed no significant improvement when tree-covering and fanout-optimization were applied more than once.

Rather than apply the two classes of techniques during separate passes, this section examines an approach that combines both types of optimizations in a single iteration. For this we use the machinery developed in Chapter 3 to apply local transformations. The set of transformations that work on mapped circuits replace a section of the circuit with another structure composed of gates from the cell-library

All the local transformations described in Section 3.1 may also be viewed as mapped transformations if the optimized sub-circuits are implemented in the target library. This is accomplished by mapping the optimized scopes for minimum delay. Since mapping is performed for minimum delay, a delay improvement may be achieved even when the scope is not transformed. The mapped transformation that does not perform any optimization is named **noalg**. It is intended to be used to replace a slow implementation of a function with a faster one by changing the gates used to implement it. In addition to the transformations that try to reduce the arrival time there are local transformations that operate on required time data. They try to increase the required time at the input of their scope. These include the **fanout** transformation that builds a fanout-tree according to the algorithm *top_down* at the output of a gate to improve the required time at the gate input. Another local transformation is called **duplicate**. It is similar to *fanout* but it splits the fanouts into two groups and drives the critical destinations with a copy of the function. The required-time-based transformations were not used at the technology-independent level due to lack of accurate delays as well as the absence of inverters and buffers of different strengths.

For arrival-time-based transformations the scope is determined by one of the strategies — *critical*, *transitive* or *compromise* — just as it was at the technology-independent stage. However, due to the non-uniform size of gates, the size of the scope may vary considerably from node to node. When the scope spans a multiple-fanout gate, some logic is duplicated resulting in new multi-fanout gates in the circuit. To avoid the addition of new multiple-fanout gates in mapped circuits, the scope may be determined by considering entire trees when traversing the logic along the critical path and in the transitive fanin of the root gate. This strategy is called **tree-based** and was used in the experiment in Section 4.1 where tree-duplication was proposed as a local transformation. We can consider the duplication of one tree to be the same as applying the *noalg* transformation on a scope determined by the *tree-based* strategy with a depth equal to one tree. For the required-time-based transformations, *fanout* and *duplicate* the scope consists of the gate and all its fanouts. This scope is transformed by either creation of a fanout-tree or by duplicating the root gate.

Once the different transformations have been evaluated on their scopes, the guaranteed saving at the output can be computed by using Equation 3.1. This computation can accommodate required-time-based optimizations as well. The delay saving at gate n as a result of a required-time-based transformation is the improvement in required time at the input of node n. $A(n)$ represents the area increase of the transformed scope. During fanout-optimization, the gate at node n might be replaced by a faster gate which in turn leads to increased load on the fanins of node n. It is important to account for this while computing $D(n)$, the amount of delay improvement.

The selection procedure is similar to that for the technology-independent representation. A maximum achievable slack s^* is computed based on the predicted improvement. A selection-function $\mathcal{F}$ is derived by propagating the desired savings backward and the propagating selection functions forward (see Section 3.2.4). Satisfying assignments of $\mathcal{F}$ represent choices of transformations that provide the required improvement. However, there is one important factor which complicates matters for mapped circuits. The technique used to determine s^* assumes that the transformations are made independent of other changes in the network. When the assumption is valid the minimum-cost satisfying assignment of the selection function is the appropriate selection set. For mapped circuits the transformations do affect each other and the independence assumption is not valid. The set of selected transformations may have common inputs or one transformed circuit may be in the fanout of another. *Due to the interaction between transformations s^* may not be achievable.* Earlier work suggests ways of overcoming this problem —

1. Evaluate the actual improvement in circuit delay by introducing the optimized sub-circuits into the original circuit. This approach is used in [18] to evaluate transformations along the critical path.

2. Recursively apply optimization to the network after the selected sub-networks are removed [80]. The original motivation for doing so is to improve delays along non-critical paths. In many cases this allows increased optimization along the critical path. We wish to use the recursion to buffer gates whose fanout may have increased as a result of applying the selected transformations.

For mapped circuits, the procedure used to select the selection-set $\mathcal{S}$ is outlined in Figure 4.6. The first step in MAPPED_SELECTION is to evaluate s^*, the slack that can be achieved assuming that there is no interaction between the transformations. This is an upper limit for the achievable slack. The selection function $\mathcal{F}$ is generated for this value of achievable slack. A fixed (user defined) number of selection-sets are produced by generating satisfying assignments of $\mathcal{F}$ in order of increasing area penalty. The selection-sets are evaluated to see if they provide improvement in circuit performance by inserting the optimized sub-circuits into the original circuit. The selection-set with the best delay improvement per unit area increase is accepted. In case there is no selection which results in the desired improvement the acceptable slack is reduced. By reducing the slack a larger selection function is generated leading to greater choices for the transformation set. For every value of the s^* that is evaluated, the transformations that have been rejected for previous choices of s^* are removed from the selection function. This is done to explore new transformation sets and not re-evaluate previously rejected choices. R denotes the characteristic function of the rejected selections. The value of s^* is reduced as long as no acceptable selection-set is obtained or until the improvement becomes negligible ($s^* - s(O_1) <$`TOLERANCE`).

4.3.1 Experiments on optimizing mapped circuits

The use of local transformations to reduce circuit delay is similar to the application of transformations to reduce the circuit depth. However, mapped transformations are better suited to improving performance since they have an accurate notion of circuit delay. This section is devoted to experiments that illustrate the various choices and considerations possible when optimizing mapped circuits.

Alternating iteration vs. combined iteration

One objective of combining fanout-optimization with tree-mapping is to explore if the tighter coupling achieves better results than alternating the optimizations. The experiment of Section 4.1 that applies tree-duplication followed by application of buffering is compared with an iterative application of both techniques during a single pass. The results of this experiment are summarized in Table 4.4. The combined application of buffering and tree-duplication (**dupl+buf**) produces smaller improvements in delay compared to the separate application of tree-duplication followed by fanout-optimization (**dupl;buf**).

Improvement everywhere vs. critical path based

The paradigm of applying mapped transformations to reduce delay can be used to compare different strategies for mapping a circuit. The two strategies are illustrated in Figure 4.7. The **direct** strategy finds an initial implementation with small delay, perhaps with large area, and then tries to recover area while ensuring that the delay does not increase. An example of this approach is performance-oriented technology-mapping [66]. The **iterative** strategy, on the other hand, starts with an area-optimized circuit and then reduces delay by application of local transformations along critical paths at the expense of area. An interesting question is to compare these heuristics, both of which address the problem of optimizing the delay with a small increase in area, from different starting conditions. A similar comparison was made in Section 4.2.1 to compare *delay-based* buffering and *area-based* buffering strategies. The comparison here includes gate-selection as well as fanout-optimization.

Performance-oriented technology-mapping is used as a representative of the *direct* approach. The SIS command `map` with options to reduce delay (`-m1`) and recover area (`-A`) implements the *direct* approach. For a fair comparison, the *iterative* approach is provided only with transformations that are capable of the same optimizations as the *direct* approach. This means that the local transformations are restricted to remapping a single tree and creation of fanout-trees. The SIS command `speedup_alg noalg fanout` specifies the transformations and the *iterative* strategy is applied using the command `speed_up` with options to compute delays based on the cell-library (`-m mapped`) and to restrict the scope to a single (`-d 0`) tree (`-s tree`). Table 4.5 shows the results of this experiment.

The iterative technique consistently produces the smaller circuit while the direct technique frequently produces the smaller delay. This suggests that there is scope for improvement in the iterative improvement procedure. On the average, technology-mapping based on the *direct* approach reduced the delay by 34% with a 26% increase in area. The *iterative* approach reduced the delay by 30% for a 15% area penalty. This indicates that the improvement in delay per unit increase in area is better for the iterative procedure. The iterative technique produces a series of circuits with different area/delay characteristics and the optimization can be terminated when timing constraints are met. The *direct* approach produces only the final optimized circuit as its output.

Some observations on this experiment, that might explain the difference in results, are in order. The direct technique uses estimates of fanout to control the remapping of trees while in the iterative technique buffering and remapping of trees is done independently. This puts the iterative strategy at a disadvantage. Furthermore, a final area-recovery step can be applied even with the iterative strategy. This observation stems from the fact that the critical paths may change during successive iterations, thereby resulting in unwanted optimization on parts of the circuit that are eventually non-critical. Since area-recovery often reduces the delay in addition to reducing the area, it is always beneficial to apply area-recovery as a post processing operation. No post-processing is performed in this experiment. There may be no need to apply area-recovery after the iterative strategy if local transformations that down-size non-critical parts can also be used.

Comparison with LATTIS

LATTIS [18] is an optimization heuristic based on selecting local transformations along the most critical path that result in the most improvement in delay per unit area increase. In addition to fanout-optimization, re-mapping and duplication of gates, a transformation that reduces the load contributed by non-critical fanouts is also suggested. This transformation is not implemented in the current set of transformations that we use. In addition, the PEEPHOLE transformation of [18] differs from the restructuring transformations described. PEEPHOLE evaluates a number of scopes rooted at a node until the size of the scope exceeds a threshold, and then picks the best one. Our approach to restructuring uses only a fixed scope whose size is determined by the user. With these differences in mind, we compare the heuristic of LATTIS with that of evaluating different selection sets. The experimental settings are the same as that suggested in [18]. The delay data provided for the benchmark examples in [18] is used to derive the maximum speedup and the corresponding area increase. The optimization strategy that we use is to first apply re-mapping and fanout-optimization on single trees (like in the *iterative* strategy of the previous experiment). This is followed by more powerful restructuring that only considers a scope of depth 2 along the relevant paths. The restructuring uses the transformations `noalg`, `divisor`, `fanout` and `duplicate` and the SIS command `speed_up -m mapped -d 2`. A cpu-limit of 4 hours was specified for the iterative procedure. The examples for which the time limit expired are marked with an asterisk. Table 4.6 shows the maximum speedup obtained and the resulting area increase for the two heuristic techniques.

From the experiment, it appears that the LATTIS heuristic is superior in reducing delay. However, it should be mentioned that the PEEPHOLE strategy of LATTIS is more powerful than the restructuring employed by the iterative improvement algorithm. In addition down-sizing of the non-critical fanouts is not used in our experiment.

Mapped vs. unmapped optimization

An important reason for considering mapped transformations was that optimizations at the technology-independent level do not always lead to a smaller delay. This section explores the quality of optimizations when accurate delay data is available. The experimental setup is to use the area-optimized circuits, mapped for minimum area and delay, as the initial circuit. In the case of mapped circuits we do not need to perform the initial 2-input gate decomposition. For the circuits mapped for minimum-area, a first pass is made using only `fanout` and `noalg` transformations restricted to individual trees to generate a starting point for the restructuring techniques. The scope of transformations is limited to a depth of 2 mapped gates. Transformations used for this experiment are `noalg`, `divisor`, and `fanout` and a limit of 4 hours is set for the optimization. The result of applying mapped transformations is compared with the result of first restructuring the logic at the technology-independent stage and then mapping it for minimum delay.

Table 4.7 shows the results of this experiment and the results are somewhat mixed.

The restructuring techniques initiated from the minimum-delay mapping perform better than the two-pass iterative improvement starting from the area-mapped circuit. However, working on the mapped circuits does not significantly improve the level of optimization. In fact, for a substantial number of examples the delay improvement is poorer when local transformations are applied on mapped circuits. The possible reasons for this anomaly are elaborated on in the next section.

4.4 Conclusions

The techniques presented in this chapter show various optimizations that can be applied on mapped circuits. The advantages of working with mapped circuits is that delay data is accurate. This overcomes the unpredictability of optimizations at the technology-independent stage.

The delay of mapped circuits may be improved by combining trees to allow the tree-mapping algorithm greater flexibility in reducing the delay. The choice of fanout-trees is also important and we have proposed an algorithm for the creation of fanout-trees that produces solutions that are close to the minimum-area and minimum-delay solutions. The restructuring and buffering techniques have been combined into a unified framework that can exploit the characteristics of the gates in the cell-library to reduce the delay of logic circuits.

Unfortunately, the results of applying local transformations on mapped circuits are not very encouraging. The possible explanations for this are:

1. The techniques to determine the scope of the transformations are poor and this restricts the optimizations.

2. The current implementation is very memory intensive and results in large run-times for evaluating the mapped transformations on large circuits.

3. The set of local transformations do not include techniques to down-size the non-critical path nodes. This is crucial in getting out of local minima during the iteration.

4. A drawback of the procedure to select local transformations is that it terminates whenever there is no improvement possible. Hill-climbing techniques that perturb the circuit so that further optimization can be carried out need to be investigated. In the

absence of a strategy that can overcome local minima, the optimization procedure is highly sensitive to the initial starting circuit.

These issues have to be addressed to make the techniques viable.

The current set of transformations operate either on gates or on the fanout points. It is possible that by combining both types of techniques, e.g. generating different implementations for the function and its complement and using the two implementations to selectively drive the fanouts, there may be possibility of reducing delay. The current procedure is unable to handle such transformations, in which the scope has multiple inputs and multiple outputs, since it is not possible to determine the local improvement at a single node. Such general scopes are considered, for a simple delay model which ignores fanout considerations, in the work described in [35].

Having addressed the combinational optimization problem in this and the previous chapters it is time to turn to a more general class of circuits. This leads us to studying the optimization of synchronous sequential circuits which is the subject of the next chapter.

file	cpr		clustering		Ratio	
	Delay	Area	Delay	Area	Delay	Area
C1355	1318	17.82	1611	16.76	1.22	0.94
C1908	1354	24.40	1523	25.71	1.12	1.05
C2670	1890	16.11	TO	TO		
C3540	3213	32.13	3211	32.53	1.00	1.01
C432	797	18.41	662	22.61	0.83	1.23
C7552	5922	22.90	6103	24.76	1.03	1.08
C6288	8114	74.87	8211	75.43	1.01	1.01
ampbpreg	2612	16.18	2213	12.20	0.85	0.75
ampbsm	1843	11.77	1831	11.24	0.99	0.95
amppint2	1441	13.76	1370	13.09	0.95	0.95
ampxhdl	844	12.70	847	11.51	1.00	0.91
b12	221	5.99	194	5.24	0.88	0.87
b9	304	8.12	328	6.34	1.08	0.78
cordic	222	7.71	213	9.26	0.96	1.20
cps	2924	14.96	3190	13.34	1.09	0.89
dalu	2319	15.69	2376	15.13	1.02	0.96
des	8392	16.71	8409	18.10	1.00	1.08
dflgrcb1	753	8.64	786	8.74	1.04	1.01
duke2	1104	13.18	1024	12.39	0.93	0.94
ex1010	5422	15.87	TO	TO		
ex4	1224	8.61	1229	10.62	1.00	1.23
fconrcb1	590	9.00	586	10.14	0.99	1.13
k2	2975	17.07	3607	14.38	1.21	0.84
kcctlcb3	612	8.05	708	7.41	1.16	0.92
misex2	322	7.02	297	6.12	0.92	0.87
misex3c	1500	19.98	1163	18.81	0.78	0.94
pdc	1050	10.77	879	8.41	0.84	0.78
rd84	330	10.76	412	11.35	1.25	1.05
rot	1787	14.76	1739	16.55	0.97	1.12
sbiucb1	645	12.34	646	11.41	1.00	0.92
spla	1671	13.07	1655	11.19	0.99	0.86
t481	1753	15.68	TO	TO		
tfaultcb1	468	7.50	500	6.17	1.07	0.82
G. MEAN					1.00	0.96

cpr	Critical-path-restructuring using 2-cube divisors + `map -m1 -A`
clustering	Clustering performed using `script.delay`
Ratio	using the **cpr** data as the base case
Area	area of the circuit (MCNC `lib2` data divided by common divisor 464)
Delay	delay of the circuit (MCNC `lib2` data in nanoseconds)

Table 4.3: Comparison of mapped delays of optimized circuits

```
MAPPED_SELECTION($\eta$) {
    $\mathcal{S} = \emptyset$ /* records the best selection */
    $R = 0$ /* characteristic function of rejected selections */
    found_good_set = FALSE
    $s^*$ = compute_achievable_slack($\eta$)
    do {
        $\mathcal{F}$ = build_selection_function($\eta$, $s^*$)
        $\mathcal{F} = \mathcal{F} - R$;
        for ($i$ = 0; i < NUM_SELECTIONS; $i$++) {
            $a$ = min-weight-satisfying-assignment($\mathcal{F}$);
            if (selection_is_acceptable($a$)) {
                found_good_set = TRUE;
                $\mathcal{S}$ = best_selection_so_far($\mathcal{S}$,$a$);
            } else if (selection_degrades_performance($a$)) {
                $R = R \cdot a$
            }
            $\mathcal{F} = \mathcal{F} - a$
        }
        if (NOT found_good_set) $s^* = (s^* + s(O_1))/2$
    } while (NOT found_good_set AND $(s^* - s(O_1)) >$ TOLERANCE)
    return $\mathcal{S}$;
}
```

Figure 4.6: Selection procedure for mapped circuits

operation	1 level		2 level	
	Area	**Delay**	**Area**	**Delay**
delay-map	1.0	1.0	1.0	1.0
dupl	1.03	0.95	1.04	0.95
dupl;buf	1.03	0.94	1.05	0.93
dupl+buf	1.03	0.95	1.07	0.95

delay-map	Circuit mapped for minimum delay (`map -m 1 -A`)
dupl	Duplicating trees of specified depth along the critical paths
dupl;buf	Duplicating trees followed by buffering
dupl+buf	Using duplication of trees and buffering as local transformations
Area	Average normalized area (normalized to min-delay mapping)
Delay	Average normalized delay (normalized to min-delay mapping)

Table 4.4: Combining tree-duplications and fanout correction

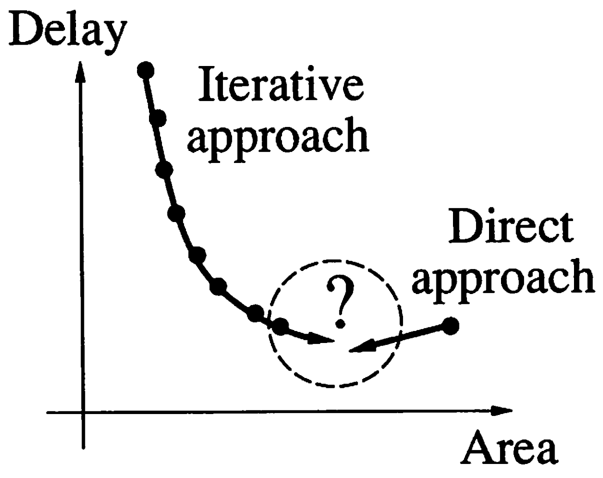


Figure 4.7: Two approaches to delay optimization

file	**area-map**		**Iterative**		**Direct**	
	Area	Delay	Area	Delay	Area	Delay
C1355	914	25.42	1.22	0.89	1.29	0.80
C1908	960	39.19	1.12	0.82	1.18	0.83
C2670	1334	31.52	1.06	0.74	1.22	0.71
C3540	2273	51.44	1.07	0.77	1.23	0.73
C432	381	39.04	1.24	0.75	1.25	0.71
C7552	4087	74.44	1.05	0.46	1.19	0.44
ampbpreg	1503	33.12	1.06	0.60	1.21	0.61
ampbsm	1257	29.99	1.06	0.64	1.26	0.56
amppint2	965	32.44	1.11	0.68	1.17	0.67
ampxhdl	595	23.33	1.16	0.60	1.23	0.59
b12	151	8.11	1.07	0.83	1.19	0.71
b9	248	8.90	1.12	0.90	1.16	0.94
cordic	128	12.04	1.06	0.96	1.43	0.81
cps	2123	36.17	1.10	0.52	1.22	0.52
dalu	1612	48.27	1.09	0.47	1.25	0.46
des	6224	126.04	1.15	0.18	1.20	0.16
dflgrcbl	587	12.93	1.03	0.93	1.19	0.82
duke2	751	21.87	1.16	0.80	1.21	0.80
ex1010	2935	31.82	1.52	0.66	1.73	0.54
ex4	861	13.34	1.17	0.86	1.30	0.76
fconrcbl	445	14.76	1.08	0.78	1.21	0.88
k2	2184	30.94	1.09	0.67	1.23	0.68
kcctlcb3	452	12.43	1.07	0.86	1.17	0.80
misex2	201	10.48	1.18	0.85	1.35	0.73
misex3c	799	48.79	1.31	0.66	1.33	0.60
pdc	682	18.04	1.12	0.81	1.27	0.67
rd84	247	14.32	1.17	0.87	1.19	0.82
rot	1299	28.52	1.06	0.75	1.19	0.65
sbiucbl	453	22.71	1.21	0.75	1.21	0.69
spla	1129	23.62	1.13	0.74	1.30	0.72
t481	906	29.42	1.95	0.62	1.87	0.62
tfaultcbl	365	8.89	1.08	0.92	1.21	0.75
G.MEAN			1.15	0.70	1.26	0.66

area-map	Area optimized circuit mapped for minimum area (`map -m0`)
Iterative	Area optimized circuit improved by local transformations
Direct	Area optimized circuit mapped for minimum delay
Area	area of the circuit (MCNC `lib2` data divided by common divisor 464)
Delay	delay of the circuit (MCNC `lib2` data in nanoseconds)

Table 4.5: Comparison of direct and iterative optimization strategies

Example	LATTIS		proposed	
	greatest speedup	**area increase**	**greatest speedup**	**area increase**
C1355	0.70	2.36	0.65	1.32
C1908	0.53	1.88	0.65	1.15
C2670*	0.45	1.16	0.54	1.19
C3540	0.42	1.36	0.60	1.08
C432	0.38	2.09	0.48	1.66
C6288*	0.52	1.89	0.68	1.06
C7552*	0.23	1.22	0.47	1.12
cordic	0.81	1.23	0.82	1.21
dalu	0.31	1.16	0.61	1.20
des*	0.12	1.17	0.36	1.01
k2*	0.20	1.30	0.73	1.06
rot	0.46	1.23	0.48	1.27
t481	0.76	1.51	0.98	1.01

Initial	Area optimized circuit mapped for minimum area (`map -m0`)
LATTIS	Circuit optimized using the LATTIS heuristic
proposed	Circuit optimized using local transformations
greatest speedup	Ratio of final delay to initial delay
area increase	Ratio of finial area to initial area

For the examples marked with a * the CPU limit of 4 hours was exceeded

Table 4.6: Comparison with the LATTIS heuristic

Example	unmapped		mapped-area		mapped-delay	
	Area	**Delay**	**Area**	**Delay**	**Area**	**Delay**
ampbpreg	2612	16.18	1723	17.75	2435	15.04
ampbsm	1843	11.77	1534	14.58	2075	14.67
amppint2	1441	13.76	1349	17.35	1706	15.48
ampxhdl	844	12.70	800	12.18	1099	10.95
b12	221	5.99	226	5.64	240	5.35
b9	304	8.12	356	6.54	390	6.65
cordic	222	7.71	250	6.93	253	6.75
cps	2924	14.96	TO	TO	3589	13.58
dalu	2319	15.69	1939	20.97	2715	17.68
des	8392	16.71	6335	49.46	TO	TO
dflgrcb1	753	8.64	711	8.11	910	8.60
duke2	1104	13.18	1155	13.83	1493	13.37
ex4	1224	8.61	1144	8.83	1569	8.57
fconrcb1	590	9.00	673	8.41	802	9.08
k2	2975	17.07	2318	20.60	3382	18.29
kcctlcb3	612	8.05	657	7.51	698	7.26
misex2	322	7.02	262	7.04	369	6.64
misex3c	1500	19.98	1726	26.28	1818	23.63
pdc	1050	10.77	872	12.83	1259	10.63
rd84	330	10.76	298	11.70	440	10.41
rot	1787	14.76	1808	13.95	2439	15.05
C1355	1318	17.82	1298	20.70	1378	18.98
C1908	1354	24.40	1498	26.01	1587	26.92
C3540	3213	32.13	2725	35.19	4006	34.86
sbiucb1	645	12.34	739	15.12	865	13.52
spla	1671	13.07	1560	14.43	2034	12.68
tfaultcb1	468	7.50	518	6.03	507	6.45
C432	797	18.41	414	31.25	1093	20.37
C7552	5922	22.90	4723	43.82	6262	29.88
t481	1753	15.68	1923	16.78	2173	16.87
C2670	1884	16.11	1795	18.90	2219	18.57
ex1010	5422	15.87	4890	19.09	TO	TO

unmapped	Unmapped optimizations followed by `map -m1 -A`
mapped-area	Optimizations performed on the minimum-area mapped circuit
mapped-delay	Optimizations performed on the minimum-delay mapped circuit
Area	Area of the circuit (MCNC `lib2` data divided by common divisor 464)
Delay	Delay of the circuit (MCNC `lib2` data in nanoseconds)

Table 4.7: Comparison of mapped and unmapped optimizations

Chapter 5

Synchronous circuit optimization

The focus of previous chapters has been on synthesizing a combinational circuit subject to timing constraints. This chapter deals with techniques to design synchronous circuits that meet the user-specified clocking scheme.

There are two well-known approaches for optimizing synchronous circuits. The first one is to partition the circuit into pieces of logic bounded by latches, develop timing constraints for the combinational modules and optimize them using combinational techniques. The key problem here is to generate the constraints for combinational blocks and will be discussed in Section 5.1. The second approach derives from the flexibility in repositioning registers. **Retiming** [33] is a method to determine the locations for the registers in a single-phase, edge-triggered circuit that minimize the clock cycle. For circuits with multi-phase clocking and level-sensitive latches, heuristic positioning of latches has been attempted [4]. Latches are moved in response to the slack at a node. Our approach is also based on using the output of timing analysis to guide the optimization. In response to the slacks in the circuit, logic can either be resynthesized or moved into a different time period. Both combinational resynthesis and latch movement are regarded as local transformations. The transformation that leads to performance improvement at smaller cost is selected.

Combinational resynthesis methods and retiming techniques address the optimization from separate and non-interacting perspectives. Combinational resynthesis focuses on improving the structure of the logic without changing the positions of the latches. On the other hand, retiming moves the registers while retaining the structure of the combinational logic. The merger of the two techniques is called retiming-and-resynthesis (RandR) [41]. RandR is also applicable to the performance optimization of pipelined circuits and this is discussed in Section 5.2.

Circuits with single-phase clocking may be viewed as finite-state machines. The *state* of the machine is represented by the data stored in the registers. The state is updated in response to the inputs when a clock pulse is applied. Section 5.3 addresses the optimization of such structures. The machines can be unrolled to provide greater flexibility in their optimization. For finite-state machines that are explicitly specified as a collection of transitions between states, the flexibility in assigning binary values to symbolic states (state assignment) can be exploited to reduce circuit delay as well.

Finally, in Section 5.4, the need for a different type of combinational optimization procedure is introduced. The problem of resynthesizing a circuit to meet timing constraints can be viewed as a mathematical programming problem that provides constraints on the delays along paths of the circuits. This results in the need for an optimization system that can ensure that point-to-point delays are met and that specific paths meet certain delay requirements. Currently, such an optimization system is not available to us. The general problem of ensuring point-to-point delays can be transformed into the combinational optimization problem based on terminal constraints (arrival and required time constraints) that was studied in Chapters 3 and 4. Techniques for making such a transformation are discussed.

5.1 Timing analysis and optimization

A **synchronous circuit** consists of combinational logic and memory elements. Every memory elements must be either "edge-triggered" or "level-sensitive". Each memory element has a data input, a clock input and a single output. An edge-triggered memory element, **flip-flop**, samples and stores the input data at the appropriate edge of the clock signal and presents the stored value at the output. This output remains stable until the next occurrence of the active edge. From a timing perspective, this results in decoupling the logic on either side of a flip-flop. For a level-sensitive memory element, **latch**, the data at the input is transmitted to the output during the active period of the clock. The output is held at the data value from the time the active period ends until the start of the next active period. Thus the input and output are not isolated during the active period. Time may thus be "borrowed" across a latch. For correct operation of both types of memory elements, the data signal must be stable at the input of the memory element before the latching edge occurs by an amount called the **setup time,** S. It is also required that the signal be stable after the latching edge by an amount called the **hold time,** H. We assume, without loss of generality, that level-sensitive latches are active when the clock signal is high and that the flip-flops are triggered by a falling signal at the clock input. Thus, the falling edge of each phase is the critical edge with respect to which setup and hold constraints must be satisfied.

A clocking scheme, Φ, is a collection of l periodic signals with a common period c and is represented by $\Phi = (\phi_1, \phi_2, \cdots, \phi_l)$. c is called the **clock period** or the **cycle time.** Associated with each phase ϕ_i are two real numbers r_i and e_i, the time of occurrence respectively of the rising and falling edges of ϕ_i $(0 \leq (r_i, e_i) \leq c)$. Associated with each phase i is its local time zone, an interval of time of length c, such that the end of the active phase coincides with the end of the local time zone. Let $0 \leq e_1 \leq e_2 \cdots \leq e_l = c$; thus we choose the global reference time frame as the last phase e_l. The clocking scheme specifies a complete ordering of the rise and fall of the phases.

For any path of purely combinational elements from a latch u clocked on phase $\phi(u)$ to a latch v clocked on phase $\phi(v)$ define

$$K_{uv} = \begin{cases} 0 & \text{if } e_{\phi(u)} < e_{\phi(v)} \\ 1 & \text{otherwise.} \end{cases}$$

K_{uv} is simply a token to keep track of the fact that a cycle boundary has been crossed. For a path $p : u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k$, where u_i is a latch on the path and each sub-path $u_i \rightarrow u_{i+1}$ contains no latches on it, the number of clock cycles available for computation along it is $\sum_{i=1}^{n-1}(K_{u_i u_{i+1}})$.

The **synchronous circuit optimization problem** is —

> *Given a circuit using a clocking scheme Φ with l phases, find an implementation that meets a given cycle time constraint c and has an equivalent sequential behavior.*

The problem implicitly expects that the rise and fall times for the phases are consistent with the clocking scheme (and possibly other clocking constraints). For a discussion of what constitutes *equivalent sequential behavior* the reader is referred to [60]. In a system design it is usually difficult to tailor individual phases of clocks. More often than not the duty-cycle (the ratio of the active time to the clock cycle) of the phases is fixed. Under such constraints only the cycle time may be changed. This may be viewed as a scaling of the clocking scheme. Depending on the design methodology, the optimization scenario may allow for only clock scaling or may be more flexible and allow changing the individual phases of the clocks as well.

We are interested in necessary and sufficient conditions for correct clocking for an arbitrary multi-phase circuit. The topological structure of a circuit and the distribution of delays within it give rise to a set of constraints called the *internal clocking constraints.* The internal clocking constraints may be classified into two categories — long path constraints and short path constraints. For a path $u \rightarrow v$ that starts at a latch u, clocked on phase $\phi(u)$, and ends at a latch v, clocked on phase $\phi(v)$, the constraints are —

long-path constraints Let $p : u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k$ be a path with latches $u_1, \cdots u_k$ on it, with each sub-path $u_i \rightarrow u_{i+1}$ containing no latches and the path p containing no repeated latches then we require $e_{\phi(u_k)} - r_{\phi(u_1)} \geq \sum_{i=1}^{n-1}(D_{u_i u_{i+1}} - K_{u_i u_{i+1}} c) + S$. These are also known as the set-up constraints.

short-path constraints For every path $p : u \rightarrow v$, with only combinational elements on p, we require $e_{\phi(v)} - r_{\phi(u)} \leq (d_{uv}) + (1 - K_{uv})c - H$. These are also called the hold constraints.

A more detailed discussion on these constraints can be found in [64] and [59].

Techniques used to solve these constraints fall into two broad classes — mathematical programming based approaches [52, 64] and relaxation based, or iterative, procedures such as those in [76, 10] and [65]. The mathematical programming approaches are ideally suited to find optimum clocking schemes, i.e. given the delays in the circuit, what is the smallest clock cycle at which the circuit will operate. Although the iterative techniques can also address optimum clocking (by repeatedly checking different values of clock cycle), they are more suited to identifying parts of the circuit that fail to meet timing constraints. We use the timing analyzer `hummingbird` [76] to identify portions of the circuit that violate timing constraints.

In designs with level-sensitive latches, the short-path constraints also need to be satisfied. This leads to a different problem for combinational circuits. The output should arrive later than an early-required time and before a late-required time constraint. The traditional approach has been to ignore the short-path constraints under the assumption that these paths can be slowed down by adding delays. This is certainly true if delays can be tailored continuously as in ECL circuits [78]. When the added delay elements can take on only discrete values, the problem of meeting long and short paths simultaneously is not as simple. We follow the traditional approach and concentrate on ensuring that long paths meet desired constraints and ignore the short path constraints.

5.1.1 Optimization based on timing analysis

Timing analysis is used to determine which parts of a circuit fail to meet the timing constraints. These parts then need to be resynthesized. The optimization of sequential circuits by resynthesis of combinational modules is described in the procedure ANALYSIS_REDESIGN_LOOP of Figure 5.1.

```
ANALYSIS_REDESIGN_LOOP {
    Generate initial area-optimized implementation
    do {
        Use timing-analysis to identify slow paths
        Generate performance constraints for modules with slow paths
        Select one module and resynthesize
    while (timing constraints are violated)
}
```

Figure 5.1: Analysis-redesign loop for circuit optimization

Timing constraints are easy to derive for circuits that contain only flip-flops and logic due to the isolation between the logic on either side of a flip-flop. Furthermore, since there is no possibility of "borrowing" time across flip-flops, the timing constraints on a combinational sub-circuit are precisely the constraints that are needed to ensure that the setup and hold-time constraints are satisfied for the flip-flops that surround the combinational logic.

For circuits containing latches, the generation of constraints for the logic modules poses an interesting problem. The slow paths may span several combinational modules and one or more modules may have to be improved to meet the timing constraint along the path. The problem is to partition the slack along the path into the desired improvement for each module. This problem is similar to the problem of distributing slacks across the connections of a netlist so that the layout of the circuit meets the delay constraint for which several algorithms have been proposed [47, 20]. The equivalence of the two problems is as follows — latches in the circuit are in correspondence with the nodes in the netlist and logic modules correspond to the connections between the nodes. The Zero Slack Algorithm (ZSA) proposed in [47] starts with computing the initial slacks in the circuit. It then identifies the path segment with minimum non-zero slack. The excess delay (amount of violation) is distributed uniformly among the modules on the path segment (the target delay of the module along the path is reduced). After this the slacks are updated and the process repeated till every connection has zero slack. At this point if the target delays for the combinational modules are met by resynthesis then the circuit will operate at the desired clock cycle. Several variations of the ZSA exist that differ in the way that the excess slack is allocated among the modules. A popular heuristic is to weigh the allocation proportional to the existing delay. The Limit Bumping Algorithm (LBA) of [20] provides a general framework for allocating the excess slack and provides the conditions to test if a particular schedule of allocating the excess slack will converge or not.

Another method that can be used to reduce timing violations is to move some logic from the critical sections to the non-critical sections. This may be viewed as retiming some nodes in the circuit. The essential difference between the critical path based movement of latches and retiming is that the former is applicable to multi-phase designs as well as circuits containing level-sensitive latches. Even for single-phase edge triggered designs the retiming algorithm suffers from the limitation that the delay of each node is assumed to be constant throughout the retiming process. In a circuit, this is not the case. As the position of a latch changes the load that a gate drives may change leading to a change in the gate delay. The formulation for retiming can be extended to handle accurate delay modeling. However, doing so destroys the structure of the tableau and the resulting mathematical programming problem cannot be solved efficiently. The heuristic procedure of moving logic across latches [4] in response to the slacks in the circuit can handle multi-phase, level-sensitive systems as well as use accurate delay models.

The clock phases may be tailored as well to meet the target clock cycle. This is also done based on the slacks generated by the timing analysis tool. The output of timing analysis identifies paths that are too slow. These paths are terminated by synchronizing elements — latches or flip-flops. The slack along the critical paths indicates the magnitude of the violation of a timing constraint. The clock phases that drive the synchronizing elements terminating slow paths may be moved apart by an amount equal to the maximum violation. By setting up a constraint graph which shows the desired separation between the phases, we can determine if the clock cycle constraint can be met by simply adjusting the clock phases. After a change is made to the clocking scheme, a timing analysis is carried out to determine if more changes are required. It should be remembered that these changes only ensure that the long-path constraints are satisfied. The short-path constraints have been ignored throughout this discussion and have to be ensured by a post-processing phase that introduces additional delay elements.

To choose from among the different optimizations that work to ensure that a design meets the timing constraints, we have introduced an interactive mechanism for the timing analyzer `hummingbird` and integrated it into the SIS synthesis system. During the interactive mode, the user can guide the ANALYSIS_REDESIGN_LOOP. Some of the operations available during interactive use include:

- Generation of maximal combinational logic clusters and queries regarding the delay through each cluster.
- Identification of slow paths along with queries regarding the slack of a node (the slack is a measure of violations of the timing constraints).
- Scaling of clocks and changing the duration of clock phases to ensure that timing constraints are met.
- Generation of terminal constraints for combinational clusters, resynthesis of the clusters using combinational techniques.
- Incremental re-analysis after any of the above changes.

The movement of logic across latches in response to slacks is not implemented in the interactive `hummingbird` interface. Repositioning the latches changes the number of latches and combinational clusters and this prevents incremental re-analysis.

5.1.2 Evaluation of sequential optimization

In this section we evaluate the different synchronous optimizations on an ASIC chip for speech recognition called `viterbi` [62]. It was designed by the DSP group at University

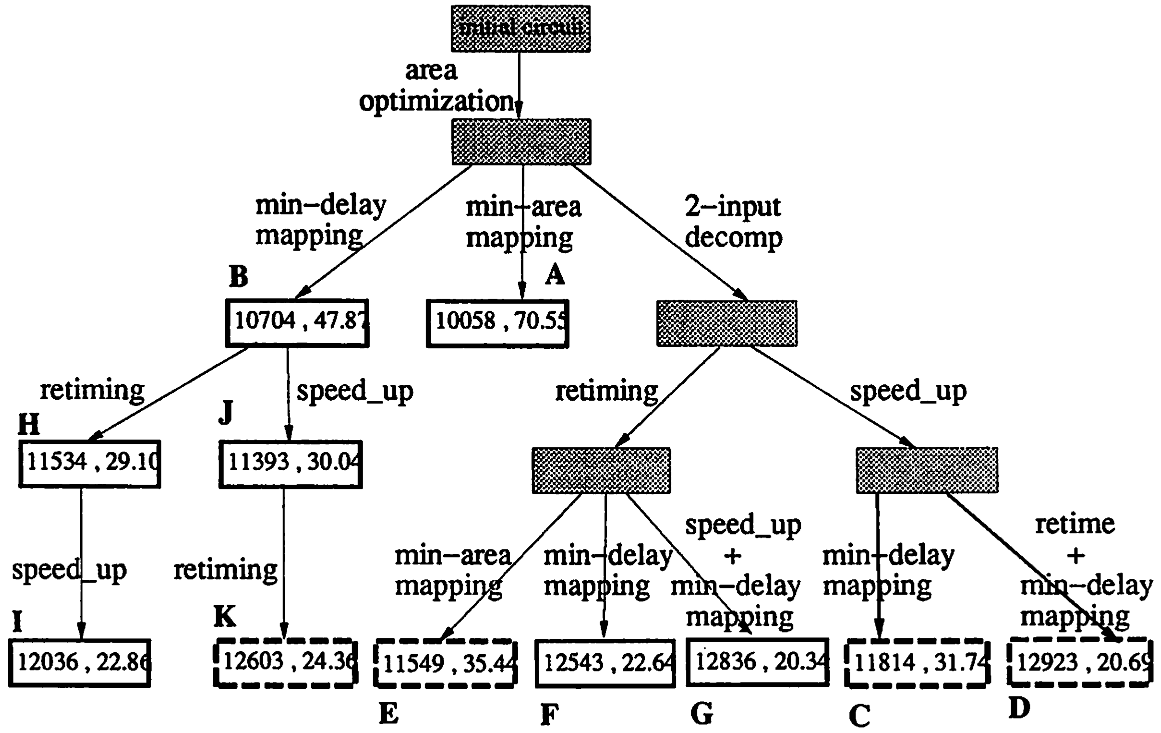


Figure 5.2: Design scenarios for the `viterbi` chip

of California at Berkeley. The design uses single-phase clocking and all memory-elements are edge-triggered. The specification consists of behavioral descriptions of the control and datapaths, and the specification of how these functional blocks are interconnected. The initial description of the design is converted into the extended-BLIF format [79] that handles combinational and memory elements in a hierarchical fashion. Currently the conversion process requires manual intervention.

The initial circuit description is optimized to reduce the area of the design. The optimization is performed using the SIS script `script.rugged`. The circuit is mapped into the cell-library `lib2.genlib` that is part of the MCNC benchmark set [79]. A flip-flop and a latch, both with delay characteristics of the *inv2x* inverter, are added to the cell-library to allow the memory elements to be matched. Figure 5.2 shows a number of ways of reducing the circuit delay. The choices available are based on the type of optimization (combinational optimization or retiming) and the representation on which the optimizations are applied (on a 2-input gate description or on a mapped circuit). Unmapped representations are indicated by shaded boxes while mapped circuits have bold edges. Boxes with dashed edges represent sub-optimal designs. A design is called **sub-optimal** if there is some other design that has a smaller area and a smaller delay. We use the notation $X <_{a,d} Y$ to denote that X is an **inferior design** to Y with respect to area or delay. If $X <_a Y$ then Y has smaller area than X and if $X <_d Y$ then X needs a larger cycle time than Y. A design X is sub-optimal if there exists another design Y such that $X <_a Y$ and $X <_d Y$. We denote the sub-optimality relation by $X < Y$. Every different mapped representation is labeled with a letter. The designs may be viewed as points in a design space whose coordinates are the circuit area and the cycle time. For each labeled design, the design coordinate is shown in Figure 5.2. A graphical presentation of the design space is made in Figure 5.3.

By using different design scenarios, it is possible to get a range of designs that differ in their area-delay characteristics. Designs **A, B, J, H, I, F** and **G** are optimal. It should also be noted that a number of designs are produced when combinational timing optimization called **speed_up** is applied, one after every iteration of applying local transformations. Thus between the pairs (**B, J**) and (**H, I**) there are a number of optimal designs. The initial area-optimized circuit is mapped for minimum-area (**A**) and minimum-delay (**B**). This is the range of circuits that can be obtained if the only optimization used is gate selection and buffering. However, since the design contains arithmetic functions that can be optimized well (e.g. ripple-carry adders) and is highly pipelined (the memory elements may be repositioned) there is reason to believe that both retiming and combinational optimization will produce significant improvements. A 33% reduction in cycle time is obtained after combinational resynthesis (**J**). A similar reduction is obtained if only retiming (**H**) is applied to the optimized circuit. When these techniques are applied in concert further reduction in delay is achieved. The fastest implementation obtained is the design labeled (**G**). It uses an initial retiming to balance the paths between registers, followed by combinational restructuring to decrease the number of levels of logic between latches and finally a minimum-delay technology mapping to generate a fast implementation.

The exploration of the design-space allows us to compare different optimizations and postulate why some design scenarios produce sub-optimal designs.

C < J Both **C** and **J** are obtained by only applying combinational restructuring. The

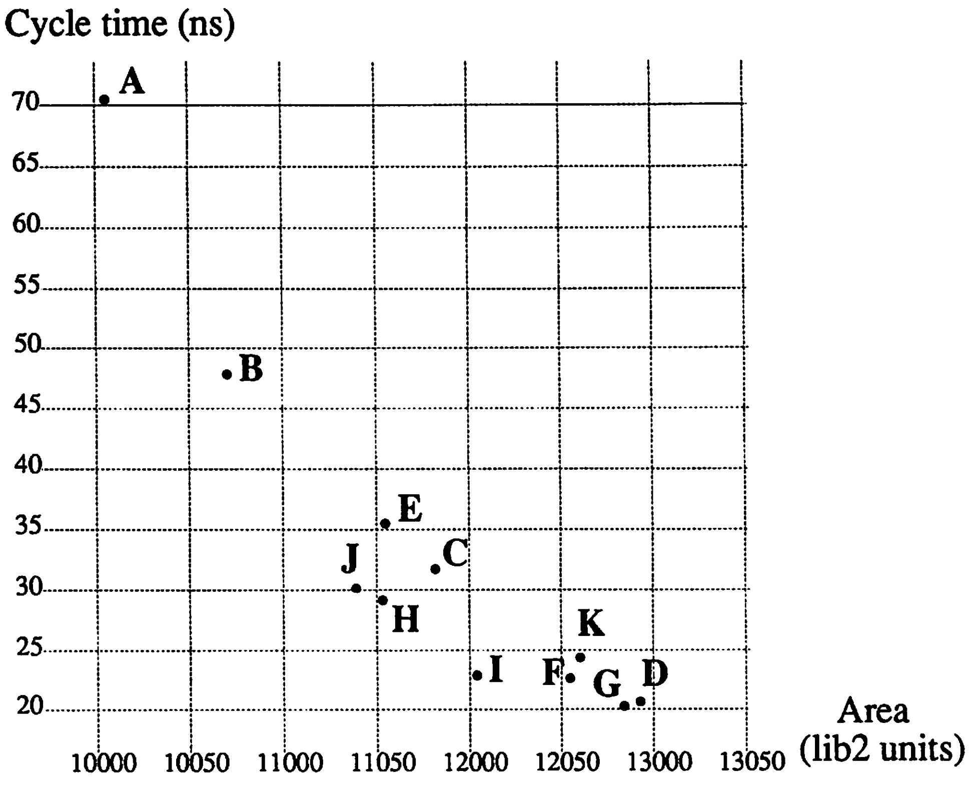


Figure 5.3: Area-delay design space for the `viterbi` chip

optimization to produce **J** works on mapped circuits where accurate delay data is available and produces better results.

D < G, K < I In this case the inferior designs, **D** and **K**, are the result of retiming followed by combinational speed_up while the superior designs, **G** and **I**, are obtained by first applying retiming and then combinational speedup. This result is not true in general. The order of applying retiming and combinational resynthesis impacts the quality of optimization.

H $<_d$ **F** When the logic is in 2-input gate representation there is a small granularity in delay and the position of latches can be controlled very precisely. For mapped circuits the coarse granularity leads to lower improvement. Another contributing factor may be the delay model used during retiming. Retiming assumes a constant delay between any gate input and the output independent of the fanout load. For a gate, the delay to the output may differ from one input to the next. Setting the gate delay to be the maximum pin-to-pin delay increases the granularity in delay.

The resynthesis of synchronous circuits in response to timing analysis may use either combinational restructuring techniques or it may move registers. Both techniques are orthogonal and it is natural to ask if these two techniques can be combined to create a powerful optimization scheme. This is addressed in the next section.

5.2 Retiming and Resynthesis

A technique to combine combinational resynthesis and retiming to address area optimization was proposed in [41]. It is called Retiming and Resynthesis (RandR). The key idea is to push the flip-flops to the periphery of the circuit, even if it involves temporarily *borrowing* latches from the environment. This creates a larger combinational circuit that can be resynthesized using any technique. Following the resynthesis, the borrowed flip-flops are returned to the environment using a retiming algorithm.

Moving the flip-flops to the periphery is done by examining the **path-weight matrix**. For a circuit with m inputs and n outputs the matrix has a size $m \times n$. The entries of the path-weight matrix, W, are

$$w_{ij} = \begin{cases} * & \text{if no path exists between input } i \text{ and output } j \\ \tilde{} & \text{if two paths between } i \text{ and } j \text{ have different number of flip-flops} \\ q & \text{if \textbf{all} paths from } i \text{ to } j \text{ have } q \text{ flip-flops} \end{cases}$$

For a circuit to be peripherally retimed, W should have no $\tilde{}$ entries and there should exist vectors α and β, such that for all $w_{ij} \neq *, w_{ij} = \alpha[i] + \beta[j]$. In a peripherally retimed circuit, there are $\alpha[i]$ flop-flops placed after input i and $\beta[j]$ latches placed before output j. An entry in the α and β vectors may be negative. This is acceptable even though there is no physical implementation of a *negative* flip-flop. The interpretation of a negative register is that it has been temporarily borrowed from the environment. Peripheral retiming exposes all the combinational logic as a single module that can be optimized using combinational techniques. One concern with using RandR is that there is no control on the delay of the circuit after the procedure has been applied. The retiming that follows the resynthesis of the peripherally retimed circuit, inserts flip-flops to make the retiming "legal" i.e. returns all flip-flops that may have been borrowed from the environment thereby ensuring that no edge weight in the retime graph is negative. There is no guarantee that the cycle time will be reduced. In order to provide a guarantee on the cycle time, it stands to reason that the resynthesis of the peripherally retimed circuit be constrained so that the final retiming produces a circuit that meets the desired cycle time.

It is clear that only acyclic circuits can have a peripheral retiming since any cycle (which, for a synchronous circuit, must have a flip-flop on it) will result in a $^-$ entry in W. For acyclic circuits, there is no guarantee that a peripheral retiming exist. However, for pipelined circuit structures like the one in Figure 5.4(a), a peripheral retiming is guaranteed to exist (the peripheral retiming is shown in Figure 5.4(b)). The negative values represent flip-flops that have been borrowed from the environment. For such circuit structures it is possible to generate timing constraints on the combinational logic, C, such that meeting these constraints ensures that the pipeline meets the desired cycle time after retiming.

A pipelined circuit, P, is described as a directed, acyclic circuit composed of a number of combinational circuits $C_1, \ldots, C_l$. Each C_i may have a set of primary inputs I_i and primary outputs O_i. Flip-flops are used to communicate between adjacent stages. Timing constraints on the inputs and outputs are used to represent delays through the logic that generates the inputs or makes use of the circuit outputs. The timing constraints are relative to the active-edge of the flip-flops (all flip-flops are triggered on the same clock event). $a'(i)$ represents the time after the clock event at which the input i is available and $r'(j)$ represents the time before the clock event that output j must be generated by the pipelined circuit. The presence of inputs and outputs at different stages of the pipeline provides greater flexibility than the traditional view of a pipeline in which inputs arrive at the first stage and outputs are generated at the last stage.

The question arises,"**What timing constraints for the combinational circuit C will guarantee that after resynthesis and retiming the circuit will have a clock cycle less than the target clock period c?**". Clearly, if a specific path between an input and an output meets a certain delay then flip-flops can be placed along it at appropriate intervals to equalize the segments between successive flip-flops. The discrete nature of flip-flop positions results in a coarse granularity with which the desired clock period can be met. Thus, if the timing constraint for a path with l flip-flops is cl and the largest gate delay along the path is d_{max}, then after positioning flip-flops at appropriate points along the path only a clock period of $c + d_{max}$ can be guaranteed. Between input I_i^m of stage i

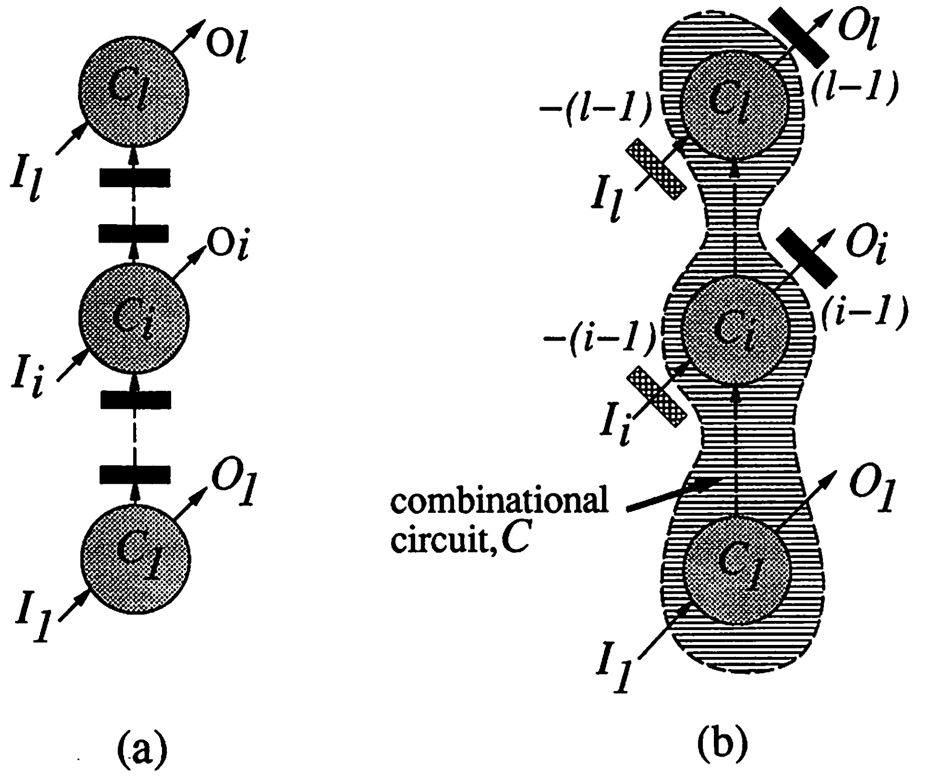


Figure 5.4: Pipelined circuit structure

(with arrival time $a'(I_i^m)$ after the clock edge) and output O_j^n in stage j (with a required time $r'(O_j^m)$ before the clock edge) there are $(j-i)$ flip-flops. The time available for logic computation between this input and output is $c \cdot (j-i-1) + (c - a'(I_i^m)) + (c - r'(O_j^n))$. Meeting this constraint will guarantee that the $(j-i)$ flip-flops can be placed to meet a clock cycle of $c + d_{max}$ along all paths between I_i^m and O_j^n. For every input-output pair a similar constraint can be specified. It seems unlikely that constraints for every input-output pair can be satisfied simultaneously. However, by translating the relative arrival and required times into an absolute frame of reference, timing constraints for the combinational logic C can be generated that ensure that the desired clock cycle for the pipeline can be met to within the gate granularity d_{max}.

We assume that we are given a pipelined circuit P, consisting of l stages $C_i, \ldots, C_l$

and a desired clock cycle c along with arrival times and required times specified relative to the clock edge. A combinational circuit C, is derived by peripheral retiming as in Figure 5.4. The inputs of C are in one-to-one correspondence with the inputs of P. Timing constraint for input I_i^m of C (that corresponds to input I_i^m in stage i in P) is set to be $a(I_i^m) = a'(I_i^m) + (i-1)c$ and similarly the required time for output O_j^n of C (the corresponding output appears in stage j in P) is set to be $r(O_j^n) = r'(O_j^n) + jc$. By doing so, the timing constraints for P which were relative to the clock edge within a cycle have been converted into the same absolute time frame.

Theorem 5.2.1 *[42] If the combinational circuit C satisfies the timing constraints generated above, then the corresponding pipelined circuit P can be resynthesized to have a clock cycle of $c + d_{max}$ where d_{max} is the maximum delay of any gate after resynthesis of C.*

In addition to being able to solve the pipeline optimization problem via combinational resynthesis, it is also obvious that if the pipeline can be synthesized for a clock cycle c, then the corresponding maximal combinational circuit C meets the timing constraints generated above. Since the clock period c is achieved, every segment between successive flip-flops has delay less than c. In addition paths between input and flip-flop as well as flip-flop and output have delays such that the path delay and the terminal constraint is less than c. Using these facts, it is clear that every path between input and output has delay less than the terminal constraints generated for C.

The equivalence between combinational optimization and pipeline optimization problems is a useful result. However, for more general feed-forward topologies, a peripheral retiming is not guaranteed and the synthesis procedure (of peripheral retiming, combinational resynthesis with appropriate constraints and finally flip-flop insertion) is not applicable. Recently, a procedure to remove retiming bottlenecks in general circuits has been proposed [15]. A retiming bottleneck is a circuit structure that prevents retiming from achieving the desired clock cycle. The removal of retiming bottlenecks is based on the resynthesis of the logic under timing constraints that guarantee that if the constraints are met then a subsequent retiming will meet the desired cycle time. However, the conditions to eliminate retiming bottlenecks are only sufficient. It should be noted that for the pipeline optimization problem the timing constraints were both necessary and sufficient.

For specific cyclic structures we are able to use, with some modification, the pipeline optimization procedure. This is the subject of the next section.

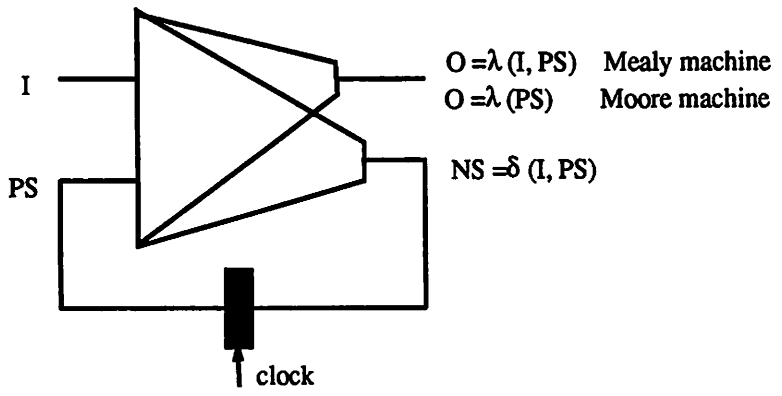


Figure 5.5: Standard finite-state machine structure

5.3 Finite-state machine optimization

A class of sequential circuits that are common in designs are finite-state machines (FSM's). Figure 5.5 shows a FSM structure. It consists of the set of inputs, I, the outputs O and the set of present states PS, along with two combinational functions λ and δ. λ is the output function map, $\lambda : I \times PS \rightarrow O$ for a *Mealy* machine and $\lambda : PS \rightarrow O$ for a *Moore* machine. The function δ computes the next-state (NS) based on the present-state and the input, i.e. $\delta : I \times PS \rightarrow NS$. The next-state during the current clock cycle becomes the present-state during the next cycle for which a new next-state is computed. Thus, based on the state of the machine and the inputs, the machine produces a sequence of outputs based on a sequence of inputs. Any single-phase synchronous circuit may be viewed an FSM. To distinguish between FSM's used as controllers and arbitrary synchronous circuits represented as FSM's, we introduce the notion of standard FSM structure. A **standard FSM** represents a structure in which there is at most one latch on any simple cycle or simple input-output path. A simple cycle or path has no repeated edges along it.

In this section we consider only the standard FSM structure and assume that all inputs to the combinational logic of the FSM are latched. This is certainly true for the *state* inputs. The primary inputs, I, are also considered to be outputs of latches (this poses no loss of generality since any logic preceding the inputs, in the environment of the FSM, may be considered to be part of the FSM). The performance of the FSM is determined by the maximum arrival time from among the primary outputs and the next-state outputs.

5.3.1 Optimization via unrolling

At first glance it seems that the only option of optimizing standard FSM's is via combinational resynthesis of the logic. Retiming the FSM cannot reduce the cycle time since a standard FSM has a single register on every simple cycle. Positioning the register anywhere along the cycle does not reduce the cycle time. Furthermore, since the structure is cyclic, no peripheral retiming exists making the application of RandR techniques impossible. However, on closer examination, we find that the result of Section 5.2 can be applied to the iterative array model of a finite-state machine. To generate the iterative array representation the logic is duplicated, along with the inputs as shown in Figure 5.6(a). In such a structure the the registers can be moved to the boundary as shown in Figure 5.6(b). The movement of registers to the boundary exposes a larger piece of combinational circuit that can be optimized. After timing optimization the circuit is retimed to reduce the cycle time.

To experiment with unrolling of FSM's for performance optimization, the suggested subset of finite-state machines from the 1991 MCNC benchmark set [79] is used. These FSM's are specified in symbolic form and a circuit implementation is obtained by using the state-assignment program JEDI [37]. Table 5.1 shows the results of optimization when the FSM is unrolled for 1 time step. For each example, the original and unrolled circuits are optimized using local transformations at the technology-independent stage (Chapter 3). After resynthesis, retiming is applied to position registers to minimize the cycle time. The depth refers to the maximum number of levels in the circuit after retiming.

As expected, the area increase as a result of unrolling and resynthesis is high. There is also a considerable increase the number of registers (since the inputs to the following stage are latched). Therefore, in order to make this method practical, the area overhead needs to be reduced. We observe that only the next state functions that have an arrival time greater than that of the longest input-output path, need to be unrolled. If the delay is determined by a path from input to a true output, unrolling will be of no use since such a path will dominate the delay even after resynthesis. Another approach to reducing the delay of unrolled circuits has been proposed by Malik *et.al.* [3]. In that work, optimization is performed by deleting connections along false paths in the unrolled circuit. Due to the very local nature of optimization, the optimized circuit can be re-folded to reduce the

Example	original			unroll		
	latches	lits	depth	latches	lits	depth
cse	4	353	19	42	831	16
donfile	5	163	11	41	471	9
dk16	5	460	22	51	1132	20
dk512	4	117	7	14	221	7
ex1	5	440	16	60	938	19
keyb	5	363	18	35	800	18
styr	5	874	23	44	1799	22
s1	5	336	14	14	686	13
s1a	5	357	15	54	758	14
tbk	5	447	17	56	1068	16
tma	5	350	15	41	715	15

original	Iterative improvement applied to the combinational portion of the FSM
unroll	Single stage unrolling, resynthesis followed by retiming
registers	Number of registers
lits	Number of literals (in 2-input gate form) in the logic
depth	Depth of the combinational logic in 2-input gate form

Table 5.1: Optimization via unrolling of FSM's

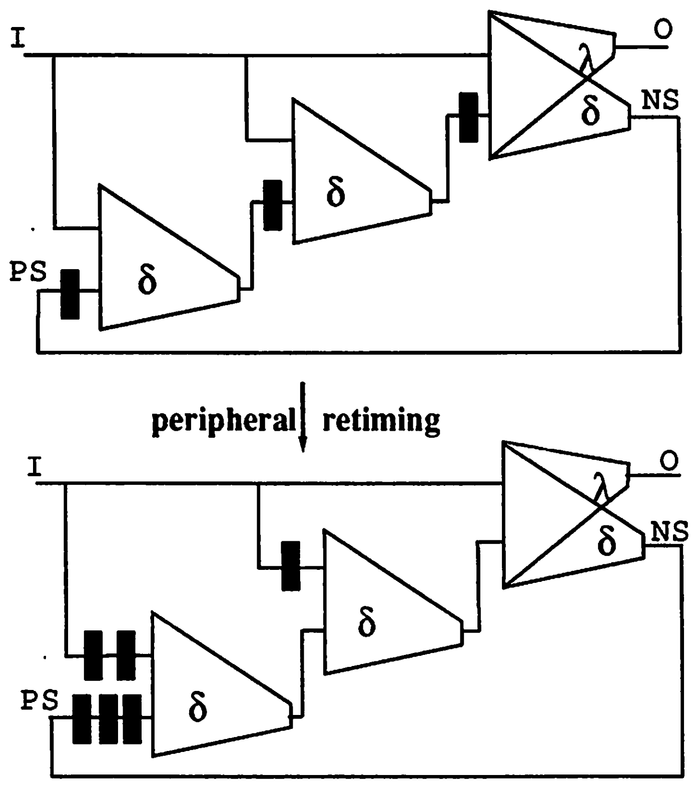


Figure 5.6: The iterative array structure

area overhead. We are investigating the effectiveness of re-folding when a more general restructuring technique is used.

The optimization of FSM's to exploit multi-cycle optimizations (be it via the elimination of multi-cycle false paths or simply by restructuring the logic) provides a degree of optimization that goes beyond the Retiming and Resynthesis approach. Sequential circuit optimizations may be viewed as generating different encodings of equivalent symbolic machines. Even though this view provides very little intuition as to why some techniques work better than others, the process of assigning binary codes to the symbolic states is an important part of synthesis. The next section describes some methods of state assignment and their impact on circuit size and delay.

5.3.2 Performance directed state assignment

For finite-state machines described in symbolic form, the encoding of the symbolic states with binary values provides an additional degree of flexibility. This process is called state-assignment and several techniques have been proposed to produce implementations that have small area — [72] for two-level implementations and [37, 55] for multi-level representations. However, there has been no state-assignment method that addresses the delay of the encoded circuit.

In an attempt to address this question, two extreme state-assignment methods are compared. At the one end are **minimum-width** encodings that use a small number of bits to encode the symbolic states. The other extreme is the **1-hot** encoding strategy where the length of the binary encoding is equal to the number of symbolic states. In the case of manually described FSM's used in controllers most states have relatively few predecessor and successor states. For such machines 1-hot encoding results in relatively simple logic equations describing the transition from one state to another. The drawback of 1-hot encoding is that the number of registers required to store the state of the machine may be large.

For both types of state-assignments, using JEDI for minimum-width encoding and using 1-hot encoding, further optimization is performed by exploiting the don't cares that correspond to the set of unreachable states. These don't care conditions are exploited using the SIS script `script.rugged`. The area-optimized circuits are then subjected to performance improvement via local transformations to reduce their depth. TDD_2cube is used as a local transformation with the "compromise" strategy and a depth of 3 for the scope (refer to Chapter 3 for details). Table 5.2 shows the outcome of this experiment.

The experiment shows that 1-hot encoding results in smaller delay. Intuitively, this is to be expected since the 1-hot encoding adds the least amount of logic to the minimum required to compute the next-state function. Consider any state t. For t to be generated as the output of the next-state logic, the inputs and present-state should correspond to one of the edges that lead to state t. Thus

$$\chi(t) = \sum_{s \in pred(t)} I(s,t) \cdot \chi(s)$$

where $I(s,t)$ is the input condition under which a transition occurs from state s to state t and $\chi(a)$ is a function that evaluates to 1 for state a. For a 1-hot encoding, each state

Example	JEDI				1-hot			
	registers	lits	area	depth	registers	lits	area	depth
cse	4	353	385	19	16	379	507	7
donfile	5	163	203	11	24	370	562	5
dk16	5	460	500	22	27	428	644	9
dk512	4	117	149	7	14	113	225	6
ex1	5	440	480	16	20	343	503	12
keyb	5	363	403	18	19	445	597	11
styr	5	874	914	23	30	704	944	12
s1	5	336	376	14	20	592	752	11
s1a	5	357	397	15	20	375	535	9
tbk	5	447	487	17	32	1307	1563	12
tma	5	350	390	15	20	245	405	9

JEDI , **1-hot**	Method used to assign binary codes to symbolic states
registers	Number of registers used
lits	Number of literals (in 2-input gate form) in the combinational logic
area	equal to **lits** + 8 × **registers**
depth	Depth of the combinational logic in 2-input gate form

Table 5.2: Effect of state-assignment on delay

is represented by a single variable and so the complexity of the next-state function is only slightly more than the minimum required (the sum of the input conditions for the transition to state t).

The size of the optimized combinational logic may be higher for the 1-hot encoding since there are more outputs to be generated. The number of registers is certainly larger for the 1-hot encoding as compared to minimum-width encodings. This results in the area of 1-hot implementations being high. As a means of reducing the area overhead we would like to reduce the number of registers while ensuring that the logic cost and the circuit delay do not increase significantly.

An extension of the 1-hot encoding methodology is k-hot encoding wherein each state code has exactly k bits that are set. A simple computation helps us determine what value of k to use. Figure 5.7 shows how an implementation of a machine with k-hot encoding

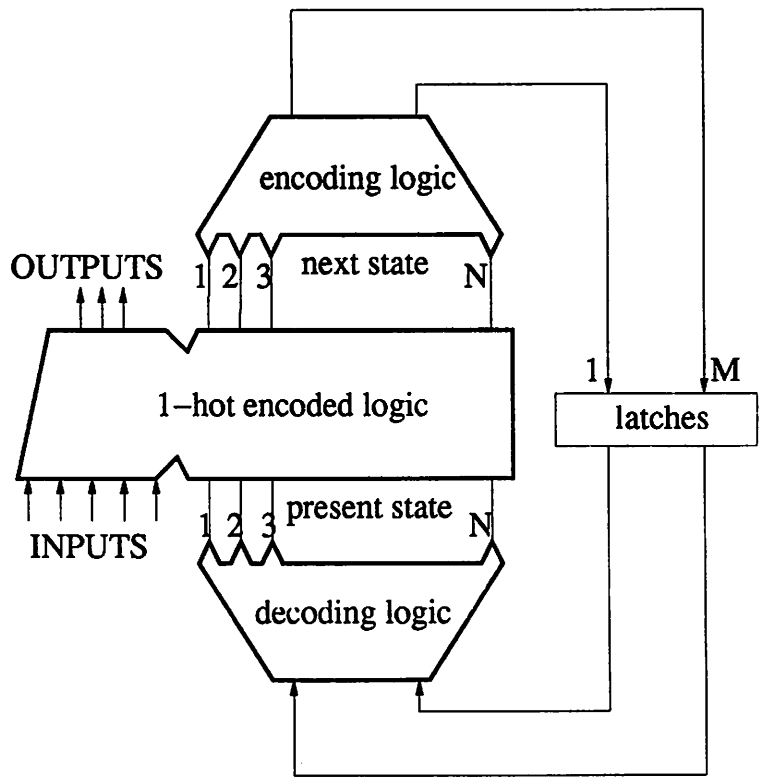


Figure 5.7: FSM with a k-hot encoding

derived from a 1-hot encoded machine.

For an FSM with N states, the number of registers, M, required when k-hot codes are used is given by the equation

$$\binom{M-1}{k} < N \leq \binom{M}{k}$$

Asymptotically $M \sim O(N^{1/k})$. The decoding logic consists of N functions, each of which produces one input to the the 1-hot encoding logic (corresponding to a state). Each state is determined by a conjunction of k variables. Thus the complexity of the decoding logic (without any sharing of common cubes) is kN. For the encoding logic that follows the 1-hot logic, there are M outputs to be produced from the N inputs. For every state, k of the M

outputs are turned on. Thus, each output is turned on for approximately $\begin{pmatrix} M-1 \\ k-1 \end{pmatrix}$ states. The complexity of the encoding logic is therefore $M \begin{pmatrix} M-1 \\ k-1 \end{pmatrix}$. Since $M \sim O(N^{1/k})$, the output logic has a complexity that is of the order of N.

By using a k-hot encoding, an additional area of $(k+1)N$ is required over the 1-hot implementation. However, the number of registers reduced from N to $N^{1/k}$. Assuming that a register has an area equal to C literals (typically $C = 8$), the area of the k-hot implementation exceeds the area of the 1-hot implementation by

$$(k+1)N - C(N - N^{1/k})$$

For a choice of $C = 8$ and small values of k, the area of the 2-hot implementation has smaller total area than the 1-hot implementation. The largest reduction in area occurs for $k = 2$. As is clear, there is a delay penalty associated with the k-hot encoding. The initial decoding logic consists of gates with k inputs. These can be represented using a circuit of depth $\lceil \log_2 k \rceil$. Similarly the encoding logic consists of gates with approximately $\begin{pmatrix} N-1 \\ k-1 \end{pmatrix}$ inputs and so the depth of the encoding logic is logarithmic in this number. For large N, the depth of the output logic is simply $(1 - 1/k)\log_2 N$. Thus the depth of the circuit increases by $\lceil \log_2 k \rceil + (1 - 1/k)\log_2 N$. The additional depth increase is smallest for $k = 2$. This calculation is very approximate as it ignores that arrival times of the outputs of the 1-hot implementation. For the case when these arrival times are skewed, the trees generating the encoded next-states can be implemented in depth much less than the suggested value of $(1 - 1/k)\log_2 N$.

This approximate computation suggests that 2-hot encoding of FSM's would yield a compromise solution, one that has depth close to that of the 1-hot implementation and area that is better than the 1-hot machine. In assigning the 2-hot codes, there is a certain degree of optimization that can be performed. Neighboring codes are assigned to states that result in the same next state under the same input. This allows common cubes to be extracted from the implementation. Recall that the state-assignment program MUSTANG [14] uses a similar heuristic to assign codes. The difference between 2-hot encoding and MUSTANG is that the codes in the latter are not restricted to have only 2 bits set. Table 5.3 shows the total area (measured as the sum of the logic and register area) and the depth of circuits

Example	JEDI		1-hot		2-hot	
	area	**depth**	**area**	**depth**	**area**	**depth**
cse	385	19	507	7	532	10
donfile	203	11	562	5	486	9
dk16	500	22	644	9	815	9
dk512	149	7	225	6	200	7
exl	480	16	503	12	550	11
keyb	403	18	597	11	812	12
styr	914	23	944	12	1113	12
sl	376	14	752	11	813	10
sla	397	15	535	9	755	10
tbk	487	17	1563	12	1858	13
tma	390	15	405	9	425	8

JEDI , **1-hot, 2-hot**	Method used to assign binary codes to symbolic states
area	equal to **lits** + 8 × **registers**
depth	Depth of optimized logic in 2-input gate form

Table 5.3: Comparison of state assignment techniques

obtained using JEDI , 1-hot and 2-hot encodings.

For this experiment the 2-hot encoding did not exploit any optimization from assigning codes based on the MUSTANG heuristic. Long run-times were encountered when optimizing circuits that were assigned binary codes using the 2-hot heuristic based on MUSTANG. The reason for this is under investigation. One possible explanation may be based on the structure of the don't care set that corresponds to the unreachable states. If the set of reachable states is S, then the representation of the don't care set corresponding to the unreachable states is

$$\overline{\bigvee_{(l_i,l_j)\in S} l_i \cdot l_j}$$

Since the sum-of-products representation of such functions can be very large, the SIS command **full_simplify** fails to finish in a reasonable amount of time. Only 5 of the 11 examples finished in reasonable time when the area-saving heuristic of MUSTANG was used to generate 2-hot encoded implementations.

The results of this experiment indicate that the 2-hot encoding style has area and delay comparable to the 1-hot encoded machine (assuming that a register has an area equivalent to 8 literals). The depth of the circuits encoded using JEDI is consistently larger. The reason for this may be twofold — the next-state equations are more complex and the greater sharing between the next-state functions (that results in smaller area) leads to greater depth of the optimized network. Surprisingly, the 2-hot encoding has area worse than the 1-hot encoding. Apart from not using the area-saving heuristic to assign 2-hot codes we also note that the asymptotic computation does not hold when the machines have a small number of states. The 2-hot implementation was not generated from the 1-hot implementation by appending the encoding and decoding logic. Instead, 2-hot codes for the states were used to generate the initial two-level description of the logic which was subsequently optimized. It is unclear how the state-assignment affected the subsequent performance optimization.

5.4 Limitations of current approaches

In previous sections we formulated the combinational optimization problem by specifying performance constraints in the form of input arrival times and output required times. This formulation is adequate when there are only flip-flops in the design. The clocking scheme gives rise to arrival and required time constraints for the combinational logic bounded by flip-flops. In designs containing level-sensitive latches, logic paths extend over multiple cycles and it is the total delay *along a path* that determines whether timing constraints are met or not. Section 5.1 described briefly the constraints that need to be met for proper circuit operation of level-sensitive circuits.

If short-path constraints are ignored then the long-path constraints lead to constraints between latch outputs and latch inputs. For a piece of combinational logic with m inputs and n outputs, the point-to-point specification of constraints consists of mn values. c_{ij} is the constraint on the longest path between input I_i and output O_j. In the absence of point-to-point optimization, we would like to set appropriate terminal constraints (arrival and required times) such that meeting the terminal constraints will ensure that the point-to-point constraints are met.

One way of doing so is to assign all the inputs an arrival time of 0. The output

O_j is assigned a required time

$$r(O_j) = \min_{i \in \{1,\ldots,m\}} c_{ij} \qquad \forall j = \{1,\ldots,n\}$$

However, this may drastically overconstrain the problem. We would like the optimization problem specified by the terminal constraints to be "similar" to the one posed by the point-to-point formulation. Since there are only $m + n$ terminal constraints compared to mn point-to-point constraints, the terminal constrained problem has a smaller feasible region that is contained in the feasible region for the point-to-point constrained problem.

It is possible to treat the arrival and required times as variables and then determine the values that provide the greatest flexibility in optimization by solving a mathematical programming problem. If a_i represents the arrival time of input I_i and r_j represents the required time for output O_j, then in order to satisfy the point-to-point constraint between I_i and O_j we require that

$$(r_j - a_i) \leq c_{ij} \qquad \forall (i,j) | c_{ij} < \infty$$

These inequalities need to be satisfied for any set of terminal constraints to be consistent with the point-to-point constraints. For each solution to the inequalities, we derive a measure of how much the terminal constraints overconstrain the optimization compared to the point-to-point constraints. The delay between an input-output pair, d_{ij}, is less than $r_j - a_i$. We would like this delay to be as close to the constraint. In other words we would like to keep $c_{ij} - r_j + a_i$ small. Since there are many valid input-output pairs, we consider the objective function to minimize the sum of this difference. This poses a linear objective function. This formulation is called CAV (Cumulative Absolute Violation) and its linear program formulation is

$$\min \sum_{(i,j)|c_{ij}<\infty} (c_{ij} + a_i - r_j)$$

subject to

$$\begin{array}{rl}
(r_j - a_i) \leq c_{ij} & \forall (i,j) | c_{ij} < \infty \\
a_i \geq 0 & i = \{1,\ldots,m\} \\
r_j \geq 0 & j = \{1,\ldots,n\}
\end{array}$$

Example

Consider the following set of point-to-point constraints for a circuit with three inputs and three outputs. The c_{ij} values are specified in arbitrary delay units (du).

c_{ij}	I_1	I_2	I_3
O_1	4	2	6
O_2	6	1	4
O_3	4	3	6

If all the inputs are given arrival time equal to 0, then the required times at the outputs are forced to be $r(O_1) = 2$, $r(O_2) = 1$, $r(O_3) = 3$. The total overconstraining that this causes adds up to 16 du. On the other hand, solving the linear program formulation results in the solution $a(I_1) = 2$, $a(I_2) = 3$, $a(I_3) = 3$, $r(O_1) = 5$, $r(O_2) = 4$, $r(O_1) = 5$ with an objective value of 6. Thus this assignment of terminal constraints overconstrains the original problem by only 6 du. The path between I_1 and O_2 has been constrained the worst — a target constraint $c_{12} = 6$ has been reduced to a path constraint of 2 du ($r(O_2) - a(I_1)$ = 2). Such a severe overconstraining (the new target is 33% of the actual constraint) may not be desirable. One way to avoid this is to introduce scaling factors into the objective function. The objective function

$$\min \sum_{(i,j)|c_{ij}<\infty} (a_i - r_j)/c_{ij}$$

results in the problem CPV (Cumulative Percentage Violation) since it minimizes the percentage violations. For the above example, the CPV formulation gives the same solution as the CAV formulation.

A linear sum of the violations provides an objective function that judges the total violation. This may result in severly overconstraining some paths. In practice, it is possible to achieve only a certain degree of speedup along a path. Therefore an alternative objective may be to minimize the maximum absolute violation (MAV) or to minimize the maximum percentage violation (MPV). The problem MAV results in the following mathematical programming problem

$$\min \max_{(i,j)|c_{ij}<\infty} (c_{ij} + a_i - r_j)$$

subject to

$$\begin{aligned} (r_j - a_i) &\le c_{ij} & \forall (i,j)|c_{ij} < \infty \\ a_i &\ge 0 & i = \{1,\ldots,m\} \\ r_j &\ge 0 & j = \{1,\ldots,n\} \end{aligned}$$

By setting $c_{ij} + a_i - r_j$ to be another variable K_{ij} and introducing a parameter p that exceeds all the K_{ij} values, this problem can be rewritten as a linear program.

$$\min p$$

subject to

$$\begin{array}{rl} 0 \leq (K_{ij}) \leq p & \forall (i,j) | c_{ij} < \infty \\ (K_{ij}) = c_{ij} + a_i - r_j & \forall (i,j) | c_{ij} < \infty \\ a_i \geq 0 & i = \{1, \ldots, m\} \\ r_j \geq 0 & j = \{1, \ldots, n\} \end{array}$$

When applied to the example in this section, the MAV formulation produces the following terminal constraints — $a(I_1) = 0$, $a(I_2) = 3$, $a(I_3) = 0$, $r(O_1) = 4$, $r(O_2) = 4$, $r(O_1) = 4$. The maximum violation for any path is just 2 du as opposed to the 4 du for the CTV and CPV formulations. The total violation, over all paths, increases from 6 du to 9 du. This solution has the drawback that between I_2 and O_3, a delay constraint of 3 has been replaced by a constraint of 1 (a 66% decrease in the constraint).

The maximum percentage violation can be reduced to 44% by using the MPV formulation. The MPV formulation is identical to the MAV formulation except that the constraint relating p and K_{ij}'s is replaced by

$$0 \leq K_{ij} \leq p\, c_{ij} \qquad \forall (i,j) | c_{ij} < \infty$$

The solution for the MPV formulation is to set $a(I_1) = 0$, $a(I_2) = 2.33$, $a(I_3) = 0$, $r(O_1) = 3.44$, $r(O_2) = 3.33$, $r(O_1) = 4$. The total violation increases to 11. However, using these terminal constraints, no path is constrained by more than 44% over the point-to-point constraints.

The above experiments and formulations provide two different objectives that can be used to judge the quality of the terminal constraints in approximating the point-to-point constraints. The CAV formulation is useful as a measure of getting the circuit into a configuration that has the least violation. It may result in severly overconstraining some paths. This can be overcome using the MPV formulation which will reduce the overconstraining to be a small fraction of the current constraint. Since it is difficult to judge the potential for reducing delays along specific paths, there is no straightforward way of determining which formulation will be more likely to succeed.

5.5 Conclusions

The optimization of synchronous circuits is the key problem that is addressed in this thesis. The techniques of combinational logic resynthesis developed in the previous chapters are used to reduce the cycle time of a synchronous circuit. For general cyclic circuits the generation of constraints on combinational logic is done using heuristic techniques whereas in the case of pipelined circuits the timing constraints on the combinational logic are tight, i.e. those constraints are necessary and sufficient to meet the desired cycle-time for the pipeline.

The need for a more general "point-to-point" optimization is introduced. This type of optimization procedure can address multi-cycle paths that have not been addressed in this chapter. Short-path constraints have also been ignored in this discussion. The presence of such constraints results in ensuring that the delay along specific paths lie within a range. The lower limit of this range is required to satisfy short-path constraints while the upper-limit ensures that the long-path constraints will be satisfied. The presence of two-sided constraints on path delays requires accurate delay computation. We have used the topological delay as an approximation of circuit delay. More accurate delay computation that accounts for the variability in gate delays and the logic functionality of gates [44] needs to be used. This complicates the analysis problem and the optimization problem under such conditions is overwhelmingly difficult.

To explore the full range of synchronous circuit optimization — the repositioning of registers, modification of the clocking scheme or re-encoding the symbolic machine — a change in design methodology may be required. The use of these optimizations may be restricted by the design-methodology used at a particular design site. As an example, consider the repositioning of registers. This optimization may not be allowed when simulation data for a design has been designed based on specific latch positions. By moving the latches the simulation data may be invalidated. Design methodologies may constrain the degree to which synchronous optimizations can be applied. However, for designers of high-performance systems who want to explore the full range of optimizations, these techniques offer considerable advantage over the traditional synthesis techniques of only modifying the combinational logic.

Chapter 6

Conclusions

The main contribution of this research has been to develop an understanding of the factors that affect the circuit delay. This understanding has allowed us to build an optimization procedure that can exploit the many different techniques available to reduce the circuit delay at different stages of synthesis.

The application of delay improving local transformations (those that provide an improvement in delay locally) to affect a global decrease in delay has been proposed. The proposed algorithm to improve the circuit delay can identify the parts of a circuit that have to be improved to guarantee a decrease in delay and is able to predict the amount of improvement that is possible. A set of transformations that meet this predicted improvement with a small cost in area can be selected. The application of local transformations presents a framework in which different optimization techniques can be simultaneously leveraged to reduce circuit area. This flexibility allows a circuit designer to use the power of all the existing techniques to improve the performance of their circuits. It should be mentioned that local transformations may be used to reduce the depth of an abstract representation of a function or to minimize the delay of a mapped circuit. In the latter case the local transformations also include buffering techniques that enable a gate to drive a large number of destinations without a significant loss in speed. A top-down algorithm that generates fanout-trees with close to the minimum area and delay has been proposed. It outperforms other techniques suggested in the literature if the area-delay product is used as the comparison measure. However, improvements to existing procedure are required to make the application of local transformations on mapped circuits competitive with other heuristics like that of [18].

For circuits containing memory elements, an additional degree of flexibility is available since the registers may be repositioned without changing the circuit functionality. The different techniques that are available to get a circuit that can be clocked faster — optimization of combinational logic, moving registers and changing the clocking scheme — can all be treated as transformations that exploit the slack along a path to reduce the violation of the timing constraints. For circuits that have special structures like pipelines or standard FSM structures, optimizations that exploit that these structures are presented. For pipeline circuits, the latches can be moved to the periphery of the circuit to expose the maximum amount of combinational logic to timing optimization techniques. Finite-state machines may be unrolled over several time periods to create greater opportunities for optimization.

The techniques presented in this thesis augment the manual improvement techniques currently used by designers. Although no theoretical bounds can be proved on the quality of the circuits generated, experimental results indicate that the techniques are useful in improving the delay of synthesized circuit. The optimization of an entire ASIC chip has been performed to demonstrate the application of the proposed techniques to improve the performance of the design.

In conclusion, logic synthesis enables designers to
design circuits fast
and the current research adds to the capability by being able to
design fast circuits.

Bibliography

[1] Jonathan Allen. Performance-Directed Synthesis of VLSI Systems. *Proceedings of the IEEE*, 78(2):336–355, February 1990.

[2] Hiromu Ariyoshi. Cut-Set Graph and Systematic Generation of Separating sets. In *IEEE Transactions on Circuit Theory*, pages 233–240, May 1972.

[3] P. Ashar, S. Dey, and S. Malik. Exploiting Multi-Cycle False Paths in the Performance Optimization of Sequential Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 510–517, 1992.

[4] K. Bartlett, G. Borriello, and S. Raju. Timing Optimization of Multiphase Sequential Logic. *IEEE Transactions on Computer-Aided Design*, 10(1):51–62, January 1991.

[5] K. Bartlett, W. Cohen, A. deGeus, and G. Hachtel. Synthesis and Optimization of Multilevel Logic under Timing Constraints. *IEEE Transactions on Computer-Aided Design*, CAD-5(4):582–595, October 1986.

[6] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, and L. H. Trevillyan. Efficient Techniques for Timing Correction. In *Proceedings of the Internationmal Symposium on Circuits and Systems*, pages 415–419, 1990.

[7] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35, August 1986.

[8] A. F. Champernowne, L. B. Bushard, J. T. Rusternolz, and J. R. Schomburg. Latch-to-Latch Timing Rules. *IEEE Transactions on Computers*, C-39(6):798–808, June 1990.

[9] K. C. Chen and S Muroga. Timing Optimization for Multi-Level Combinational Circuits. In *Proceedings of the Design Automation Conference*, pages 339–344, 1990.

[10] M. Dagenais and N. Rumin. Automatic Determination of Optimal Clocking Parameters in Synchronous MOS Circuits. In *Advanced Research in VLSI: Proc. of the 5th MIT Conference*, pages 19–33, 1988.

[11] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L.Trevillyan. LSS: A system for Production Logic Synthesis. *IBM Journal of Research and Development*, 28(5):326–328, September 1984.

[12] G. De Micheli. Performance-Oriented Synthesis of Large-Scale Domino CMOS Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):751–765, 1987.

[13] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology Mapping in MIS. In *Proceedings of the International Conference on Computer-Aided Design*, pages 116–119, November 1987.

[14] S. Devadas, H. K. Ma, A. R. Newton, and A. Sngiovanni-Vincentelli. MUSTANG: State Assignment of Finite State Machines targeting Multilevel Implementations. *IEEE Transactions on Computer-Aided Design*, CAD-7:1290–1299, December 1988.

[15] S. Dey, M. Potkonjak, and S. G. Rothweiler. Performance Optimization of Sequential Circuits by Elimiating Retiming Bottlenecks. In *Proceedings of the International Conference on Computer-Aided Design*, pages 504–509, 1992.

[16] Paul E. Dunne. *The complexity of Boolean networks.* Academic Press, 1988.

[17] J. P. Fishburn. A Depth-Decreasing Heuristic for Combinational Logic. In *Proceedings of the Design Automation Conference*, pages 361–364, 1990.

[18] J. P. Fishburn. LATTIS: An Iterative Speedup Heuristic for Mapped Logic. In *29 ACM/IEEE Design Automation Conference*, pages 488–491, 1992.

[19] J. P. Fishburn and A. E. Dunlop. TILOS: A Posynomial Programming Approach to Transistor Sizing. In *Proceedings of the International Conference on Computer-Aided Design*, pages 326–328. IEEE, 1985.

[20] J. Frankle. Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing. In *Proceedings of the Design Automation Conference*, pages 536–542, 1992.

[21] T. Gao, P. M. Vaidya, and C. L. Liu. A New Performance Driven Placement Algorithm. *Proceedings of the International Conference on Computer-Aided Design*, pages 44–47, 1991.

[22] M. C. Golumbic. Combinatorial Merging. *IEEE Transactions on Computers*, 25(11):1164–1167, November 1976.

[23] P. Gutwin and P. C. McGeer. Delay Predictor for Technology-Independent Logic Equations. In *Proceedings of the International Conference on Computer Design*, pages 468–471, 1992.

[24] L. Hagen and A. B. Kahng. A New Approach to Effective Circuit Clustering. In *Proceedings of the International Conference on Computer-Aided Design*, pages 422–427, 1992.

[25] Mark Hoffman and Jac. K. Lim. Delay Optimization of Combinational Static CMOS Logic. In *Proceedings of the Design Automation Conference*, 1987.

[26] H. J. Hoover, M. M. Klawe, and N. J. Pippenger. Bounding Fan-out in Logical Networks. *Journal of the Association for Computing Machinery*, 31(1):13–18, January 1984.

[27] V. M. Hrapcenko. Depth and Delay in a Network. In *Soviet Math Dokl.*, pages 1006–1009, 1978.

[28] M. A. B. Jackson, A. Srinivasan, and E. S. Kuh. A Fast Algorithm for Perfromance-Driven Placement. *Proceedings of the International Conference on Computer-Aided Design*, pages 328–331, 1990.

[29] R. Kamiwakai, M. Yamada, T. Chiba, K. Furumaya, and Y. Tsuchiya. A Critical Path Delay Check System. In *Proceedings of the Design Automation Conference*, pages 118–123, 1981.

[30] K. Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *Proceedings of the Design Automation Conference*, pages 341–347. ACM/IEEE, June 1987.

[31] K. Keutzer, S. Malik, and A. Saldanha. Is Redundancy Necessary to Reduce Delay? In *Proceedings of the Design Automation Conference*, pages 228–234, 1990.

[32] M. C. Lega. Mapping Properties of Multi-Level Logic Synthesis Operations. In *Proceedings of the International Conference on Computer Design*, pages 257–261. IEEE, October 1988.

[33] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Advanced Research in VLSI: Proceedings of the Third Caltech Conference*, pages 86–116. Computer Science Press, 1983.

[34] C.E. Leiserson and J.B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.

[35] W. N. Li, A. Lim, P. Agarwal, and S. Sahni. On the Circuit Implementation Problem. In *Proceedings of the Design Automation Conference*, pages 478–483, 1992.

[36] M. Lightner and W. Wolf. Experiments in Logic Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 286–289, 1988.

[37] B. Lin and A. R. Newton. Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages. In *Proceedings of the International Conference on VLSI*, pages 187–196, August 1989.

[38] Bill Lin and F. Somenzi. Minimization of Symbolic Relations. In *Proceedings of the International Conference on Computer-Aided Design*, pages 88–91. IEEE, 1990.

[39] P. P. P. van Ginneken Lukas. Timing optimization in the BooleDozer synthesis system. Berkeley-Boulder-Stanford summer meeting, April 1991.

[40] F. Mailhot and G. De Micheli. Technology Mapping using Boolean Matching. In *The Proceedings of the European Conference on Design Automation*, pages 180–185, March 1990.

[41] S. Malik, E.M. Sentovich, R.K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimization of Sequential Networks with Combinational Techniques. *IEEE Transactions on Computer-Aided Design*, 10(1):74–84, January 1991.

[42] S. Malik, K.J. Singh, R.K. Brayton, and A. Sangiovanni-Vincentelli. Performance Optimization of Pipelined Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 410–413, November 1990.

[43] Morris M. Mano. *Digital Logic and Computer Design.* Prentice-Hall, 1979.

[44] P. McGeer, A. Saldanha, R. K. Brayton, and A. Sangiovanni-Vincentelli. *Logic Synthesis and Optimization*, chapter Delay models and Exact Timing Analysis. Kluwer Academic Press, T. Sasao editor edition, October 1992.

[45] P. C. McGeer, R. K. Brayton, A. L. Sangiovanni-Vincentelli, and S. K. Sahni. Performance Enhancement through the Generalized Bypass Transform. In *Proceedings of the International Conference on Computer-Aided Design*, pages 184–187. IEEE, 1991.

[46] L Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits.* Memorandum No. UCB/ERL M85/90, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, November 1985.

[47] R. Nair, C. L. Berman, P. S. Hague, and E. J. Yoffa. Generation of Performance Constraints for Layout. *IEEE Transactions on Computer-Aided Design*, CAD-8(7):860–874, July 1989.

[48] P. G. Paulin and F. Poirot. Logic Decomposition Algorithms for the Timing Optimization of Multi-Level Logic. In *Proceedings of the International Conference on Computer Design*, pages 329–333, 1989.

[49] P. Penfield and J. Rubinstein. Signal Delay in RC Tree Networks. In *Proceedings of the 2nd Caltech VLSI Conf.*, pages 269–283, 1981.

[50] R. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, UC Berkeley, April 1989. UCB/ERL M89/49.

[51] R. Rudell. Technology Mapping for Delay. part of Ph.D. thesis, March 1989.

[52] K. Sakallah, T. N. Mudge, and O. A. Olukotun. Check t_c and min t_c: Timing Verification and Optimium Clocking of Synchronous Digital Circuits . In *Proceedings of the International Conference on Computer-Aided Design*, pages 552–555, 1990.

[53] Alexander Saldanha. *Performance and Testability Interactions in Logic Synthesis.* PhD thesis, UC Berkeley, October 1991. UCB/ERL M91/100.

[54] T. Sasaki, A. Yamada, T. Aoyama, K. Hasegawa, S. Kato, and S. Sata. Hierarchical Design Verification for Large Digital Systems. In *Proceedings of the Design Automation Conference*, pages 105–112, 1981.

[55] G. Saucier, C. Duff, and F. Poirot. State Assignment Using a New Embedding Method Based on an Intersecting Cube Theory. In *Proceedings of the Design Automation Conference*, pages 321–326, 1989.

[56] John E. Savage. *The Complexity of Computation.* Wiley Interscience Publications, 1976.

[57] H. Savoj, R. K. Brayton, and H. J. Touati. Extracting Local Don't Cares for Network Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 514–517. IEEE, November 1991.

[58] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis.* Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[59] N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Graph Algorithms for Efficient Clock Schedule Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 132–136, 1992.

[60] N. V. Shenoy, K. J. Singh, R. K. Brayton, and L. M. Sangiovanni-Vincentelli. On the Temporal Equivalence of Sequential Circuits. In *Proceedings of the Design Automation Conference*, pages 405–409, 1992.

[61] K. J. Singh, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing Optimization of Combinational Logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 282–285. IEEE, 1988.

[62] A. Stoelzle, S. Narayanaswamy, K. Kornegay, H. Murveit, and J. Rabaey. A VLSI Wordprocessing Subsystem for a Real Time Large Vocabulary Continuous Speech Recognition System. In *Proceedings of the Custom Integrated Circuits Conference*, 1989.

[63] T. G. Szymanski. LEADOUT: A Static Timing Analyser for MOS Circuits. *Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, 1986.

[64] T. G. Szymanski. Computing Optimal Clock Schedules. In *Proceedings of the Design Automation Conference*, pages 399–404. IEEE/ACM, 1992.

[65] T. G. Szymanski and N. Shenoy. Verifying Clock Schedules. In *Proceedings of the International Conference on Computer-Aided Design*, pages 124–131, 1992.

[66] H. Touati. *Performance-oriented Technology Mapping.* PhD thesis, UC Berkeley, November 1990. UCB/ERL M90/109.

[67] H. Touati, C. Moon, R. K. Brayton, and A. Wang. Performance-Oriented Technology Mapping. In *Proceedings of the sixth MIT VLSI Conference*, pages 79–97. MIT Press, April 1990.

[68] H. J. Touati, H. Savoj, and R. K. Brayton. Delay Optimization of Combinational Logic circuits by Clustering and Partial Collapsing. In *Proceedings of the International Conference on Computer-Aided Design*, pages 188–191. IEEE, November 1991.

[69] S. Tsukiyama, I. Shirakawa, and H. Ozaki. An Algorithm to Enumerate All Cutsets of a Graph in Linear Time per Cutset. In *Journal of the Association for Computing Machinery*, pages 619–632, October 1980.

[70] S. H. Unger and C Tan. Clocking Schemes for High-Speed Digital Circuits. *IEEE Transactions on Computers*, C-35(10):880–895, October 1986.

[71] J. Vasudevamurthy and J. Rajski. A Method for Concurrent Decomposition and Factorization of Boolean Expressions. In *Proceedings of the International Conference on Computer-Aided Design*, pages 510–513, November 1990.

[72] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations. *IEEE Transactions on Computer-Aided Design*, 9(9):905–924, September 1990.

[73] D. E. Wallace and M. S. Chandrashekhar. High-Level Delay Estimation for Technolgy-Independent Logic Equations. In *Proceedings of the International Conference on Computer-Aided Design*, pages 188–191. IEEE, 1990.

[74] D. E. Wallace and C. H. Sequin. ATV: An Abstract Timing Verifier. *Proceedings of the Design Automation Conference*, pages 154–159, 1988.

[75] A. R. R. Wang. *Algorithms for Multi-level Logic Optimization.* PhD thesis, UC Berkeley, April 1989. UCB/ERL M89/50.

[76] N. Weiner and A. L. Sangiovanni-Vincentelli. Timing Analysis in a Logic Synthesis Environment. *Proceedings of the Design Automation Conference*, pages 655–661, 1989.

[77] M. A. Wold. Design Verification and Performance Analysis. In *Proceedings of the Design Automation Conference*, pages 264–270, 1978.

[78] D. Wong, G. De Micheli, and M. Flynn. Inserting Active Delay Elements to Achieve Wave Pipelining. In *Proceedings of the International Conference on Computer-Aided Design*, pages 270–273, 1989.

[79] Saeyang Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. In *Technical report.* MCNC, January 1991.

[80] K. Yoshikawa, H. Ichiryu, H. Tanishita, S. Suzuki, N. Nomizu, and A. Kondoh. Timing Optimization on Mapped Circuits. In *Proceedings of the Design Automation Conference*, June 1991.

[81] G. Zewi, U. Barkai, J. Ben-Simon, and E. Kadar. An Accurate Slope-Dependent Delay Model. Communication from Intel Corporation.