# PERFORMANCE-ORIENTED TECHNOLOGY MAPPING

by

Hervé J. Touati

Memorandum No. UCB/ERL M90/109

28 November 1990

# PERFORMANCE-ORIENTED TECHNOLOGY MAPPING

by

Hervé J. Touati

Memorandum No. UCB/ERL M90/109

28 November 1990

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# Performance-Oriented Technology Mapping



Hervé J. Touati

# Performance-Oriented Technology Mapping

Hervé J. Touati

University of California
Berkeley, California
Ph.D. Thesis

Department of Electrical Engineering
and Computer Science
Computer Science Division

## Abstract

This thesis presents a variety of techniques to minimize circuit delay during the translation of a set of Boolean equations into a list of connected logic gates that can be used for the manufacturing of combinational digital circuits. This translation process is called *technology mapping.* The first contribution of this work is to present an optimal algorithm to implement a Boolean circuits that can be represented as trees using an extension of known tree covering algorithms. The second and more important contribution of this work is an in-depth analysis of *fanout optimization.* The fanout problem is the problem of distributing a signal to several destinations, where the signal may be required at different times, in order to minimize the overall delay. This work presents the most detailed theoretical study of the complexity of fanout optimization published so far, and a spectrum of heuristics to solve the fanout problem under realistic delay models. This thesis also introduces a simple algorithm that can be used to apply fanout optimization to an entire network. This algorithm yields an optimal application of fanout optimization in terms of delay, while keeping area increase of the circuit to a low value. To study the integration of tree covering and fanout optimization, this work introduces a technology independent delay model that characterizes precisely suboptimalities due to inbalances in a network. This is the first technology independent delay model that models the delay through a node as a function of the arrival time distribution at a node. This delay model can be used to derive analytically optimal solutions in simple cases, which can be used to measure the suboptimality of heuristics. An extension to tree covering is then suggested, and shown to provide significant delay reductions for a relatively heavy cost in area. Finally this work investigates technology independent delay optimization techniques based on partial or total collapsing of logic, and shows that further delay reductions can be achieved with these techniques possibly at a higher cost in area.

---

Prof. Robert K. Brayton
Thesis Committee Chairman

# Acknowledgements

First I would like to thank my advisor, Pr. Robert Brayton, for his undivided support and encouragement. Much of this work would not have been possible without his help. I am also very grateful to Pr. Sangiovanni-Vincentelli for accepting the burden of being a second reader of this thesis and for many exciting discussions. I am also indebted to Pr. Kobayashi, of the Mathematics Department, who was kind enough to accept to be a member of my thesis committee. I owe much to Pr. Alan Smith and Pr. Alvin Despain for teaching me many of the fundamentals of computer science research. I also would like to thank Dr. Kurshan of AT&T Bell Laboratories for many interesting discussions, Pr. Susan Graham and Dr. Kurokawa of IBM Japan for their support and advice early in my graduate student life. Finally I would like to acknowledge the French Ministry of Transportation, DARPA, IBM and DEC who granted me the privilege of their support during my graduate studies.

Above all, I would like to thank my wife, Atsuko, and my daughter, Marianne Ayaka, for their love and support, and for coping with the hardships of graduate student life. I owe to Marianne the habit of waking up early every day and to Atsuko a strong desire to graduate. These would be major contributions in any field of research.

I would like to express my thanks to the CAD group in general, for providing such a stimulating and exciting research environment.

Of course, many friends deserve special thanks. I was told that it is more polite to list one's friends by alphabetic order, so that only those whose name start by 'A' do not get offended. So many thanks to Pranav Ashar; Mary and Wendell Baker, for tolerating me as their office mate; Andrea Casotto; Fred Douglis; Paul Gutwin; Bruce Holmer; Kurt Keutzer; Luciano Lavagno; Bill Lin; Sharad Malik; Rick McGeer; Cho Moon; Rajeev Murgai; Antony Ng; Terry Regier; Rick Rudell; Rafael Saavedra-Barrera; Alex Saldanha; Hamid Savoj; Luigi Semenzato; Ellen Sentovitch; Narendra Shenoy; K. J. Singh; Ashok Singhal; Greg Sorkin; Peter Van Roy, for sharing Belgian chocolates with me; Steve Viavant; Albert Wang; Yosinori Watanabe; and Greg Whitcomb.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Books are not made to be believed,
but to be submitted to examination.
— UMBERTO ECO, *The Name of the Rose* (1980)

## 1.1 Overview

*Logic synthesis* is the process of transforming a set of Boolean equations into a network of gates that realizes the logic and minimizes some cost function. The cost function can be area, delay, testability or power consumption. Most often than not, it is a combination of these factors. For simplicity and efficiency, logic synthesis is usually decomposed in two phases: a *technology independent* phase, whose main objective is to simplify the logic; and a *technology dependent* phase, also called *technology mapping*, whose main role is to implement the logic using well characterized logic gates realized in a given technology [8].

The main focus of this thesis is to develop techniques to reduce circuit delay during the technology dependent phase of logic synthesis. Despite this focus of delay optimization, other costs are not ignored. In particular, some care has been given to limit the increase in circuit area whenever possible.

Technology independent delay optimizations are just as important as technology dependent delay optimizations. However, since they are performed at a higher level, they cannot be used with great accuracy until technology independent delay models and technology mapping algorithms of reliable performance are developed. Since high level delay models are still the subject of current research, technology independent delay optimizations

are only a topic of secondary priority in this research. Much work remains to be done in this area.

The technology mapping algorithms presented in this work rely on two main techniques: *tree covering* and *fanout optimization*. Tree covering algorithms were first studied Aho *et al.* [2, 3, 1] in the context of code generation for expressions, and were later adapted to technology mapping by Keutzer [24, 27] and Rudell [14, 36]. If the objective is to minimize circuit area, tree covering algorithms produce good quality results and are straightforward to use. However if the objective is to minimize circuit delay, tree covering needs to be extended even to generate optimal solution for trees. This extension is discussed in chapter 2.

Tree covering alone tends to generate poor quality implementations in terms of delay, because most circuits are not trees but directed acyclic graphs. The signal available at the output of a tree needs to be distributed to several destinations. Such a signal is called a *fanout signal*. With tree covering alone, the circuitry used to distribute a fanout signal to its destinations is implemented by default with a wire. In first approximation, if $n$ is the number of destinations of a fanout signal, the delay through this wire is of order $O(n)$. Using a simple buffer tree, this delay can be reduced to $O(\log n)$. It is thus very important, to minimize delay, to be able to insert buffer trees to reduce the delay incurred by the distribution of fanout signals. This optimization, called *fanout optimization* [5], is the main focus of chapter 3.

Tree covering can be formulated as the problem of minimizing delay through a *fanin node*, i.e. a node with several inputs but only one output. In a very similar way, fanout optimization can be formulated as the problem of minimizing delay through a *fanout node*, i.e. a node with one input and many outputs. To minimize delay through an entire circuit, we need to coordinate the use of tree covering and fanout optimization on the fanin and fanout nodes that compose the circuit. This global optimization problem is the focus of chapter 4.

These techniques provide a solid set of technology mapping algorithms on which we can rely to measure the effect of technology independent optimizations. Chapter 5 contains the results of some empirical studies of the effect of technology independent optimizations on the quality of the final, technology mapped implementation of a circuit. Finally chapter 6 summarizes the main results of this work.

In the remainder of this chapter we give the main definitions and notation used

throughout this thesis, and a description of the abstraction we use to model the physical behavior of circuit components.

## 1.2 Terminology and Notation

### 1.2.1 Mathematical Notation

By convention we use the letters $n$ and $m$ to denote integer valued constants, and the letters $i$, $j$ and $k$ to denote integer valued variables, usually indices. We use the letters $a$, $b$, $c$ to designate real valued constants, and $x$, $y$, $z$, $t$, $r$ and $\rho$ to designate real valued variables. Occasionally, we also used upper case letters.

We use other letters to designate certain quantities in specific contexts. In particular $b$, $g$ and $s$ are used as indices in a library of gates; $b$ is usually denotes a buffer and $s$ a source, i.e. a gate used at the root of a tree. We use $\alpha$, $\beta$ and $\gamma$ as constants that characterize the delay through a gate of a gate library. We also commonly use $d$ to denote the number of buffers in a library, and $n$ to denote the number of destinations of a fanout signal.

We use $\Sigma_n$ to denote the group of permutations of a set of $n$ elements. In that context, $\sigma$ is used to denote an element of $\Sigma_n$, i.e. a permutation. We use $\pi$ to denote the natural projection from a Cartesian to one of its components. For example, $\pi_A : A \times B \rightarrow A$, where $\pi_A((a,b)) = a$. $\mathcal{R}$ denotes the field of real numbers, and $\mathcal{R}_+$ the subset of $\mathcal{R}$ formed of the nonnegative real numbers.

### 1.2.2 Combinational Logic

We also use the letters $x$, $y$, $z$ to denote Boolean variables. We denote the Boolean *and* with a "." or with a space if the meaning is clear. We denote the Boolean *or* with a "+" and the negation with a bar over the expression to be negated. For example we would write $\overline{x+y} = \overline{x}.\overline{y}$.

A combinational logic circuit can be specified as a set of Boolean equations, with no cyclic dependencies, of the form $y_j = f_j(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ and $y_j$ denote Boolean variables and $f_j$ a Boolean function. There is a one-to-one mapping between a set of Boolean equations with no cyclic dependencies and a *Boolean network*, i.e. a directed acyclic graph graph $G = (V, E)$, in which a logic equation is associated to each node.

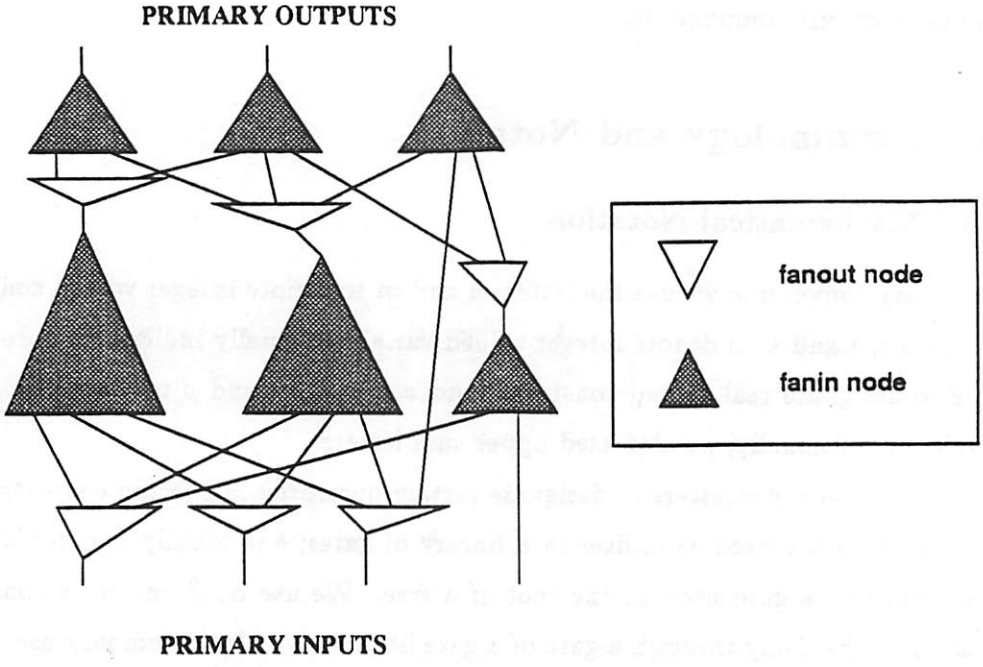


Figure 1.1: A Boolean Network

Boolean networks form a natural, very general, multi-level representation of a piece of combinational logic. Other representations of combinational logic, such as two-level *sum-of-products* [7, 37] or *binary decision diagrams* [10] are more useful in some contexts, but are more limited in terms of the class of functions they can express in a reasonable amount of space.

Some of the nodes of a Boolean network are distinguished as *primary inputs* and *primary outputs*, and represent respectively the inputs and outputs of the corresponding circuit. The *fanin* of a node $v$ of a Boolean network is the set of nodes $u$ whose output is directly connected to an input of $v$. Similarly, the *fanout* of a node $v$ is the set of nodes $u$ that have an input directly connected to the output of $v$. A node that has a fanout containing only one element is called a *fanin node*; a node that has a fanin containing only one element is called a *fanout node*. A Boolean network can always be decomposed into a network of fanin and fanout nodes, in such a way that a fanin node is only connected to fanout nodes, primary inputs or primary outputs, and a fanout node is only connected to

fanin nodes, primary inputs or primary outputs, as illustrated in Figure 1.1.

### 1.2.3 Physical Modeling

We model a technology as a reasonably small library of gates implementable in that technology. These gates are the only circuit primitives we claim to use. In addition, we suppose that these gates are fully characterized by a combinational logic function and a set of delay equations. This excludes latches and CMOS transmission gates. We also suppose that every interconnection of gates yields a valid circuit. This fails for ECL (different voltage levels; dot products) or in the presence of fanout limits. Taking care of latches or taking fanout limits into account can easily be done by a simple extension of the algorithms presented here.

Our algorithms can handle gate libraries composed of up to a few hundred gates, which is good enough to handle industrial CMOS standard cell libraries. Some of the most popular gates in standard cell CMOS technologies are 2-input NAND gates, that implement the function $out = \overline{a.b}$; inverters, that implement the function $out = \overline{a}$; XOR gates, that implement the function $out = \overline{a}b + a\overline{b}$; and-or-invert and or-and-invert gates, as for example, AOI22, that implements the function $out = \overline{a_1 a_2 + b_1 b_2}$. A given logic function can be implemented by several gates of different sizes and delay characteristics. The role of the technology mapper is to select not only a logic function that corresponds to a gate in the target library, but also to select the appropriate gate for a given logic function.

**Gate Area** In standard cell technology, we use *grid counts* to measure the area of a gate or cell. The grid count is the width of a cell relative to a standard design rule. Grid count is directly proportional to standard cell area before routing [31]. If a well characterized router is used for layout, grid count is a good estimate of final chip area.

**Gate Delay** To model gate delay, we use a simple linear delay model. This model characterizes the delay from an input pin $i$ to the output pin of a gate $g$ using a linear equation of the form $\alpha_{i,g} + \beta_{i,g}\,\gamma$. The coefficient $\alpha_{i,g}$ models the intrinsic delay through the gate; the coefficient $\beta_{i,g}$ models the load dependent delay; $\gamma$ is the capacitive load at the output of the gate, which is estimated by looking at the output connections of the gate. The model also distinguishes between rise and fall delays. If capacitive loads are restricted to be within a small range of values, and if delay coefficients were determined for that range of values,

this model is a reasonable first approximation, within 10% of actual delays [29]. Gate level delay models used in the industry are usually more complex that this linear, 0-order model. Industrial models usually rely on a simple first-order non-linear delay model, that computes the delay and its slope, i.e. a discretized version of the first derivative of the delay. A well-tuned version of this model can estimate physical delays within a few percent.

To estimate the capacitive load at the output of a gate $g$, we add the capacitive loads of the input pins of the gates driven by $g$ to a additional term representing the delay through the wires.

**Arrival Times, Required Times and Slack** Given *arrival times* at the primary inputs of the network, the arrival time at the output of any gate in the network is defined as the latest possible moment a transition may occur at the output of that gate. We use static timing analysis to compute arrival times, i.e. we assume that all paths through the logic can be activated. Techniques to detect *false paths* exist [33] but are too computationally expensive to be of practical use during technology mapping. In addition, one of the goals of technology mapping for delay is to make a large number of paths critical, reducing the need for more sophisticated delay estimators.

Given *required times* at the primary outputs of a network, we can also compute the required time at the outputs of any gate of the network by propagating required times throughout the network. The *slack* at the output of a gate is defined as the difference between the required time and the arrival time at that output.

**Technology Independent Models** The `literal` count is a good technology independent estimator of circuit area. The literal count of a Boolean network is the sum of the literal counts of each of its nodes. The literal count of a node is the number of literals appearing in a factored form representation of the logic function of the node. A literal is a term of the form $x$ or $\overline{x}$, where $x$ is a variable and $\overline{x}$ is the Boolean negation of $x$. For example, if the logic function of a node is: $x_1(x_2 + \overline{x_3}) + \overline{x_1}x_4$, the literal count of this node is 5. Literal count correlates well with circuit area [31].

Finding reliable technology independent estimators of circuit delay is still the subject of current research, despite recent advances [43]. Simple models based on the number of levels of logic with or without a corrective term for multiple fanouts are usually very unreliable. Good delay estimators are crucial to the accuracy and consistency of

technology independent delay optimization algorithms.

# Chapter 2

# Delay Optimization with Tree Covering

The most constant difficulty in contriving the engine
has arisen from the desire to reduce the time
in which the calculations were executed
to the shortest which is possible.
— CHARLES BABBAGE (1837)

## 2.1 Introduction

The problem of delay minimization of an arbitrary Boolean network is difficult in general. Ideally, we would like to find a network of gates that is functionally equivalent to a Boolean network and has minimum delay. For a given Boolean network, the number of such implementations is theoretically infinite. Even if we restrict ourselves to implementations of bounded size, the number of such implementations is still very large and there is no known method to explore the solution space efficiently. All the practical methods developed so far consist in modifying an initial representation of a Boolean network using transformations that preserve the behavior of the network and reduce the cost of its implementation.

Since the problem is so complex, it is in practice divided into two phases: a technology independent phase and a technology dependent phase, also called *technology mapping*. The purpose of the technology independent phase is to provide a Boolean network equivalent to the original circuit which can be implemented efficiently by the technology mapping algorithms. The role of the technology mapper is to compute a network of gates of

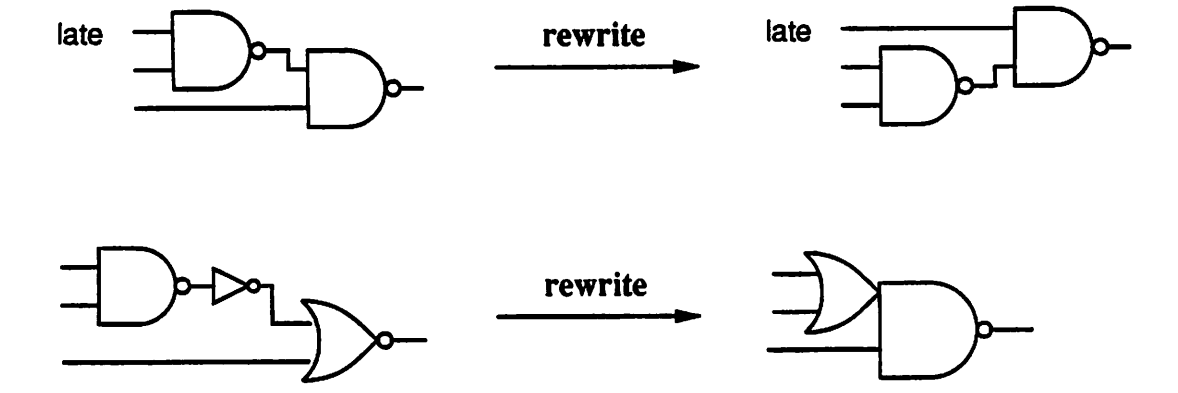


Figure 2.1: Example of Rules

minimum cost equivalent to a given Boolean network. The simplifying assumption we make about the technology mapper is that the structure of the network of gates it computes is derived from the structure of the Boolean network it takes as input. The technology mapper is not expected to modify the structure of the network drastically: such transformations are best left to the technology independent phase, not because doing so yields better results, but because it corresponds to a natural separation of concerns, and a simpler overall design.

The first technology mappers were rule-based (LSS [13], SOCRATES [19]), i.e. based on local transformations called *rules*. Examples of rules are given in Figure 2.1. Rules can be used to implement unimplemented logic, or simply to improve the quality of an already implemented set of gates. Rule-based systems are quite flexible and can generate circuits of good quality in terms of either area or delay. They can also be used in a postprocessing phase to improve the quality of a circuit generated by other algorithms. Unfortunately, rule-based systems suffer from two severe limitations: rules are library dependent and only local optimizations are possible in polynomial time.

An attractive alternative to rule-based technology mappers is the use of a divide-and-conquer strategy, where the network is partitioned into the largest pieces that can be handled with efficient, optimum, library independent algorithms. The most important of these algorithms is tree covering. Given a tree, expressed with simple primitives (e.g. 2-input NAND gates and inverters), tree covering can find a minimum cost covering of the tree by gates of a library. This covering can then be extracted to form an implementation of

the tree. This implementation is not in general a minimum cost implementation of the tree because it is a function of the decomposition of the tree into primitives, and also because in some cases the minimum cost solution cannot be expressed as a tree cover (i.e. when a minimum delay solution requires the introduction of buffers between gates, or when the use of Boolean identities is required to identify a match). However, tree covering usually yields good quality implementations, and has the merit of being very fast.

Tree covering was originally developed for code generation in retargetable compilers [1]. It was first introduced in the context of technology mapping by Keutzer [24] for producing minimum area implementations. There has been some earlier attempts to extend tree covering to produce minimum delay implementations, but these techniques were either *ad hoc* [26] or were not implemented [36]. In this chapter we introduce an elegant solution to the problem of optimal tree covering for delay, and present and discuss experimental results comparing this solution with simpler but suboptimal techniques.

Optimal tree covering can be solved in time linear in the number of nodes in the tree. The complexity of tree covering as a function of the number and size of the gates in the library depends on which tree pattern matching algorithm is used. The fastest pattern matching algorithms build an automaton that summarizes all possible patterns matched by gates in the library. At the cost of some preprocessing time, these algorithms avoid having to traverse the same substructures several times. A good overview of tree pattern matching algorithms can be found in [20]. The fastest algorithms are based on a bottom up traversal of the trees, but they have the largest space requirements. An interesting way to reduce this space overhead can be found in [11]. No technology mapper has been based on bottom-up tree covering. The second fastest algorithms are based on top down string matching techniques [1]. These algorithms were used in Dagon [24]. In contrast, `misII` technology mapper [14] uses a more conventional but slower matching algorithm, that simply enumerates all patterns and requires little preprocessing. For simplicity, we use the slower matching algorithm used in `misII`. The choice of the matching algorithm has no influence on the quality of the final result and thus has no influence on our experimental results.

The rest of this chapter is organized as follows. In section 2.2 we review the basic tree covering algorithm for minimum area and show how it can be directly extended to minimize delay if load values are supposed to be constant. Then in section 2.3 we show how to extend this algorithm to take exact load values into account. This extension was

originally suggested by Rudell [36] and implemented by us [42] using load discretization. We present a new method that relies on a functional representation of achievable arrival times at a node $v$ as a function of the load at the output of $v$. In section 2.4 we review three main factors that limits the optimality of tree covering for delay, and discuss how these limitations could be overcome. In section 2.5 we present several techniques that can be used to reduce the area of tree cover under a delay constraint. One of these techniques consists in extending tree covering further to directly minimize area under a delay constraint, at the expense of more computation time. We present an adaptive time discretization algorithm to perform this task. This algorithm was already described in [36, 42]. Finally we present and discuss our experimental results in section 2.6 and summarize the main results of this chapter in section 2.7.

## 2.2 Tree Covering

The use of tree covering algorithms for technology mapping was originally proposed by Keutzer [24]. The basis of this technique is to decompose the circuit into trees, and use tree covering on each of the trees separately. Tree covering algorithms were initially developed for code generation [2, 1]. They are based on fast tree pattern matching techniques, and use dynamic programming to implicitly enumerate all solutions efficiently. Tree covering works very well for additive cost functions, and more generally whenever the minimum cost cover of a tree rooted at a node is only function of the cost of the matches at that node and the cost of the subtrees. This is the case in code generation and logic synthesis for minimum area when we ignore non linear effects (pipelining, caching in code generation; placement and routing in logic synthesis).

The tree covering problem can be described as follows:

- **Given** a tree $T$ and a set of tree patterns $P$, representing the gates of a library, and a cost associated with each gate,
- **Find** a cover of the tree $T$ of minimum cost.

To allow the use of tree pattern matching algorithms, we need to represent both the tree and the logic functions associated with the gates in a common set of primitives or base functions. In misII and DAGON, this set of primitives is composed of only two types of elements: 2-input NAND gates and inverters. misII also adds extraneous inverters to

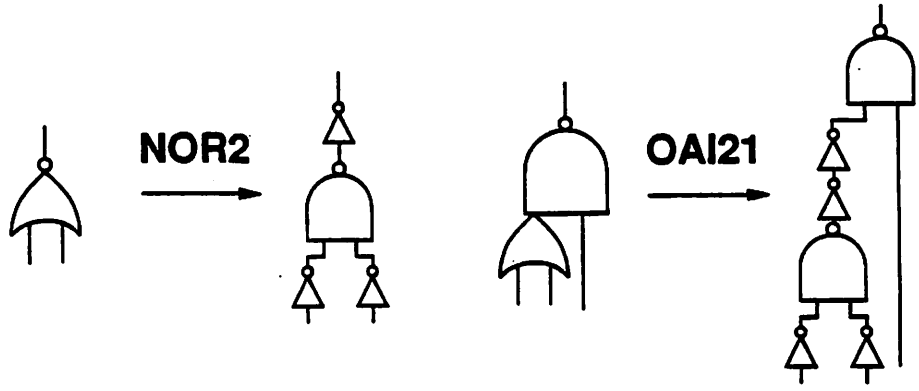


Figure 2.2: Example of Decomposition into Primitive Gates

allow the pattern matching algorithm to choose in which phase subfunctions should be implemented. An example of such a decomposition for a NOR gate and a AOI21 gate is given in figure 2.2. The network is decomposed in a similar fashion into primitive gates.

We suppose that we have at our disposal an algorithm to enumerate all matching tree patterns $m$ at a given node $v$ of tree $T$. We call this set $match(v, P)$. Associated with a given pattern $m$ in $match(v, P)$, we have a gate $g(m)$ and a list of nodes $v_i \in in(m)$, which correspond to the nodes of $T$ matching the inputs of $m$. An example of such a matching is given in figure 2.3.

In general, the cost of a match depends on some contextual information that is function of both the children and the parent nodes of given node $v$ of the tree. While for area minimization the cost of a match only depends on the children nodes, for delay minimization, or area minimization under a delay constraint, the cost function also depends on information provided by the parent node (load values, required times). We present here a general formulation of the cost function that cover all three types of optimization.

To formulate this cost function, we use a generic variable $I$ to denote the contextual information derived from the parent nodes. The minimum cost tree cover at a node $v$, $cost(v, I)$, is the minimum cost of a cover of the subtree rooted at $v$ for a given $I$.

To evaluate the cost of a pattern $m$ in $match(v, P)$ we need a cost function $cost(m, I)$ that depends on the cost of the gate $g(m)$ associated with the pattern and the costs $cost(v_{m,i}, I_{m,i})$, where the nodes $v_{m,i} \in inputs(m)$ are the nodes in the tree matching

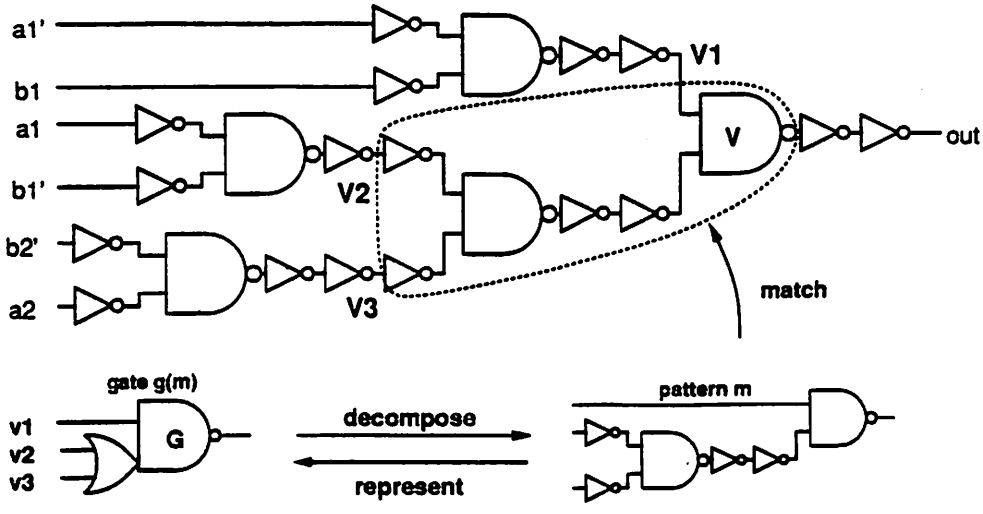


Figure 2.3: Example of Pattern Matching

the inputs of $m$, and $I_{m,i}$ is the contextual information derived from $I$ that is propagated through $m$ to node $v_{i,m}$. For example, $I_{m,i}$ may be the load of gate $g(m)$ at input pin $i$, in which case it is independent of $I$; or it can be the required time at $v_{i,m}$ derived from the required time at $v$.

To be able to use a dynamic formulation of the minimum cost covering problem we suppose in addition that the function $cost(m, I) = cost(g(m), [cost(v_{m,i}, I_{m,i}), v_{m,i} \in inputs(m)]$ is monotonic non-decreasing. We can thus use the principle of optimality to assert that there exists a minimum cost cover at a node $v$ which is made of a gate $g$ matching at node $v$ and minimum cost covers of the subtrees rooted at the input pins of $g$. It is easy to see by induction that in one bottom-up traversal of the tree, from the leaves to the root, an algorithm can obtain a minimum cost cover. A sketch of this algorithm is given in Figure 2.4, where the function $select(v, I)$ records the best pattern matching at node $v$ as a function of the contextual parameter $I$.

This formulation relies on our ability to manipulate functions of $I$ as data and in particular to compute the minimum of two such functions. In the case of area minimization, there is no contextual information to propagate: $area(m) = cost(m, I)$ and $area(v) =$

```
procedure min_cost(v, P)
    for w in children(v) do min_cost(w, P) od;
    for m in match(v, P) do
        cost(m, I) = cost(g(m), [cost(v_{i,m}, I_{i,m}), v_{i,m} ∈ inputs(m)]);
    od;
    cost(v, I) := min_{m∈match(v,P)} cost(m, I);
    select(v, I) := arg min_{m∈match(v,P)} cost(m, I)
end min_cost
```

Figure 2.4: Tree Covering for Minimum Cost

$cost(v, I)$ are simply numbers. More specifically:

$$area(m) = area(g(m)) + \sum_{v_{i,m} \in inputs(m)} area(v_{i,m}) \quad (2.1)$$

In the case of delay minimization, if we ignore the differences between loads and use a nominal load value $\gamma_0$ at the output of each gate, there is no contextual information to propagate either. In that case the cost function $arrival(m) = cost(m, I)$ becomes:

$$arrival(m) = \max_{v_{i,m} \in inputs(m)} (\alpha_{i,g(m)} + \beta_{i,g(m)}\gamma_0 + arrival(v_{i,m})) \quad (2.2)$$

However if we want to take actual load values into account, $cost(m, I)$ and $cost(v, I)$ are non constant functions of $I$. We will see next how they can be computed efficiently.

## 2.3 Handling of Load Values

To compute an exact minimum delay cover we need to take load values into account. Load information is propagated top down, from the root to the leaves of the tree. Therefore load values have to be represented as contextual information. In that case the cost function depends on the load $\gamma$ at the output of a node:

$$arrival(m, \gamma) = \max_{v_{i,m} \in inputs(m)} (\alpha_{i,g(m)} + \beta_{i,g(m)}\gamma + arrival(v_{i,m}, \gamma_{i,m})) \quad (2.3)$$

where $\gamma_{i,m}$ denotes the load on pin $i$ of gate $g(m)$.

Cost functions are now dependent on the load $\gamma$. We need to find a representation of these functions that allow us to compute their minimum as in Figure 2.4. We propose here three different representations. The first two use staircase functions, the third piece-wise linear functions.

### 2.3.1 Uniform Discretization

The simplest non constant representation of the cost functions is to use staircase functions, where the limits of the intervals are fixed and independent of the nodes. This is simple to implement but relatively inaccurate unless the discretization intervals are made fairly small, which is computationally expensive. This method was originally suggested by Rudell [36].

### 2.3.2 Adaptive Discretization

A better approach is to adapt the discretization intervals to each node. In one precomputation phase, one can determine all the possible load values at a node by examining all matches at every node. These values can then be used to determine the boundaries of discretization intervals. Since it is guaranteed that no other value is possible at an internal node, this method is exact. If the number of intervals grow too large, they can be easily reduced by merging the smaller intervals with their neighbors, for a small expected reduction in precision. This method was used in [42].

### 2.3.3 Functional Abstraction

Adaptive discretization has two drawbacks: it requires the precomputation of all possible load values at each node, which is roughly as time consuming as performing the tree covering itself; it does not work at the root of the tree, where the number of all possible load values grows exponentially and the range grows linearly with the number of fanouts of the root node.

A more general and more elegant method consists in going up one level of abstraction, by storing at each node a *function* that gives, for any given load value, the optimum choice of a gate at this node. The main difficulty is to find a data structure adapted to the representation of such a function. We first list which operations are needed on this data

structure, and then show these operations can be efficiently implemented using piece-wise linear functions.

We have to be able to perform efficiently the following operations:

- *representation of gate delays*: given a gate $g$, matching at node $v$, and arrival times $a_i$ at the inputs of the gate, we should be able to compute a function $f(\gamma)$, where $\gamma$ is the load at the output of node $v$, that gives the arrival time at the output of $g$ when $g$ has to drive a load of $\gamma$.

- *computation of the minimum of two functions*: given two functions $f$ and $g$ of one variable $\gamma$ representing the load at the output of a node $v$, we need to be able to compute the function $h(\gamma) = \min(f(\gamma), g(\gamma))$.

- *finding the minimum solution*: in addition, $h = \min(f, g)$ should be represented in such a way that we can tell, for a given value of $\gamma$, whether the minimum is realized by $f$ or $g$. If $h$ represents the minimum of all the functions representing the arrival times for all gates matching at a node $v$, we should be able to determine from $h$ not only the best arrival time, but also a gate that realizes the best arrival time.

Given these three operations, we can obtain a minimum delay tree cover by applying directly the algorithm of Figure 2.4. In the remaining of this section, we show how these three operations can be implemented efficiently using piece-wise linear functions within our delay model. For more complex delay models, piece-wise linear functions may not suffice. In that case, a more complex data structure would need to be used.

**Piece-Wise Linear Functions** A piece-wise linear function is a continuous function, formed of a finite sequence of connected, linear segments. We only consider piece-wise linear functions defined on $\mathcal{R}_+$, so we can always suppose that the support of each segment is of the form $(p_1, p2)$, with $p_1 \in \mathcal{R}_+$ and $p_2 \in \mathcal{R}_+ \cup \{+\infty\}$. We represent each segment by a tuple of the form: $(x_i, y_i, s_i, p_i)$, where $(x_i, y_i)$ are the coordinates of the leftmost point of the segment, $s_i$ is the slope of the segment, and $p_i$ is a pointer to an object that will be specified later. Strictly speaking, this representation is incomplete, because it does not specify the right bound of the segment, but it is not intended to be used in isolation.

We represent a piece-wise linear function $f$ as an array of tuples representing the connected segments forming $f$: $(x_i, y_i, s_i, p_i)_{1 \leq i \leq n}$, such that $0 = x_1 < x_2 < \ldots < x_n$, and,

for $1 \leq i \leq n-1$, $y_{i+1} = y_i + (x_{i+1} - x_i)s_i$. All segments but the last one are finite: the right bound of the $i^{th}$ segment for $i < n$ is given by $x_{i+1}$.

**Representation of the Arrival Time Function at the Output of a Gate** Given arrival times $(a_i)_{1 \leq i \leq n}$ at the input pins of a gate $g$, the arrival time at the output of $g$ as a function of the load at the output of $g$ is given, in our delay model, as the maximum of $n$ linear functions, one per input pin. This maximum is a piece-wise linear function, which is represented, as indicated in the previous paragraph, by a finite sequence of tuples: $(x_n, y_n, s_n, p)_{1 \leq n \leq N}$. The pointer $p$ is independent of $n$ and points to a record containing information pertaining to gate $g$. This information is needed to retrieve a description of gate $g$ in case the selection of $g$ yields the earliest arrival time.

**Computation of the Minimum of Two Piece-Wise Linear Functions** The minimum of two piece-wise linear functions $f_1$ and $f_2$ can be computed in time $O(n_1 + n_2)$ where $n_i$ is the number of segments contained in $f_i$. The algorithm bears some similarity with a well-known linear time algorithm used for merging two sorted lists of data.

The algorithm scans implicitly all values of $\mathcal{R}_+$ from 0 to $+\infty$, and keeps track of the best segment at each point. In practice the number of points that need to be visited does not exceed the total number of segments, $n_1 + n_2$. The algorithm maintains two indices $i_1$ and $i_2$ pointing to the currently active segments, and a scan point $x$, initially set to 0. As an invariant of the algorithm, $x$ is guaranteed to lie within segment $i_1$ of $f_1$ and segment $i_2$ of $f_2$. The next value of $x$ is the leftmost of the following three points: the rightmost limit of segment $i_1$ of $f_1$, the rightmost limit of segment $i_2$ of $f_2$ and the intersection of $i_1$ and $i_2$, provided that this intersection lies to the right of the current value of $x$. The current value of $x$ and the next value of $x$ determine a segment. This segment is a copy of $i_1$ of $f_1$ or $i_2$ of $f_2$ on that range, whichever is the lower on that range of values. $x$ is then updated to its new value, and $i_1$ or $i_2$ is incremented as necessary. The algorithm terminates when all segments have been visited.

**Finding the Optimum Solution** By computing the minimum of all the functions representing the arrival times for all gates matching at a node $v$, we obtain a piece-wise linear function $f$ representing the best arrival times realizable at $v$ for any load value. For a given load value $\gamma$, we simply perform a binary search on the array of tuples $(x_i, y_i, s_i, p_i)$

representing $f$ to identify the segment to which $\gamma$ belongs. Once this segment is identified, we use the pointer $p_i$ to retrieve the gate that realizes the minimum.

The actual implementation is slightly more complex than what we have just described, because a gate may match at a node in several different ways, and we need to keep track with $p_i$ not only of a choice of a gate but also of a choice of a matching.

## 2.4 Limits to the Optimality of Tree Covering for Delay

### 2.4.1 Initial Decomposition

The initial decomposition of a tree in simple primitives (i.e. 2-input NAND gates and inverters) is required before tree covering can be used. The principle of tree covering is to decompose the target tree and gates into the same set of primitives so that functional matching is replaced by the simpler problem of pattern matching. Since gates represent small logic functions, it is feasible to enumerate all patterns that represent a gate. This number can be reduced further by exploiting the symmetry of some inputs.

Unfortunately, for the tree itself, it is not practical in general to enumerate all possible decomposition in simple primitives. We choose one decomposition, and the result depends in general on the quality of this decomposition. Ideally, an adequate choice should be made as a function of the arrival times at the leaves of the tree. In the absence of this information, we simply generate a balanced decomposition of the nodes of the tree. This is not in general the best choice. Before we can discuss better decompositions, we need to cover the material of chapter 4. We will come back to this problem in chapter 5.

### 2.4.2 Suboptimality of Pin Assignment

During tree matching, the symmetry of gate inputs is exploited to reduce the computational overhead of the algorithm. All patterns that a gate can match are enumerated, but all possible assignments of gate inputs to pattern inputs are not. When the inputs are equivalent, only one assignment is tried. For area minimization, logically equivalent pin assignments have all the same cost, so there is no need to consider more than one. However, for delay minimization, this is not the case, since the delay through a gate is pin dependent in general.

For example, a 3-input NAND gate is represented by only one pattern, but has

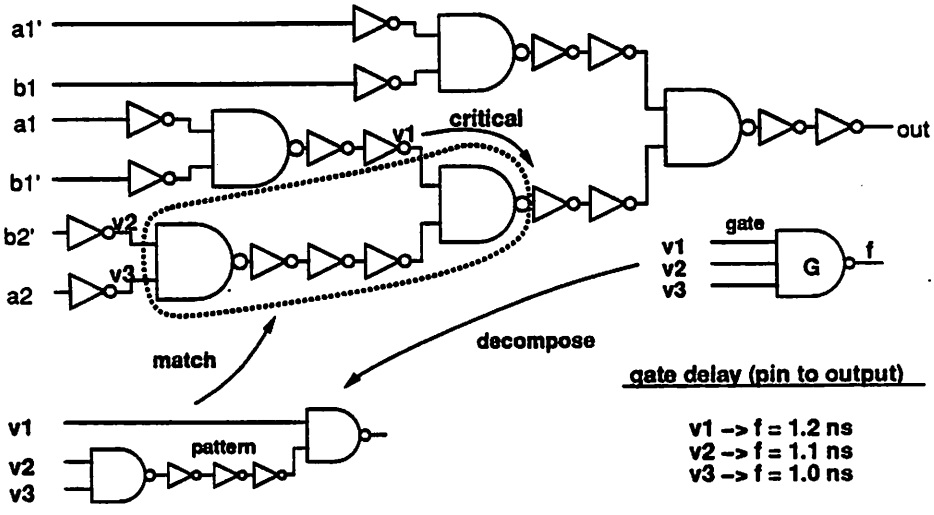


Figure 2.5: Example of Suboptimal Pin Assignment

$3! = 6$ possible pin assignments. Only one out of the six possible pin assignments is tried by the tree pattern matching algorithm. This can lead to suboptimal choices. In the example of Figure 2.5, the critical input is assigned the slowest pin $v_1$. For any preassigned choice of input ordering, there is always a simple circuit configuration for which this choice is suboptimal. We propose in this section a simple algorithm to correct this problem.

To optimize pin assignment during tree covering, we can proceed as follows. First, in a preprocessing step that can be done once and for all on the library, we identify the sets of pins that are functionally equivalent and therefore interchangeable. Then, for each gate during pattern matching, we consider each of the equivalent sets of pins separately and reorder them in order to decrease delay.

If we use constant load values, the problem of optimal pin assignment can be solved in time $O(n \log n)$ where $n$ is the number of pins in the set. In that case the problem can be reduced to the following discrete optimization problem: if $a_i$ is the arrival time at node $v_i$ and $d_j$ the delay through the gate from pin $j$, the problem is to find an assignment of nodes to pins that minimize the total delay. Let $\Sigma_n$ be the set of permutations of a set of $n$ elements, and for $\sigma \in \Sigma_n$, let $\sigma(i)$ be the image by the permutation $\sigma$ of element $i$. The pin

assignment problem can the be formulated as the following discrete optimization problem:

$$\min_{\sigma \in \Sigma_n} \max_{1 \leq i \leq n} a_i + d_{\sigma(i)}$$

An optimal solution can be computed by ordering the $a_i$ and the $d_j$ by increasing size and using the permutation $\sigma(i) = n - i + 1$. However, if we take pin load values into account, the problem has the following form, where $a_i$ denotes the arrival time at node $v_i$ and $\alpha_i$, $\beta_i$ and $\gamma_i$ are delay coefficients derived from the delay model:

$$\min_{\sigma \in \Sigma_n} \max_{1 \leq i \leq n} (a_i + \beta_i * \gamma_{\sigma(i)} + \alpha_{\sigma(i)})$$

This problem can still be solved optimally by dynamic programming in time $O(2^n)$. Though exponential, this algorithm is still practical for most libraries since the number of equivalent pins of any given gate usually does not exceed eight in CMOS technology.

### 2.4.3 Rise and Fall Delays

So far we have ignored the fact that in our delay model we distinguish between rise and fall delays. This means that arrival times are characterized by a pair of real numbers: $(a_r, a_f)$ instead of a single number. To decide which of two solutions is better, we need to decide which of two pairs of arrival times is "faster". We use the following criterion to make this decision:

$$(a_r, a_f) < (b_r, b_f) \quad if \quad \max(a_r, a_f) < \max(b_r, b_f)$$

This selection is not guaranteed to be optimal in general but works well in practice. It outperforms the other obvious choice:

$$(a_r, a_f) < (b_r, b_f) \quad if \quad \frac{a_r + a_f}{2} < \frac{b_r + b_f}{2}$$

## 2.5 Minimizing Area under a Delay Constraint

We can also use the dynamic programming algorithm of Figure 2.4 to find a minimum area cover under a delay constraint. The delay constraint is expressed as a required time at the root of the tree, and is propagated down the tree as a contextual value, together with load values. In that case the cost function is of the form $cost(m, \gamma, r)$ where $\gamma$

represents the load and $r$ the required time at the output of a node:

$$\begin{aligned} cost(m,\gamma,r) &= \{area(m), slack(m,\gamma,r)\} \\ area(m) &= area(g(m)) + \sum_{v_{i,m} \in inputs(m)} area(v_{i,m}) \\ slack(m,\gamma,r) &= r - arrival(m,\gamma,r) \\ arrival(m,\gamma,r) &= \max_{v_{i,m} \in inputs(m)} (\alpha_{i,g(m)} + \beta_{i,g(m)}\gamma + arrival(v_{i,m}, \gamma_{i,m}, r_{i,m})) \end{aligned}$$

where $r_{i,m}$ is the required time at input node $v_{i,m}$ propagated from required time $r$ at node $v$ through gate $g(m)$. To minimize area under a delay constraint, we take into account in the cost function both the area and the *slack* (the slack is the difference between required time and arrival time). At each intermediate node the minimum area solution is selected if it meets the delay requirement (that is, if its slack is non negative); otherwise, the minimum delay solution is chosen.

### 2.5.1 Adaptive Discretization of Required Times

To implement this algorithm, we need to discretize the required times. Enumerating all possible required times at a node is not feasible in this case, because required times not only depend on the match just above a node but on all the possible combinations of matches above the node up to the root of the tree.

To control the run time of the algorithm, we enforce a limit on the number of discretization intervals at each node, which is an integer-valued parameter $\tau$ specified by the user. The discretization intervals are obtained by first computing a range of interesting values for the required time at each node and then dividing this range into equal intervals.

The range of interesting values for the required time at a node $v$ is determined as follows. Let $\gamma$ be a possible load value at node $v$. Let $a^{\gamma}_{delay}(v)$ be the minimum arrival time achievable at node $v$ with a load of $\gamma$ at its output, and $a^{\gamma}_{area}(v)$ the arrival time at $v$ of a minimum area cover of the subtree rooted at $v$ for that same load value. Any required time outside the interval $[a^{\gamma}_{delay}(v), a^{\gamma}_{area}(v)]$ does not need to be considered. Indeed, if the required time at node $v$ is less than $a^{\gamma}_{delay}(v)$, no cover of the subtree rooted at $v$ can meet the timing requirement; in that case, the minimum cost cover is the minimum delay cover for this subtree. If the required time is greater than $a^{\gamma}_{area}(v)$, we can just choose the minimum area cover for this subtree.

### 2.5.2 A Greedy Approach to Area Recovery

If we neglect the inaccuracies introduced by the discretization of required times, the previous algorithm is optimum, though relatively complex. A simpler approach to area recovery consists in computing at each node the minimum delay and minimum area solution. To select the cover, we can then use a simple top down traversal of the tree starting from the root. Each time a gate is selected, we propagate the required time through its pins. The required time at the root of the tree is simply the minimum achievable arrival time. To select a cover for a subtree, we choose the minimum delay solution unless the minimum area solution is fast enough. This approach knows no compromise: it never chooses an intermediate solution in terms of area that would be acceptable in terms of delay; but is fast and simple.

### 2.5.3 Area Recovery by Optimal Inverter Selection

Another effective approach to reducing the complexity of area recovery is to concentrate on special cases. The most obvious candidates are inverters. Inverters are the most frequently used gates in circuits, and selecting the best inverter to minimize delay at any given point in the network can be done very simply by enumerating all choices and selecting the best. There is a simple and optimal algorithm that applies inverter selection to minimize delay through a *mapped network*. A mapped network is a Boolean network where each node represents a gate.

The algorithm proceeds as follows: it visits the nodes of the network in topological order from the root to the leaves. Each time an inverter is encountered, it is replaced by an inverter that has minimum delay in this context. The optimal choice depends on the drive at the input and the load at the output of the inverter. If applied to a network obtained from a minimum delay cover, this optimization has no effect. However it can be used to speed up networks obtained from the minimum delay tree covering algorithm that assumes that all load values are identical.

More importantly, inverter selection can be used for area recovery. Given a mapped network, with a required time at the root (either specified by the user, or taken as being equal to the arrival time at the root of the tree), we can traverse the network from the root to the leaves and apply the following optimization each time an inverter is encountered. If there is a smaller inverter that could be used at the node without making the slack negative

at that node, the smallest such inverter is used in place of the current one. This algorithm is greedy in the sense that it does not necessarily make the best use of the available slack to recover area. However it is guaranteed to never worsen delay, and, as we will see in the results section, is quite effective in practice.

### 2.5.4 Area Recovery by Optimal Gate Selection

There is no reason to perform optimal gate selection on inverters only. The inverter selection algorithm can be extended to all gates that come in different sizes and strengths in the library. This provides a cheap and simple way to recover area after tree covering without hampering the full power of tree covering for delay minimization. This optimization is likely to be very effective for large libraries.

## 2.6 Experimental Results

Circuits are usually not trees: they have several outputs, inputs are shared among several functions, and there may be several paths from one node to another. To assess the amount of improvement we can expect from the algorithms proposed in this chapter, we need to measure their effect in isolation on trees. To that effect, we generated three Boolean functions, `nor32`, `balanced` and `unbalanced`, that can be represented as trees:

- `nor32`, a 32 input NOR gate.
- `balanced`, a balanced binary tree with 64 inputs. Internal nodes are alternatively computing a Boolean AND or a Boolean OR, with inverters inserted randomly.
- `unbalanced`, an unbalanced binary tree with 32 inputs, where every node has at most one child that is not a leaf. Internal nodes are alternatively computing a Boolean AND or a Boolean OR, with inverters inserted randomly.

The results of these experiments are also sensitive to the gate libraries being used. To take this effect into account, we performed our experiments with four different CMOS standard cell libraries of various origins:

- `MCNC`, a public domain library available from MCNC. It is distributed with the IWLS'89 benchmark suite [32] under the name `lib2`. It is composed of 29 gates. Inverters ap-

pear in 3 different strengths; all the other gates in one strength only. Gate delay information is pin dependent.

- CMOS12 is a library from AT&T. It is composed of 189 gates, mostly AOI and OAI gates. Only a few gates appear in different strengths. Gate delays are only available for the slowest pins; all pins are assigned the worst case delay, which is too conservative.
- LIBRARY3 is an industrial library. It is composed of 80 gates. Most gates come in different strengths, and delay information is pin dependent. The library contains 4 different sizes of inverters.
- LIBRARY4 is another industrial library. It is composed of 99 gates. Like the previous library, most gates come in different strengths and delay information is pin dependent. It provides a wider selection of strengths for commonly used gates than LIBRARY3. For example, it contains 9 different sizes of inverters instead of 4, and 4 different sizes of 2-input NAND gates instead of 2.

In the following sections, we report and discuss results by library. We provide detailed experimental results for the first two libraries, and only summary information for the remaining two. For clarity, we use the following acronyms to refer to the various tree covering algorithms studied in these experiments:

- MA refers to minimum area tree covering.
- MDCL refers to minimum delay tree covering using a constant load value.
- MDCLIS refers to minimum delay tree covering using a constant load value, followed by optimum inverter selection. Inverter selection is done using the algorithm of section 2.5.
- MDEL refers to minimum delay tree covering using exact load values.
- MDELIS refers to minimum delay tree covering using exact load values, followed by optimum inverter selection.
- MADC refers to minimum area tree covering under a delay constraint.

Since MDCL is not optimum for delay, MDCLIS can outperform MDCL both in terms of area and delay. On the other hand, MDEL being optimum for delay, MDELIS can only improve over MDEL in terms of area.

| circuit | MA | | MDCL | | MDEL | | MADC | |
|---|---|---|---|---|---|---|---|---|
| | *area* | *delay* | *area* | *delay* | *area* | *delay* | *area* | *delay* |
| nor32 | 53 | 5.14 | 93 | 5.31 | 73 | 5.12 | 53 | 5.14 |
| balanced | 133 | 6.15 | 177 | 4.71 | 161 | 4.69 | 167 | 4.69 |
| unbalanced | 75 | 20.70 | 111 | 16.32 | 95 | 16.31 | 106 | 16.61 |

Table 2.1: Comparison of Tree Covering Algorithms with library MCNC

MA: Minimum Area
MDCL: Minimum Delay with Constant Loads
MDEL: Minimum Delay with Exact Loads
MADC: Minimum Area under a Delay Constraint
*area* total cell area
*delay* measured in nanoseconds

### 2.6.1 Results with the MCNC Library

In table 2.1 we show the results obtained by using minimum area tree covering, minimum delay tree covering, minimum delay tree covering with constant loads and tree covering for minimum area under a delay constraint, where the delay constraint is the minimum delay achievable by tree covering.

The results indicate that MDEL tree covering does not lead to a significant delay reduction over MDCL for this library: only 1%. However, the reduction in area is more substantial: 15%. This can be explained by the fact that using constant load values in MDCL underestimates the cost of stronger and larger gates, that have higher input loads. The penalty of using gates stronger than the optimum has a first order effect in terms of area. In terms of delay, a higher input load is partially compensated by a stronger drive capability, leading to a second order effect on delay. MADC tree covering does not lead to consistent results. This is due to the fact that MADC relies on an older implementation than MDEL, that discretizes load values instead of using piece-wise linear functions. Discretization of arrival times is another source of inaccuracy. Overall MADC reduces area by 6% and increases delay by 1% relative to MDEL.

Table 2.2 shows the effect of optimal inverter selection when used after tree covering. Inverter selection has no effect on trees built with MDEL. However it improves noticeably the quality of the covers obtained with MDCL. The benefit of MDELIS over MDCLIS is only of 7% in area for no gain in delay, down from 15% and 1% respectively for MDEL vs. MDCL.

| circuit | MDCLIS | | MDELIS | |
|---|---|---|---|---|
| | *area* | *delay* | *area* | *delay* |
| nor32 | 93 | 5.31 | 73 | 5.12 |
| balanced | 153 | 4.52 | 161 | 4.69 |
| unbalanced | 96 | 16.32 | 95 | 16.31 |

Table 2.2: Effect of Optimal Inverter Selection with library MCNC

MDCLIS: Minimum Delay with Constant Loads, with optimum Inverter Selection
MDELIS: Minimum Delay with Exact Loads, with optimum Inverter Selection
*area* total cell area
*delay* measured in nanoseconds

| circuit | MA | | MDCL | | MDEL | | MADC | |
|---|---|---|---|---|---|---|---|---|
| | *area* | *delay* | *area* | *delay* | *area* | *delay* | *area* | *delay* |
| nor32 | 53 | 4.57 | 189 | 3.79 | 234 | 3.70 | 174 | 4.00 |
| balanced | 142 | 6.92 | 574 | 4.72 | 446 | 4.46 | 478 | 4.46 |
| unbalanced | 75 | 26.10 | 351 | 17.90 | 241 | 17.36 | 175 | 17.89 |

Table 2.3: Comparison of Tree Mapping Algorithms with library CMOS12

MA: Minimum Area
MDCL: Minimum Delay with Constant Loads
MDEL: Minimum Delay with Exact Loads
MADC: Minimum Area under a Delay Constraint
*area* total cell area
*delay* measured in nanoseconds

For one example, MDCLIS actually outperforms MDELIS in terms of delay. This anomaly can be explained by the difficulty on handling rise and fall delays optimally, as explained in section 2.4.3. The anomaly disappears if we modify the library to make all rise and fall delays equal, or if we modify the comparison function used to compare pairs of rise and fall arrival times.

### 2.6.2 Results with the CMOS12 Library

We repeated the previous experiments, using the CMOS12 library. The results are reported in table 2.3 and table 2.4. The results are essentially similar to those reported in the previous section. MDCL tree covering increased area by 355% for a decrease in delay of

| circuit | MDCLIS | | MDELIS | |
|---|---|---|---|---|
| | *area* | *delay* | *area* | *delay* |
| nor32 | 189 | 3.79 | 234 | 3.70 |
| balanced | 430 | 4.46 | 430 | 4.46 |
| unbalanced | 189 | 17.70 | 193 | 17.36 |

Table 2.4: Effect of Optimal Inverter Selection with library CMOS12

MDCLIS: Minimum Delay with Constant Loads, with optimum Inverter Selection
MDELIS: Minimum Delay with Exact Loads, with optimum Inverter Selection
*area* total cell area
*delay* measured in nanoseconds

30% over MA. On average, MDEL tree covering outperformed MDCL by 13% in area and 4% in delay. Again, using exact load values during delay minimization had some effect on area but only a second order effect on delay. MADC tree covering obtained more satisfactory results with this library. Compared with MDEL tree covering, it achieved a reduction of area of 17% for an increase in delay of 4%.

For this library, inverter selection also improves the quality of MDEL tree covers in terms of area, by 8%. The effect of inverter selection on MDCL tree covers is more significant, to the point that MDCLIS tree covering actually outperforms MDELIS in terms of area by 8% for a cost in delay of only 1%.

### 2.6.3 Results with the LIBRARY3 Library

We repeated the same experiments with library LIBRARY3. MDEL tree covering outperformed MDCL by 10% in area and 1% in delay. After inverter selection, the advantage was only of 4% in area and 1% in delay for MDELIS over MDCLIS. Inverter selection reduced the area of MDEL covers by 7%.

### 2.6.4 Results with the LIBRARY4 Library

We repeated the same experiments with library LIBRARY4. For that library, which is much richer than LIBRARY3 in terms of number of inverters, MDEL tree covering outperformed MDCL by 56% in area and 24% in delay. After inverter selection, the advantage was reduced to 21% in area and 8% in delay for MDELIS over MDCLIS. Inverter selection improved

the area of MDEL covers by only 3%.

## 2.7 Conclusion

The main conclusion from this experiments is somewhat disappointing: taking load values into account during minimum delay tree covering (method MDEL) does not lead to a very significant decrease in delay over the simpler tree covering method that uses constant loads (method MDCL). The main advantage of MDEL over MDCL is actually more in terms of area. By ignoring the effect of larger input loads, MDCL tend to favor gates that are larger than necessary. Choosing larger gates than necessary has a direct effect on area but only a second order effect on delay, since the cost of higher input loads is partially compensated by an increase in drive capability.

Another interesting experimental result is that most of the advantage of using MDEL over MDCL can be obtained by the use of optimal inverter selection after tree covering. Overall, MDEL remains the best tree covering algorithm for delay, but using MDCL followed by optimal inverter selection is a very attractive choice if simplicity of implementation and cpu time are an issue. It will be interesting to see the effect of extending optimal inverter selection to optimal gate sizing as a postprocessing phase after tree covering.

Minimizing area under a delay constraint is not very effective and is computationally expensive. We strongly recommend a divide and conquer approach to this problem: first, use a fast tree covering algorithm that minimizes delay only, to obtain a minimum delay solution. Then, use, in a postprocessing phase, either an inverter selection algorithm or a gate sizing algorithm to recover area at no delay cost. This approach of area recovery is suboptimal, but leads rapidly to good quality circuits. If area is more of a concern, it is always possible to use minimum area tree covering, possibly as a postprocessing phase after delay optimization, on parts of the tree that are not time critical. Trying to find a good cover under an area and a delay constraint is too complex and time consuming: it is more efficient to start with a minimum delay cover, and modify it in a postprocessing optimization phase to recover area in a greedy fashion whenever possible.

# Chapter 3

# Fanout Optimization

Mathematicians are like Frenchmen:
whatever you say to them they translate into their own language
and forthwith it is something entirely different.
— GOETHE

## 3.1 Introduction

The objective of *fanout optimization* is to build circuits that do not compute any function but simply distribute a signal to one or more destinations at a minimum cost. If the cost to minimize is area, the problem is of very little interest at the logic design level, since a wire is the best we can do. The problem is only interesting if we are to minimize delay or area under a delay constraint. The focus of this chapter is to find methods to perform this optimization efficiently.

Fanout optimization is important for several reasons. First, it can reduce delay often quite dramatically. If the output of a gate is connected to $n$ fanout stems, in first approximation the delay through the gate is of order $O(n)$. By building a simple buffer tree at the output of the gate, we can reduce this delay to $O(\log n)$. In addition, if we know the individual times at which the signals are required at each destination, we can build a buffer tree that delivers earlier the early required signals, which has the potential of further decreasing the delay through the entire circuit. Ideally, a fanout algorithm should be able to take advantage of the slack available at some outputs to increase the slack at the initially more critical outputs, to achieve an equilibrium point where all outputs are equally critical. This is usually not achieved in practice due to the discrete nature of delay optimization at

this level.

The basic optimization techniques on which fanout optimization relies, buffering, gate resizing, critical signal isolation, are not new. There is a vast literature on timing optimization techniques that covers these optimizations [34, 17, 21, 4, 13]. What is original in the fanout optimization approach originally due to Berman, Carter and Day [5], is the idea of combining these techniques into a single algorithm. The main limitation of Berman's work is that it did not propose a very practical approach to apply a fanout algorithm to an entire circuit. It turns out that we can use a simple technique due to Hoover, Klawe and Pippenger [22] to solve this problem, as suggested by Fishburn [15]. We have extended this technique to recover area after delay minimization.

Fanout optimization can also be used to enforce fanout constraints imposed by a technology. Though in this work we ignore fanout constraints or load limitations, they can be handled by a simple modification of our algorithms, for example by modifying the cost functions to make gates infinitely slow as soon as their load constraints are violated. Another reason for using fanout optimization to enforce load limitations is to control the accuracy of gate-level delay models. The main source of inaccuracy of these models is the presence of large capacitive loads at gate outputs.

In some situations, fanout circuits with reconvergence can yield faster circuits than fanout trees, e.g. when the loads at the sinks are very high, such as for the output pads of a chip. These situations can be handled by tree-based fanout algorithms if we replace sinks with large loads by a number of virtual sinks with smaller loads before applying the algorithm. In this work we suppose that large loads are split among several virtual destinations or handled by special purpose circuitry, and we only consider fanout circuits that have no reconvergence, i.e. that are trees.

There is an interesting duality between tree covering and fanout optimization as can be seen in figure 3.1. For delay minimization, tree covering aims at minimizing the arrival time at the root of a tree given arrival times at its leaves, while fanout optimization aims at maximizing the required time at the root of a fanout tree given required times at its leaves. In terms of complexity, fanout optimization is, for all but the simplest delay models, NP-complete. But we can make, from an optimum fanout algorithm, a one pass algorithm that optimizes the fanout problems of a circuit and yields a minimum delay implementation, in the sense that there is no way to modify one or more of the fanout trees of this implementation in order to decrease delay. In contrast, tree covering itself is of linear

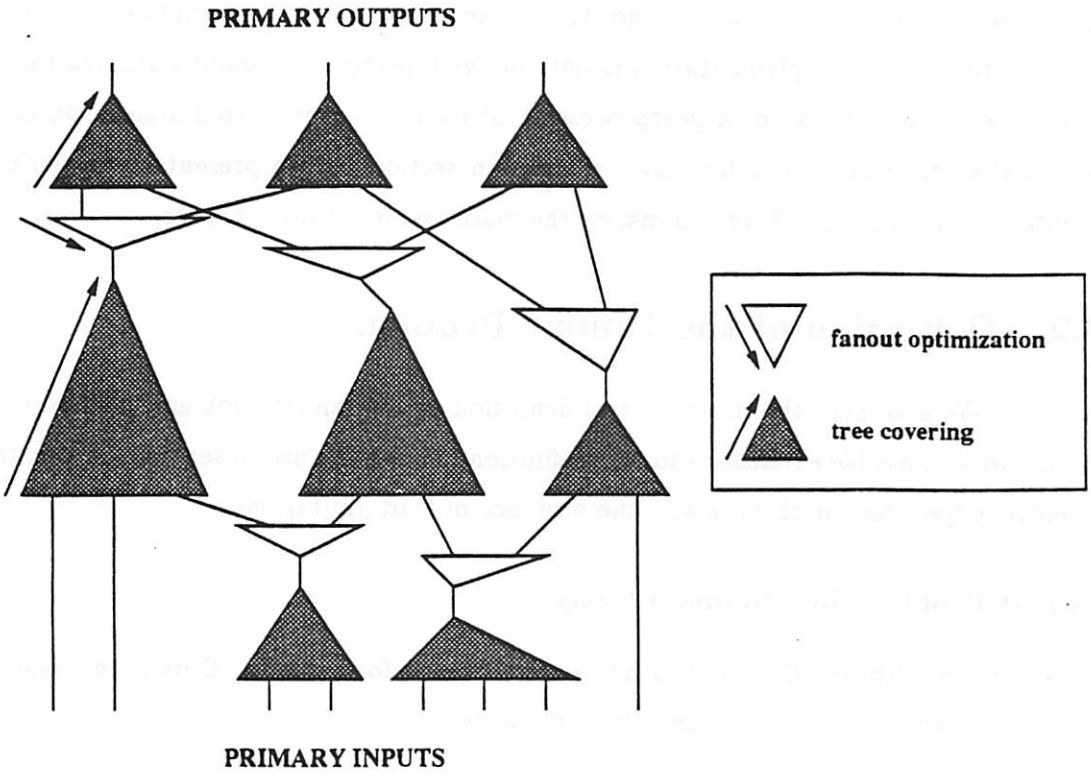


Figure 3.1: Duality of Tree Covering and Fanout Optimization

complexity, but extending it to a globally optimum algorithm is a difficult problem since it subsumes optimum DAG covering, which is NP-complete.

In section 3.2 we give a precise definition of the fanout problem and fix the terminology and notation for the rest of this chapter. In section 3.3 we give an overview of what is currently known about the complexity of the fanout problem, and how the delay model influences the complexity. In section 3.4 we present a spectrum of delay optimization algorithms for the fanout problem of increasing complexity and accuracy. In section 3.5 we describe succinctly what needs to be done to handle fanout problems where sinks are of different polarities. In section 3.6 we present a set of simple delay optimizations that can be used to improve the quality of an existing fanout tree. These optimizations take a narrow view on the problem to be optimized. By analogy to software compilation, we call them peephole optimizations. They are fast, simple to implement, effective at recovering

area, and can be applied to any fanout tree, independent of its origin. In section 3.7 we show how we can apply a fanout algorithm to an entire circuit and explain in what sense the resulting circuit implementation is optimal with respect to fanout optimization. This technique can also be used in postprocessing phase to recover wasted area in parts of the circuit that are not critical for delay. Finally in section 3.8 we present our experimental results, and in section 3.9 we summarize the main results of this chapter.

## 3.2 Definition of the Fanout Problem

We give here the most general definition of the fanout problem that we consider in our work. Possible extensions to this definition include the use of several sources, the use of reconvergent fanout circuits and the presence of load limitations.

**Fanout Problem for Minimum Delay**

- Given a library $\mathcal{L}$ of buffers and inverters, and for each $b \in \mathcal{L}$ its input load $\gamma_b$, its load dependent delay $\beta_b$ and its intrinsic delay $\alpha_b$;
- Given the *source* $s$ of a signal $X$, with its drive capability $\beta_s$;
- Given $n$ destinations or *sinks*, with separate *required times* $\tau_i$, loads $l_i$ and polarities $p_i$;
- Find a tree of buffers and inverters that distributes the signal $X$ to all the sinks and maximizes the required time at the source.

**Fanout Problem for Minimum Area under a Delay Constraint** We can extend this definition to the problem of finding a fanout tree of minimum area under a delay constraint. The delay constraint is specified as an arrival time $a$ specified at the source. The constraint is that the required time at the source should not be less than the specified arrival time.

## 3.3 Complexity of the Fanout Problem

The complexity of the fanout problem depends on the delay model. For a very simple delay model, under which the delay through a buffer is constant equal to 1 and the fanout is constrained to be less than some constant value $k$, the fanout problem for delay can

be solved in linear time using a technique called *combinational merging* [18]. Unfortunately, as soon as the delay model takes load values into account, even if all required times are equal and only one type of buffer is used, the fanout problem for delay becomes NP-complete. There is thus little hope of finding a polynomial time algorithm to solve the fanout problem optimally with delay models of the level of complexity of the ones commonly used for CMOS standard cells.

Berman *et al.* proved that the fanout problem for minimum area under a delay constraint is NP-complete under a simple delay model where gates are represented by a finite number of virtual gates with fixed fanout and constant delay. Unfortunately, the proof of this result is not very satisfactory since it requires the existence of buffers in an unrealistically wide range of sizes and delays.

In this section we present a few complexity results for the fanout problem under various delay models. To keep things simple, we ignore the issue of phase assignment: we suppose that all sinks require the signal with the same polarity as produced by the source and that only buffers are available in the library. For a simple delay model, the fanout problem can be solved optimally in time $O(n \log n)$ using combinational merging. We present this algorithm in section 3.3.1. We use combinational merging as a heuristic in one of our fanout optimization algorithms, but it is not optimum in general for more complex delay models. In section 3.3.2 we study the fanout problem for a slightly more complex delay model, for which we only have partial results. In section 3.3.3, we show that if we allow non constant load values at the sinks, the fanout problem for delay becomes NP-complete. This is the first complexity result for the fanout problem for minimum delay without the addition of an area constraint. Finally in section 3.3.4 we review briefly Berman's complexity result.

This overview is not complete unfortunately. Many complexity issues are left unresolved. However our main purpose is to provide solid evidence that the fanout problem is difficult to solve exactly for complex delay models, in order to justify our use of approximate algorithms. In that sense, our goal has been achieved.

### 3.3.1 Constant Delay Model

In the constant delay model, the library contains only one buffer. The delay through this buffer is constant equal to one delay unit when the fanout does not exceed some threshold $k$, and is infinite otherwise. For that delay model, the combinational merging

Let $S$ be a set of nodes with an individual required time associated with each node.
Initially $S$ contains all the sinks with their required times.
Sort $S$ by decreasing required times.
While $|S| > 1$ {
  At the first iteration, $r = |S| mod(k-1)$. If $r < 2$, $r = r + (k-1)$
  At all remaining iterations, $r = k$.
  Remove the first $r$ nodes of $S$, $(v_1, \ldots, v_r)$, with the largest required times.
  Make the $r$ nodes $(v_1, \ldots, v_r)$ the fanouts of a new node $v$.
  Set the required time of $v$ to be $\min_{1 \le i \le r} required(v_i) - 1$
  Add $v$ to $S$
}
The only remaining node in $S$ is the root of an optimal fanout tree.

Figure 3.2: Combinational Merging

algorithm, due to Golumbic [18], finds a minimum delay fanout tree in time $O(n \log n)$ where $n$ is the number of sinks. Moreover this tree is guaranteed to be of minimum area among all trees of finite delay. The algorithm is outlined in figure 3.2.

It is possible to prove that if $(r_1, \ldots, r_n)$ are the required times at the sinks, the required time at the root of an optimal fanout tree is given by the following formula [22]:

$$r_{root} = \left\lceil -\log_k\left(\sum_{1 \le i \le n} k^{-r_i}\right) \right\rceil \tag{3.1}$$

### 3.3.2 Unit Fanout Delay Model

In this section we introduce a slightly more realistic delay model. The library still only contains one buffer, but this time the delay through the buffer is equal to the number of its fanouts. This means that the load dependent delay coefficient of the buffer is 1, its intrinsic delay is 0, and the loads of all the gates 1. This model is similar to but simpler than the unit fanout delay model used in `misII`, which assumes a delay of $1 + 0.2 * n$ where $n$ is the number of fanouts of the buffer, i.e. it assumes an intrinsic delay of 1 and a load

dependent delay of 0.2 for a buffer.

**Equal Required Times** We can solve the fanout problem for minimum delay exactly for this delay model if all the required times at the sinks are equal. As we will see shortly, even this simple case is not completely straightforward. We will make use of the following definitions:

**Definition 1** *A* 2-3 tree *is a tree $T$ such that any intermediate node of $T$ has a fanout of 2 or 3.*

**Definition 2** *Let $\mathcal{P}(T)$ be the set of paths from the root to a leaf in a 2-3 tree $T$, and let $p$ be such a path. Let $x_p$ and $y_p$ be the number of nodes of fanout 2 and 3 respectively on that path. The weight of path $p$ is defined as follows:*

$$w(p) = 2^{x_p} \times 3^{y_p} \tag{3.2}$$

*The weight of a 2-3 tree is the maximum weight on any of its paths:*

$$w(T) = \max_{p \in \mathcal{P}(T)} w(p) \tag{3.3}$$

*The delay of a path and the delay through a tree are defined similarly:*

$$d(p) = 2\,x_p + 3\,y_p \tag{3.4}$$

$$d(T) = \max_{p \in \mathcal{P}(T)} d(p) \tag{3.5}$$

Since we suppose that all required times are equal, maximizing the required time at the source of the fanout tree is equivalent to minimizing the worst path delay through the tree, i.e. the quantity $d(T)$.

**Definition 3** *A 2-3 tree is a* simple 2-3 tree *if all nodes at the same level have the same fanout. In particular, in a simple 2-3 tree, all the leaves are at the same distance from the root.*

It is easy to see that all paths of a simple 2-3 tree have equal weight, and that the number of leaves of a simple 2-3 tree $T$ is equal to $w(T)$.

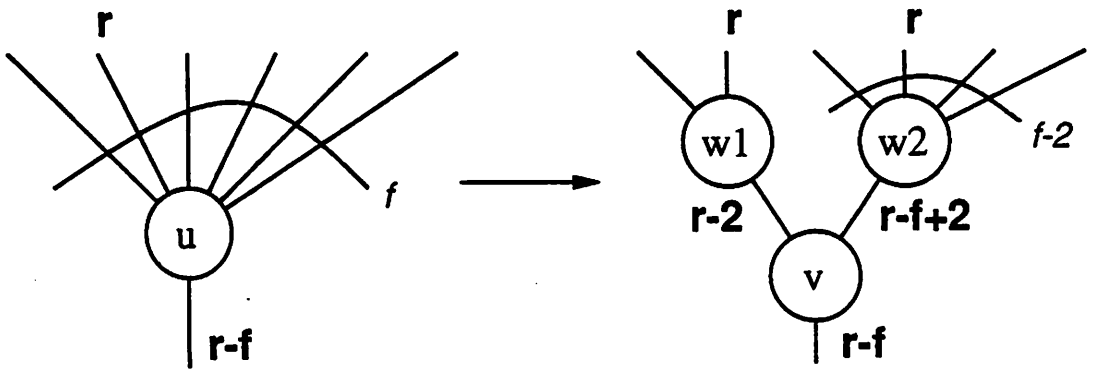


Figure 3.3: Splitting a Node Does not Increase Delay

**Theorem 3.3.1** *Let $(l_2, l_3)$ be a pair of integers that realizes the minimum of the quantity $2x + 3y$ subject to the constraint $2^x 3^y \geq n$. Let $T$ be a simple 2-3 tree with $l_2$ levels of nodes of fanout 2 and $l_3$ levels of nodes of fanout 3. Then $T$ has minimum delay among all fanout trees with $n$ leaves or more. Its delay is given by the following expression:*

$$d(T) = 3h + 1 \quad \text{if} \quad 3^h < n \leq \frac{4}{3} \times 3^h$$
$$d(T) = 3h + 2 \quad \text{if} \quad \frac{4}{3} \times 3^h < n \leq 2 \times 3^h$$
$$d(T) = 3h + 3 \quad \text{if} \quad 2 \times 3^h < n \leq 3^{h+1}$$

The proof of theorem 3.3.1 relies on the following two lemmas.

**lemma 3.3.2** *There is an optimal fanout tree that is a 2-3 tree.*

**Proof** Let $T$ be an optimal fanout tree. Its nodes with fanout equal to 1 can be eliminated without modifying the fanout of other nodes and without increasing the delay through the tree. We will show that its nodes with fanout greater than 3 can be split into nodes with strictly smaller fanouts without increasing the delay through the tree, with the transformation shown in figure 3.3. A node $u$ with a fanout $f$ of 4 or more can be replaced by three nodes $v$, $w_1$ and $w_2$, such that $v$ is directly connected to the parent of $u$, and has $w_1$ and $w_2$ as children. Two children of the children of $u$ are made children of $w_1$, while the remaining ones are made children of $w_2$. If $r$ is the earliest required time of any child of $u$,

the required time at $u$ is $r - f$. In the worst case the earliest required time of a child of $w_1$ is $r$ and the required time of $w_1$ is $r - 2$. Similarly, in the worst case the earliest required time of a child of $w_2$ is $r$ and the required time of $w_2$ is $r - (f - 2)$. Thus the required time at $v$ is no worse than $\min(r - 2, r - f + 2) + 2 = r - f$, since $f \geq 4$. ∎

Lemma 3.3.2 shows that we can find a 2-3 tree that is an optimal fanout tree. The following lemma shows that we can restrict our attention further to simple 2-3 trees.

**lemma 3.3.3** *Let $T$ be a 2-3 tree. There is a simple 2-3 tree $T'$ that is at least as fast as $T$ and has at least as many leaves as $T$.*

**Proof** Let $T$ be a 2-3 tree and let $l(T)$ be the number of leaves of $T$. The proof proceeds in two steps. We first show that $l(T) \leq w(T)$ for any 2-3 tree, and then we show that for any 2-3 tree $T$ there is a simple 2-3 tree $T'$ such that $d(T') \leq d(T)$ and $w(T') = w(T)$. This would prove the lemma, since for any simple 2-3 tree $T'$, $l(T') = w(T')$.

To prove that $l(T) \leq w(T)$, we proceed by induction on the height of $T$. The result is obviously true for trees of height zero, since the number of leaves and the weight are both equal to 1 in that case. Let us suppose that the result is true for all 2-3 trees of height $h - 1 \geq 0$. Let $(T_1, \ldots, T_k)$, with $k = 2$ or $k = 3$, be the subtrees of $T$ that are the fanouts of the root node of $T$. By induction hypothesis, $l(T_i) \leq w(T_i)$. Moreover we have $l(T) = \sum_{1 \leq i \leq k} l(T_i)$ and $w(T) = k \times \max_{1 \leq i \leq k} w(T_i)$. Thus $l(T) \leq w(T)$.

We can build a simple 2-3 tree $T'$ such that $d(T') \leq d(T)$ and $w(T') = w(T)$ as follows. Let $p$ be a path of $T$ such that $w(p) = w(T)$, and let $T'$ be a simple 2-3 tree with $x_p$ levels of nodes of fanout 2 and $y_p$ levels of nodes of fanout 3. We have $w(T') = w(p) = w(T)$ and $d(T') = d(p) \leq d(T)$. ∎

**Proof** [of theorem 3.3.1] According to the previous two lemmas, we only need to consider simple 2-3 trees. The optimal simple 2-3 tree is the simple 2-3 tree $T$ that minimizes $d(T)$ subject to the constraint that $l(T) \geq n$, where $l(T)$ is the number of leaves of tree $T$. For a given simple 2-3 tree $T$, with $x$ levels of nodes with fanout 2 and $y$ levels of nodes with fanout 3, we have: $d(T) = 2x + 3y$ and $l(T) = w(T) = 2^x \times 3^y$. Thus the problem of finding an optimal simple 2-3 tree is reduced to the problem of finding a pair of integers $(x, y)$ that is solution of the following discrete optimization problem:

$$\min_{x,y} 2x + 3y \quad with \quad 2^x \times 3^y \geq n \tag{3.6}$$

We will first show that there is always a solution of 3.6 that is such that $0 \leq x \leq 2$. For any pair $(x, y)$ of integers, let $d(x, y) = 2x + 3y$ and $l(x, y) = 2^x \times 3^y$. If $x \geq 3$, we have:

$$d(x-3, y+2) = 2\,(x-3) + 3\,(y+2) \;=\; 2\,x + 3\,y = d(x, y)$$
$$l(x-3, y+2) = 2^{x-3}\,3^{y+2} \;>\; 2^x\,3^y = l(x, y)$$

In other words, if $x \geq 3$ and $(x, y)$ is an optimum solution, we can replace $(x, y)$ by $(x - 3, y + 2)$ without loss of optimality. Let us suppose that $n > 3^h$, for some integer $h$. Then we have $2^x\,3^y > 3^h$. Since $x \leq 2$, we must have $y \geq h - 1$. In addition, if $y = h - 1$, then $x = 2$ and $d(2, h-1) = 3h + 1$. If $y = h$, then $x \geq 1$, and $d(x, h) \geq 3h + 2$. If $y > h$, then $d(x, y) > 3h + 3$. Thus $(2, h-1)$ is an optimum solution for $n$ in the range: $3^h < n \leq l(2, h-1) = \frac{4}{3} \times 3^h$. If $n > \frac{4}{3} \times 3^h$, then $y \geq h$ and $x \geq 1$. The minimum delay solution satisfying these constraints is $(1, h)$, and $d(1, h) = 3h + 2$. This solution is optimum for $n$ is the range: $\frac{4}{3} \times 3^h < n \leq l(1, h) = 2 \times 3^h$. If $n > 2 \times 3^h$, then the next minimum delay solution is $(0, h+1)$, that has a delay of $d(0, h+1) = 3h + 3$. This solution is optimum for $n$ in the range $2 \times 3^h < n \leq l(0, h+1) = 3^{h+1}$. ∎

**Arbitrary Required Times** We are not aware of any polynomial time algorithm to solve the minimum delay fanout problem for arbitrary required times under the unit fanout delay model. We conjecture that this problem is not NP-complete and that such an algorithm exists.

**Area Minimization under a Delay Constraint** We conjecture that the problem of finding minimum area fanout tree under a delay constraint for arbitrary required times and the unit fanout delay model is NP-complete.

### 3.3.3 Unit Fanout Model with Varying Sink Loads

The difference between the unit fanout model of the previous section and the unit fanout model with varying sink loads used in this section is that we now allow sink loads to take any positive rational value. There is still only one buffer in the library, and its drive capability is 1, its intrinsic delay 0 and its input load 1. Under this delay model, we will prove that the fanout problem for minimum delay is NP-complete even if we restrict the sink required times to be all equal. We will also prove that the fanout problem for minimum area under a delay constraint is NP-complete even if we restrict sink loads to be integers.

### Fanout Problem for Minimum Delay

**Theorem 3.3.4** *Given a fanout problem for the unit fanout model with rational sink loads and a constant $D$, the following decision problem is NP-complete: is there a fanout tree such that the delay through the fanout tree is less than or equal to $D$. The problem remains NP-complete even if the required times at the sinks are all equal.*

The decision problem is clearly in NP. To prove it is NP-complete, we will exhibit a polynomial time reduction of 3-partition to it. For clarity, we restate here the 3-partition problem [16]:

**Theorem 3.3.5 (3-Partition)** *Given a finite set $A$ of $3m$ elements, an integer valued bound $B$, and an integer valued size $s(a)$ for each element $a$ of $A$, such that $s(a)$ satisfies $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$, the following decision problem is NP-complete: can $A$ be partitioned into $m$ disjoint sets $S_1, \ldots, S_m$ such that for $1 \leq i \leq m$ $\sum_{a \in S_i} s(a) = B$?*

The nature of the constraints is such that if a solution exists, the sets $S_i$ contain exactly 3 elements.

A decision problem is NP-complete in the strong sense if, unless $P = NP$, there is no polynomial algorithm to solve the decision problem even if we restrict the problem to instances where the numbers appearing in an instance are bounded by a polynomial function of the size of the instance. Equivalently, a decision problem is NP-complete in the strong sense if it cannot be solved by a pseudo-polynomial algorithm, i.e. an algorithm that is polynomial in the size of the instance and the magnitude of the numbers appearing in the instance. NP-complete problems that are not strongly NP-complete derive their NP-completeness from the presence of exponentially large numbers in the formulation of their instances. PARTITION [16] is an example of an NP-complete problem that is not strongly NP-complete: it can be solved in pseudo-polynomial time by dynamic programming. When the numbers appearing in the instances of a problem are derived from finite precision physical parameters, as is the case for fanout optimization, it is not realistic to suppose the presence of exponentially large numbers in problem instances. In other words, to be relevant, proofs of NP-completeness of fanout problems need to show NP-completeness in the strong sense, by derivation from a strongly NP-complete problem. Our NP-completeness

proofs are based on 3-partition, which is one of the simplest strongly NP-complete problems [16].

The proof of theorem 3.3.4 relies on the following lemma:

**lemma 3.3.6 (Restricted 3-partition)** *3-partition remains an NP-complete problem even if we restrict the number of elements of an instance to be a power of 3, i.e. of the form* $3m = 3^h$.

**Proof** Let $A$ be an instance of 3-partition. We will exhibit an instance $A'$ of restricted 3-partition that is equivalent to $A$. $A'$ is constructed as follows. It has $3^h$ elements, where $h = \lceil \log_3(3m) \rceil$. The first $3m$ elements of $A'$ are a copy of the elements of $A$, with their size multiplied by 9. The remaining $3^h - 3m$ elements of $A'$ are grouped in triplets, of respective sizes $(3B+1, 3B+1, 3B-2)$. And $B'$ is taken to be equal to $9B$. Suppose that $A$ has a 3-partition. Then the first $3m$ elements of $A'$ can be grouped together in triplets of total size $9B$. The remaining $3^h - 3m$ elements can be kept together as they were created, in triplets of total size $9B$. Thus $A'$ has a 3-partition. Conversely, if $A'$ has a 3-partition, the sum of the sizes of any triplet $(s_1, s_2, s_3)$ in this 3-partition is equal to $9B$, and is thus divisible by 9. If any of these elements comes from an element of $A$, then all of them do, otherwise the sum of their sizes would not be 0 modulo 9. Thus a 3-partition of $A'$ yields a 3-partition of $A$. ∎

**Proof** [of theorem 3.3.4] We only need to exhibit a polynomial time reduction of 3-partition to the fanout problem for instances of 3-partitions such that $3m = 3^h$. Let $A$ be such an instance of 3-partition. We create $3^h = 3m$ sinks, one per element of $A$, and assign to the sink corresponding to element $a$ of $A$ the load $1 + s(a)/K$ where $K$ is an arbitrary integer such that $K > 3/2B$. All required times are taken to be equal and $D$ is set to be equal to $3h + B/K$. Clearly, this specifies a fanout problem for our delay model, and the construction can be done in time polynomial in the size of the instance $A$. We need now to prove that decision problem $A$ is equivalent to this fanout problem.

Suppose that $A$ has a solution. Then we can group the elements of $A$ in triplets $S_1, \ldots, S_m$, such that $\sum_{a \in S_i} s(a) = B$ for $1 \leq i \leq m$. We can then build a 3-tree with $h$ levels and $3^h$ leaves, such that the sinks corresponding to the elements of $S_i$ are siblings of each other. Any node of the tree at level $h-1$ has a fanout of 3, and the load it

drives is equal to $3 + B/K$ since $\sum_{a \in S_i} s(a) = B$. Thus the total delay through the tree is $3(h-1) + 3 + B/K = D$, which proves that the fanout decision problem has a solution.

Conversely, suppose that the fanout decision problem has a solution. There is then a fanout tree whose delay is no greater than $D$. From lemma 3.3.2, which still holds if the loads at the sinks are allowed to be larger than 1, we can assume without loss of generality that this fanout tree is a 2-3 tree. Let $T$ be the 3-tree with $3^h$ leaves and a depth of $h$, with sinks allocated to its leaves in some arbitrary way. Since the load of any sink is less than $1 + B/2K$, the load of buffers at level $h-1$ does not exceed $3(1 + B/2K) < 3 + 1$. Thus, the delay through $T$ is smaller than $3h + 1$. Since no other 2-3 tree can drive $3^h$ outputs of load 1 or more in less than $3h + 1$, $T$ is the only possible 2-3 tree that realizes the minimum delay fanout tree. Thus there is an assignment of sinks to leaves of $T$ such that the delay through $T$ is no larger than $D = 3h + B/K$. This means that the sinks can be 3-partitioned into triplets whose aggregate load is equal to $3 + B/K$. This is equivalent to saying that $A$ can be 3-partitioned. ∎

### Fanout Problem for Minimum Area under a Delay Constraint

**Theorem 3.3.7** *Given a fanout problem for the unit fanout model with integer sink loads, a delay constraint $D$ and a constant $A$, the following decision problem is NP-complete: is there a fanout tree such that the delay through the tree is less than or equal to $D$ and its area is less than $A$? The problem remains NP-complete even if the required times at the sinks are all equal.*

**Proof** This decision problem is clearly in NP. To prove it is NP-complete we will again exhibit a polynomial time reduction of 3-partition to it. From a given instance $A$ of 3-partition, we build an instance of the fanout problem as follows: we create $3m$ sinks, one for each element of $A$, having for loads the values $(m+1)s(a)$. The area constraint is $A = m$ and the delay constraint is $D = m + (m+1)B$. All required times are taken to be the same. To show that both problems are equivalent, we first note that if a fanout tree is such that less than $m$ gates are directly connected to the sinks, there must be at least one gate which fanouts to 4 or more sinks. Given the constraint that $s(a) > B/4$, the load at this gate must be greater than $(m+1)(B+1) > m + (m+1)B$. Thus to be able to meet the delay constraint, all $m$ gates that the fanout tree is allowed to contain under the area constraint must be to used to drive sinks. Moreover, each of them has to drive exactly 3 sinks. In

that case, the delay constraint will be met if and only if the loads are equilibrated, that is if and only if $A$ can be 3-partitioned. ∎

### 3.3.4 Berman's Delay Model

Berman *et al.* used a different delay model in their work [5]. In their model, gates have a fixed fanout and delay. This is equivalent to saying that the load at the output of a gate is equal to the number of fanouts, and the delay through a gate is a piece-wise linear function of the load, with a threshold value above which the delay through a gate becomes infinite. Under this delay model they show that the fanout problem for minimum area under a delay constraint is NP-complete, but their proof relies on unrealistic assumptions on gate size and gate delay parameters, which weakens their result. More specifically, they suppose a library containing gates with the following area and delay characteristics, where $N$ and $K$ are given integer-valued parameters: a gate with delay 1, fanout limit of $1 + \frac{N}{3}$ and area $(NK)^3$, and, for $i = 1, \ldots, K$, a gate with delay $2NK + i$, fanout limit 2 and area $(NK)^2 - 3i$.

## 3.4 A Spectrum of Fanout Optimization Algorithms

In what follows, we present a list of fanout optimization algorithms, sorted in order of increasing complexity. Each of the algorithms is analyzed in terms of its computational complexity and optimization ability. All of these algorithms have been implemented and an empirical analysis of their efficiency is given in section 3.8. These algorithms can introduce buffers, inverters, or a combination of both. However, to simplify the presentation, we suppose that only buffers are used, and the source and the sinks are all of the same polarity. We postpone the problem of correct phase selection until section 3.5. We first introduce some notation that are used in the rest of this section.

### 3.4.1 Notation

We use $r$, possibly with indices, to denote required times; $\alpha$ to denote intrinsic delay, $\beta$ drive capabilities and $\gamma$ loads. The number of sinks is $n$, and the number of buffers or inverters in the library $\mathcal{L}$ is $d$. The letter $b$ designates a buffer. We use $\beta_s$ to designate the drive capability of the source, and $beta_b$ the drive capability of the buffer $b$. The input

load of a buffer is $\gamma_b$.

We suppose that, in a preprocessing step, the sinks have been sorted in order of increasing required times, and their required times are denoted $(r_1, \ldots, r_n)$. In particular, the following equation holds: $r_i = \min_{j \geq i} r_j$. Similarly, their loads are denoted $(\gamma_1, \ldots, \gamma_n)$, in the same order. We also precompute the quantities $\gamma_{i,n} = \sum_{i \leq k \leq n} \gamma_k$. The quantities $\gamma_{i,j} = \sum_{i \leq k \leq j} \gamma_k$ are then available in constant time as $\gamma_{i,n} - \gamma_{j+1,n}$. The entire preprocessing can be done in time $O(n \log n)$ and is necessary in all of our algorithms, except the first two (buffer selection and two-level fanout tree selection ignoring required times).

### 3.4.2 Buffer Selection

The buffer selection algorithm is a very rudimentary fanout optimization algorithm. It does not build any fanout tree; it simply sizes existing buffers optimally. If a fanout tree is implemented as a wire, this algorithm has no effect. Buffer selection can also be used inside trees as was done in chapter 2.

For a given buffer selection $b$, the algorithm computes the required time at the input of the buffer. If $r_0$ is the required time at the output of the buffer, and $\gamma_{1,n}$ the load at the output of the buffer, the delay through the buffer is $\alpha_b + \beta_b \gamma_{1,n}$ and the required time at the input of the buffer is $r_0 - \alpha_b - \beta_b \gamma_{1,n}$. Unfortunately this quantity does not take into account the delay due to the input load of the buffer, which should be taken into account to make an optimal buffer selection. So we subtract from the required time at the input of the buffer the load dependent delay $\beta_s \gamma_b$ corresponding to the time it takes the source gate to drive the buffer input. The algorithm selects a buffer that maximizes this quantity as shown in Figure 3.4. The algorithm works in time $O(d+n)$ where $d$ is the number of buffers in the library and $n$ the number of sinks. The computation of $r_0$ is not actually necessary and is given in Figure 3.4 for clarity.

### 3.4.3 Two-Level Fanout Trees Ignoring Required Times

One of the main reasons why fanout optimization is necessary is to reduce large loads created by large fanouts. The simplest way to do so is to insert a *two-level tree* of buffers at multiple fanout points in the circuit, where a two-level tree is defined as follows:

**Definition 4** *A tree is a two-level tree if any leaf of the tree is separated from the root of the tree by exactly one intermediate node.*

**algorithm** *buffer_selection*

$r_0 = r_1 = min_{1 \le i \le n} r_i$

$b_{opt} = \arg\max_{b \in \mathcal{L}} (r_0 - \alpha_b - \beta_b \gamma_{1,n} - \beta_s \gamma_b)$

**end** *buffer_selection*

Figure 3.4: Optimal Buffer Selection

The two-level fanout tree algorithm which ignores required times is given in Figure 3.5 and explained in detail in the rest of this section.

**Two-Level vs. Multi-Level** In general, the optimal number of levels of such a tree is a logarithmic function of the ratio between the load $\gamma_{1,n}$ and the drive capability $\beta_s$ of the source of the signal. In practice, however, this ratio never grows large enough to justify the use of more than one level of buffers, i.e. of two-level buffer trees. This can be checked by a simple back-of-the-envelope computation. To make things concrete, we use delay values from the MCNC library `lib2` [32] and round to the nearest number with one significant digit. We suppose for a buffer a drive capability of 2.0, an intrinsic delay of 0.3, and a load of 0.1, a drive capability of 4.0 for the source and a load of 0.1 for each of the sinks. The delay values obtained with no buffer tree, with a two-level buffer tree, and the best multi-level buffer tree are reported in table 3.1 for a varying number of sinks. A number of fanouts in the range of 10 to 20 is typical, above 50 is rare. The largest number we have observed was 198.

As the data indicates, the gains obtained by using more than one level of buffers is only substantial for very large fanouts, and in any case is negligible compared to the gains obtained by introducing one level of buffers. In addition, in practice, libraries contain buffers of several strengths, which makes the two level fanout trees competitive in an even larger range of fanouts.

**Buffer Selection** A two-level fanout tree is composed of one level of intermediate buffers. For simplicity, we enforce the restriction that all intermediate buffers are of the same strength (or buffer type). This allows us to compute in constant time a good approximation of the number of intermediate buffers required. This computation is done once for

| # sinks | 10 | 15 | 20 | 25 | 30 | 40 | 50 | 100 | 200 |
|---|---|---|---|---|---|---|---|---|---|
| no opt | 4.0 | 6.0 | 8.0 | 10.0 | 12.0 | 16.0 | 20.0 | 40.0 | 80.0 |
| two-level | 2.1 | 2.5 | 2.9 | 3.3 | 3.5 | 3.9 | 4.3 | 6.1 | 8.3 |
| multi-level | 2.1 | 2.5 | 2.8 | 3.0 | 3.0 | 3.2 | 3.4 | 4.1 | 4.5 |

Table 3.1: Fanout Trees: Two-Level vs. Multi-Level

| | |
|---|---|
| **# sinks:** | number of sinks; all required times are equal |
| **no opt:** | delay with no fanout optimization |
| **two-level:** | delay with one level of buffers (e.g. two-level fanout trees) |
| **multi-level:** | delay with the best multi-level fanout tree |

each buffer type $b$ in the library.

We first compute the total load of all the sinks, $\gamma_{1,n}$, and then we determine the optimum number $k^b_{opt}$ of intermediate buffers needed for a two-level fanout tree, supposing that all buffers are of type $b$ and the load is equally divided among the intermediate buffers. This minimization problem can be formulated as a quadratic minimization problem and can be solved exactly in constant time, using the following two formulas:

$$k^b_{opt} = \arg\min_k(\beta_s\gamma_b k + \beta_b\frac{\gamma_{i,n}}{k}) \tag{3.7}$$

$$k^b_{opt} \in \left\{\left\lfloor\sqrt{\frac{\beta_b\gamma_{1,n}}{\beta_s\gamma_b}}\right\rfloor, \left\lceil\sqrt{\frac{\beta_b\gamma_{1,n}}{\beta_s\gamma_b}}\right\rceil\right\} \tag{3.8}$$

**Sink Assignment** The number of intermediate buffers to be used is computed by supposing that the loads are equally divisible among all intermediate buffers. This is not the case in general. Unfortunately, even if the number of intermediate buffers is given, assigning sinks of varying loads to these buffers is a difficult problem. It is equivalent to multiprocessor scheduling, which is known to be NP-complete (see [16] page 238).

To perform sink assignment, we use a simple greedy algorithm that allocates the next sink to the intermediate buffer that has been assigned the least amount of load so far. The best results are obtained if the sinks are sorted in order of decreasing loads. This assignment is made for each buffer type $b$, but only for $k^b_{opt}$ intermediate buffers. The best solution is then retained.

**Complexity** The number of intermediate buffers is of the order of $\sqrt{n}$. The best fit algorithm spends $\sqrt{n}$ time to determine where how to assign each sink. Moreover this

```
algorithm two_level_no_required_times
    sort the sinks by decreasing load values (γ_1, ..., γ_n)
    compute γ_{1,n} = Σ_{1≤i≤n} γ_i
    foreach b ∈ L {
        k^b_opt = sqrt( β_b γ_{1,n} / (β_s γ_b) )
        foreach 1 ≤ i ≤ k^b_opt load[i] = 0.0
        foreach 1 ≤ i ≤ n {
            j_i = arg min_{1≤j≤k^b_opt} load[j]
            load[j_i] = load[j_i] + γ_i
            assign[i] = j_i
        }
    }
end two_level_no_required_times
```

Figure 3.5: Two-Level Fanout Tree Ignoring Required Times

computation is done once for each buffer type. Thus, in total, the complexity of this algorithm is $O(d\ n^{1.5})$.

### 3.4.4 Two-Level Fanout Trees Taking Required Times into Account

The previous algorithm ignores sink required times. Though it can handle non constant load values, it does not perform very well in the presence of wide variations in required times. It is possible to compensate for this deficiency by taking required times into account during sink assignment. To do so we still use a best fit, greedy algorithm but instead of assigning a sink to the intermediate buffer that has so far the least amount of load to drive, we assign a sink to an intermediate buffer in such a way that the required time at the source of the fanout tree is decreased the least by this assignment. If all required times are equal, these two greedy algorithms produce the same result. In the preprocessing phase, we sort the sinks in order of increasing required times, and, in case of ties, in order of decreasing loads. The complexity of this algorithm is also $O(d\ n^{1.5})$. The algorithm is sketched in Figure 3.6.

```
algorithm two_level_with_required_times
    sort the sinks by increasing required times (r_1, ..., r_n)
    compute γ_{1,n} = Σ_{1≤i≤n} γ_i
    foreach b ∈ L {
        k^b_opt = sqrt(β_b γ_{1,n} / β_s γ_b)
        foreach 1 ≤ i ≤ k^b_opt {
            required[i] = 0.0;
            load[i] = 0.0;
        }
        foreach 1 ≤ i ≤ n {
            foreach 1 ≤ j ≤ k^b_opt {
                required[i, j] = min(r_i − β_b load[j], required[j]) − β_b γ_i
            }
            j_i = arg max_{1≤j≤k^b_opt} required[i, j]
            required[j_i] = required[i, j_i]
            load[j_i] = load[j_i] + γ_i
            assign[i] = j_i
        }
    }
end two_level_with_required_times
```

Figure 3.6: Two-Level Fanout Tree Taking Required Times into Account

### Optimality

**lemma 3.4.1** *The greedy sink assignment algorithm is not optimal. This is still true even if all sink loads are equal and all required times are integer valued.*

**Proof** A counter example is given in figure 3.7. ∎

We use the greedy sink assignment algorithm because it can handle non constant required times as well as non constant loads. However, if all loads and all buffer drives are equal, and if we assume for simplicity that all required times are integer valued, we can solve the sink assignment problem in time $O(n \log n)$. This result is achieved by using a decision procedure that can decide in linear time whether there exists a sink assignment that can produce a

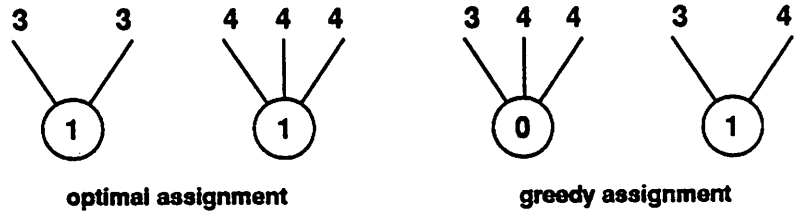


Figure 3.7: The Greedy Sink Assignment Algorithm is Not Optimal

given required time at the source of the fanout tree. An $O(n \log n)$ optimal algorithm can be derived from this decision procedure by using binary search.

**lemma 3.4.2** *Given $k$ identical buffers, with drive capacity equal to 1, $n$ sinks, with loads equal to 1 and integer required times, and an integer constant $D$, the following decision problem can be solved in linear time: is there a sink assignment such that the required time at any of the buffers is no less than $D$?*

**Proof** We assume that the sinks are sorted in order of increasing required times. By convention we will use the letters $i$ and $j$ as sink indices, and $b$ as a buffer index. We have: $1 \leq b \leq k$. We first prove that there is an optimal sink assignment that assigns consecutive sinks to each buffer. Let $\sigma$ be an optimal sink assignment. Let $R_b^\sigma = \min_{i,\sigma(i)=b} r_i$. The required time of buffer $b$ is equal to $R_b^\sigma - |\{i, \sigma(i) = b\}|$. Without loss of generality, we can always suppose that the buffers are sorted in order of increasing $R_b$, i.e. if $b < b'$ then $R_b \leq R_{b'}$. If sinks are not assigned consecutively by $\sigma$, there exists a pair of sinks, $(s_i, s_j)$ such that $\sigma(i) < \sigma(j)$ and $r_i > r_j$. Interchanging the sinks $s_i$ and $s_j$ does not change the loads of the buffers, does not decrease the value of $R_{\sigma(i)}$ since $\sigma(i) < \sigma(j)$ and thus $R_{\sigma(i)} \leq R_{\sigma(j)} \leq r_j$, and does not decrease the value of $R_{\sigma(j)}$ since $r_i > r_j \geq R_{\sigma(j)}$. By interchanging pairs of sinks that are assigned out of order, we can therefore obtain a sink assignment that is no worse than the original optimal assignment and is such that each buffer is assigned consecutive sinks in required time order. We can thus limit our attention to consecutive assignments.

With the following algorithm, we can decide in linear time whether there exists a consecutive assignment such that the required time at the input of each buffer is at least $D$.

If such a consecutive assignment exists, it assigns to the first buffer the sinks $(s_1, \ldots, s_{i_1} - 1)$ where $i_1$ is the first index for which the required time at the input of the first buffer becomes less than $D$. It assigns the remaining sinks to the remaining buffers recursively using the same principle. More precisely, it assigns to the $b^{th}$ buffer the sinks $(s_{j_{b-1}}, \ldots, s_{j_b-1})$ where the finite sequence $(j_b)_{0 \le b \le k}$ is determined by the following recurrence equations:

$$\begin{aligned} i_0 &= 1 \\ i_{b+1} &= \min\{i, r_{i_b} - (i - i_b) < D\} \end{aligned}$$

The required time at the input of buffer $b$ is given by the expression $r_{i_{b-1}} - (i_b - i_{b-1})$ since all sink loads are equal to 1 and the drives of the buffers are all equal to 1. The answer to the decision problem is "yes" if and only if any $i_b$ exceeds $n$. ∎

### 3.4.5 Combinational Merging

The previous two algorithms are very limited in the kind of fanout trees they can produce. These structures are sufficient for most practical fanout sizes only if the required times at the sinks are close to each other. This is not often the case. To be able to obtain faster fanout trees, it is desirable to explore a larger set of fanout structures than just two-level trees.

Combinational merging is a simple, $O(n \log n)$ algorithm which has the ability to generate a rich set of fanout tree structures. Unfortunately, it relies on the characteristics of a simple delay model, and needs to be adapted heuristically to a more complex delay model. The basic step of this algorithm, illustrated in Figure 3.2, is simple. We first suppose that the sinks have been sorted in order in increasing required times $(r_1, \ldots, r_n)$. We take a group of sinks with the largest required times $(r_k, \ldots, r_n)$, make them the children of a new buffer node and remove them from the list of sinks. We then compute the required time at the new buffer node, and merge it with the sorted list of sinks. This transformation is applied until $k$ becomes equal to 1. In that case the source is used to drive the remaining sinks directly, unless insertering a buffer between the source and the sinks yields a faster circuit.

With our delay model, there are two questions to be answered before combinational merging can be used: how $k$ should be chosen, and which type $b$ of buffer should be used. We use a heuristic that computes $k = k_b$ as a function of $b$, and we select the best $b$ using

**algorithm** *bottom_up_fanout_tree_construction*
  sort the sinks $s_i$ by increasing required times $(r_1, \ldots, r_n)$
  **while** $n > 1$ {
    **foreach** $1 \leq i \leq n$ compute $\gamma_{i,n} = \sum_{i \leq j \leq n} \gamma_j$
    **foreach** $b \in \mathcal{L}$ {
      $k_{opt}^b = \sqrt{\frac{\beta_b \gamma_{1,n}}{\beta_s \gamma_b}}$
      $k_b = \max\left\{i, 1 \leq i \leq n, \gamma_{i,n} \geq \frac{\gamma_{1,n}}{k_{opt}^b}\right\}$
      $required[b] = r_{k_b} - \beta_b \gamma_{k_b,n} - \alpha_b - \beta_s \gamma_b$
      }
    $b = \arg\max_{g \in \mathcal{L}} required[g]$
    $k = k_b$
    create a new buffer node $v$ of type $b$
    attach the sinks $(s_k, \ldots, s_n)$ to $v$
    remove the sinks $(s_k, \ldots, s_n)$
    compute the required time of $v$: $r_v = \min_{k \leq i \leq n} r_i - \beta_b \gamma_{k,n} - \alpha_b$
    add $v$ in order to the list of remaining sinks $(s_1, \ldots, s_{k-1})$
    set $n = k$
  }
  attach the only remaining sink to the source node
**end** *bottom_up_fanout_tree_construction*

Figure 3.8: Combinational Merging as Fanout Algorithm

some cost function. For each buffer $b$, we compute $k_{opt}^b$ as in equation 3.7. We take $k_b$ such that the sum of the loads of the sinks of index $k_b$ to $n$ just exceed the quantity $\frac{\gamma_{1,n}}{k_{opt}^b}$. We then create a new buffer of type $b$, connect it directly to a new copy of the source, and make it drive the $k_b$ sinks with largest required times. We compute the required time at this new source, and select the buffer type $b$ that maximizes this required time, and use $k_b$ for $k$. This algorithm is given in Figure 3.8.

The choice of this heuristic can be motivated as follows. For a given $b$, we need to determine an adequate value for $k$. A choice based on taking a $\frac{1}{k_{opt}^b}$ fraction of the total remaining sink load appears reasonable, since, in the case where all required times are equal, it leads to a two-level tree that is close to the optimum fanout tree. To compute $k_{opt}^b$, we

suppose that the buffers are driven by the source gate; this is too restrictive in general and is likely to lead to suboptimal results. Yoshikawa *et al.* [44] recently proposed an extension of this algorithm, based on branch and bound techniques, that does not suffer from this limitation.

**Complexity** The complexity of this algorithm can be analyzed, provided that we make the following simplifying assumption: at each step of the algorithm, the number of sinks is decreased by $\alpha\sqrt{n}$ for some constant $\alpha$. If we suppose that all sinks have the same load, and all loads and all drive capabilities are equal, we have $k_{opt} = \sqrt{n}$, and $k = n + 1 - \lceil\sqrt{n}\rceil$. If we simply suppose that all sink loads are equal to some nominal value $\gamma$, we have: $k = n + 1 - \left\lceil\sqrt{n}\sqrt{\frac{\gamma_b\beta_s}{\gamma\beta_b}}\right\rceil$. To make things more concrete, we computed the quantity $\sqrt{\frac{\gamma_b\beta_s}{\gamma\beta_b}}$ for all three inverters of the MCNC library, using as source a 2-input NAND gate and as sink the input of a 2-input NAND gate. The actual values of this coefficient were 0.76, 1.64 and 3.04. The larger the buffer is, the larger this coefficient is, since increasing the size of the buffer has the effect of decreasing the load dependent delay coefficient $\beta_b$ and increasing the input load $\gamma_b$.

Thus each step of the algorithm is guaranteed to reduce the number of sinks by $\alpha\sqrt{n}$, where $\alpha$ is a constant depending on the library, the drive of the source and the loads of the sinks. Inserting a new sink takes $O(\log n)$ time and recomputing the quantities $\gamma_{i,n}$ take $O(n)$. Each step is done $d$ times, once per buffer. In total, the complexity of the algorithm is $O(dn\ f(n))$ where $f(x) = \min\{k, g^k(x) \leq 1\}$ with $g(x) = x - \alpha\sqrt{x}$ and $g^k(x) = g^{k-1}(g(x))$. Using the result of the following lemma, we deduce that the complexity of this algorithm is $O(dn^{1.5})$.

**lemma 3.4.3** *Asymptotically, $f(n)$ does not exceed $\frac{2}{\alpha}\sqrt{n}$:*

$$\limsup_{n\to+\infty} \frac{f(n)}{\sqrt{n}} \geq \frac{2}{\alpha} \tag{3.9}$$

**Proof** First we note that $f$ is unchanged if we modify $g$ so that $g(x) = 0$ for all values of $x$ satisfying $x < \alpha^2$. For any given $n \geq \alpha^2$ we have, since the sequence $g^i(n)$ is monotonically nondecreasing:

$$g^k(n) = n \prod_{1\leq i<k} \left(1 - \frac{\alpha}{g^i(n)}\right) \tag{3.10}$$

$$\leq n \prod_{1 \leq i < k} \left(1 - \frac{\alpha}{n}\right) \tag{3.11}$$

$$\leq n \left(1 - \frac{\alpha}{\sqrt{n}}\right)^k \tag{3.12}$$

Using the inequality $(1 - \frac{1}{x})^x \leq e$ valid for all $x \geq 1$, we can derive, for any given real number $\epsilon > 0$, the following inequality:

$$g^{\lceil \epsilon\sqrt{n} \rceil}(n) \leq n \left(1 - \frac{\alpha}{\sqrt{n}}\right)^{\lceil \epsilon\sqrt{n} \rceil} \tag{3.13}$$

$$\leq n \left(1 - \frac{\alpha}{\sqrt{n}}\right)^{\epsilon\alpha\frac{\sqrt{n}}{\alpha}} \tag{3.14}$$

$$\leq \frac{n}{e^{\epsilon\alpha}} \tag{3.15}$$

This inequality is also valid for $0 < n < \alpha^2$, since in that case $\lceil \epsilon\sqrt{n} \rceil \geq 1$, and thus $g(n) = 0 < \frac{n}{e^{\epsilon\alpha}}$. By applying this inequality to $\frac{n}{e^{\epsilon\alpha}}$ we obtain:

$$g^{\lceil \epsilon\sqrt{\frac{n}{e^{\epsilon\alpha}}} \rceil}\left(\frac{n}{e^{\epsilon\alpha}}\right) \leq \frac{n}{e^{2\epsilon\alpha}} \tag{3.16}$$

Since $g$ is monotonic nondecreasing, we deduce from the previous two inequalities:

$$g^{\lceil \epsilon\sqrt{n} \rceil + \lceil \epsilon\sqrt{\frac{n}{e^{\epsilon\alpha}}} \rceil}(n) \leq \frac{n}{e^{2\epsilon\alpha}} \tag{3.17}$$

By induction we obtain, for any $k \geq 1$:

$$g^{\sum_{i=0}^{k-1} \left\lceil \epsilon\sqrt{\frac{n}{e^{i\epsilon\alpha}}} \right\rceil}(n) \leq \frac{n}{e^{k\epsilon\alpha}} \tag{3.18}$$

In particular, if $k = \left\lceil \frac{\log n}{\epsilon\alpha} \right\rceil$, we have $\frac{n}{e^{k\epsilon\alpha}} \leq 1$, which proves that:

$$f(n) \leq \sum_{i=0}^{\lceil \frac{\log n}{\epsilon\alpha} \rceil - 1} \lceil \epsilon\sqrt{\frac{n}{e^{i\epsilon\alpha}}} \rceil \tag{3.19}$$

We can deduce from this inequality that:

$$f(n) \leq \sum_{i=0}^{\lceil \frac{\log n}{\epsilon\alpha} \rceil - 1} \left(1 + \epsilon\sqrt{\frac{n}{e^{i\epsilon\alpha}}}\right) \tag{3.20}$$

$$\leq 1 + \frac{\log n}{\epsilon\alpha} + \epsilon\sqrt{n} \sum_{i=0}^{+\infty} e^{-i\frac{\epsilon\alpha}{2}} \tag{3.21}$$

$$\leq 1 + \frac{\log n}{\epsilon\alpha} + \frac{\epsilon\sqrt{n}}{1 - e^{\frac{\epsilon\alpha}{2}}} \tag{3.22}$$

Since $\epsilon$ is arbitrary, we can select it to be equal to $\frac{\log n}{\alpha n^{1/4}}$. We then obtain:

$$\frac{f(n)}{\sqrt{n}} \leq n^{-1/2} + n^{-1/4} + \frac{n^{-1/4}\log n}{\alpha\left(1 - e^{\frac{n^{-1/4}\log n}{2}}\right)} \tag{3.23}$$

We obtain finally:

$$\limsup_{n\to+\infty} \frac{f(n)}{\sqrt{n}} \leq \lim_{n\to+\infty} n^{-1/2} + n^{-1/4} + \frac{n^{-1/4}\log n}{\alpha\left(1 - e^{\frac{n^{-1/4}\log n}{2}}\right)} \tag{3.24}$$

$$\leq \frac{2}{\alpha} \tag{3.25}$$

■

### 3.4.6 Fanout Optimization based on *LT*-Trees

The main weakness of the combinational merging algorithm is that it relies on a simple-minded heuristic to determine which type of buffer to use and how many sinks among those with the largest required times should be grouped under one buffer. In this section, we propose a new fanout algorithm that realizes a compromise between the two-level fanout tree algorithms and the combinational merging algorithm. This new algorithm is based on *LT-trees*, a restricted class of fanout trees that is described below.

Like the two-level algorithms, the *LT*-tree algorithm only considers a subset of the set of all possible fanout trees. This subset is small enough for the algorithm to be practical, but large enough to allow the algorithm to perform not only buffering of large capacitive loads, but also, like the combinational merging algorithm, critical signal isolation. The *LT*-tree algorithm has the additional advantage of using dynamic programming both to select the shape of the tree and the types of buffers to be used at intermediate nodes: it gets much of its ability to generate fast fanout trees from the tight connection between gate selection and pattern selection. In that sense, it is a direct analog of tree covering algorithms.

In the rest of this section, we give a definition of *LT*-trees, describe and analyze the fanout algorithm based on *LT*-trees for delay minimization, and show how it can be extended to perform area optimization under a delay constraint, using the same technique as the one we used with tree covering.

*LT*-Trees *LT* trees are designed to be just complex enough to allow for critical signal isolation and buffering of large capacitive loads. A recursive definition of the set of *LT*

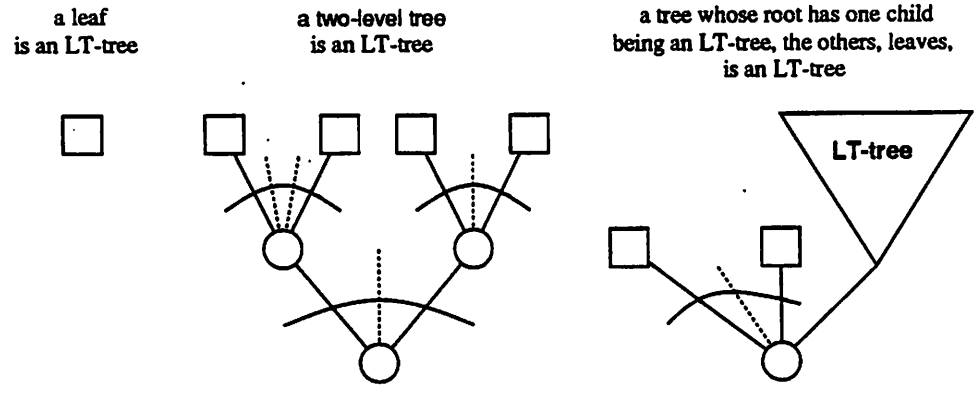


Figure 3.9: Definition of $LT$-trees

trees is given below. Figure 3.9 illustrates this definition. When an $LT$ tree is used as a fanout circuit, its root corresponds to the source of the signal, and its leaves to the sinks. If the tree is composed of a single leaf, the source and the sink are not distinguished here. In the actual circuit, they would be distinguished, and connected by a wire.

**Definition 5** *(1) A leaf is an LT tree.*

*(2) A two-level tree is a LT-tree.*

*(3) Let $T$ be a tree rooted at $r$ such that one child of $r$ is an LT tree and all the other children of $r$ are leaves. Then $T$ is an LT tree.*

In a $LT$-tree, there is at most one intermediate node that has more than one intermediate node as a child. If there is no such node, the $LT$-tree is terminated according to case (1) of the definition, and is called a $LT$-tree of type 1. If there is such a node, the node is the root of a two-level tree which terminates the $LT$-tree. In that case, the $LT$-tree is of type 2. Examples of type 1 and type 2 $LT$-trees are given in Figure 3.10. In the example of type 2, the intermediate node with more than one intermediate node as a child is highlighted.

**Theorem 3.4.4** *The number of LT-trees of type 1 is equal to $(d+1)^{n-2}$, where $n$ is the number of sinks and $d$ the number of buffers in the library.*

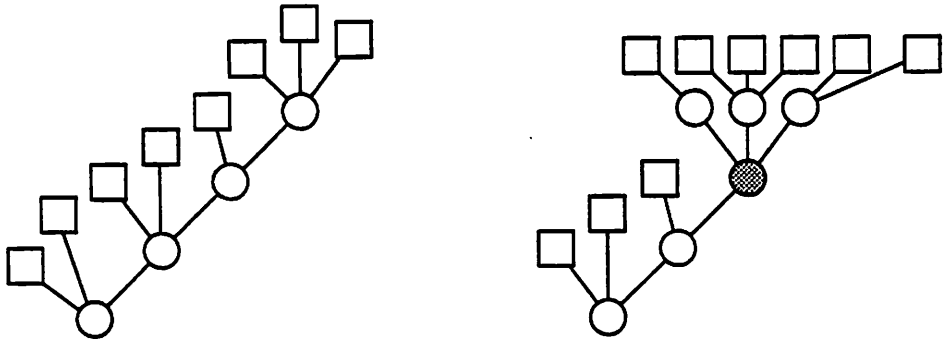


Figure 3.10: Examples of $LT$-trees

**Proof** A $LT$-tree of type 1 is entirely determined by the number $k$ of intermediate nodes of the tree, the number of leaves attached to each intermediate node, and the buffer selected at each intermediate node. For a given $k$, we can represent the topological structure of a $LT$-tree by a unique k-tuple of integers $(x_1, \leq, x_k)$ satisfying: $1 \leq x_1 < x_2 < \ldots < x_k \leq n-2$, where leaves $x_j + 1$ to $x_{j+1}$ are the leaves connected to the $j^{th}$ intermediate node (by convention we set $x_{k+1}$ to be equal to $n$). In particular, leaves 1 to $x_1$ are connected to the root node, and leaves $x_k + 1$ to $n$ are connected to the last intermediate node of the tree. The inequality $x_k \leq n-2$ is there to guarantee that the last intermediate node is connected to at least 2 sinks, as required by the definition of type 1 $LT$-trees. The number of such k-tuples of integers is equal to the number of distinct ways of choosing $k$ elements among $n-2$, i.e. $\begin{pmatrix} n-2 \\ k \end{pmatrix}$. In addition, for a given topological structure with $k$ intermediate nodes, there are exactly $d^k$ possible assignments of buffers to intermediate nodes of the tree. In total, the number of distinct $LT$-trees of type 1 is given by the formula:

$$\sum_{0 \leq k \leq n-2} \begin{pmatrix} n-2 \\ k \end{pmatrix} d^k = (d+1)^{n-2} \tag{3.26}$$

■

The $LT$-tree based algorithm explores implicitly, using dynamic programming, all $LT$-trees of type 1. For $LT$-trees of type 2, the $LT$-tree based algorithm only considers those trees whose two-level subtrees are derived from the two-level algorithm of section 3.4.4. In

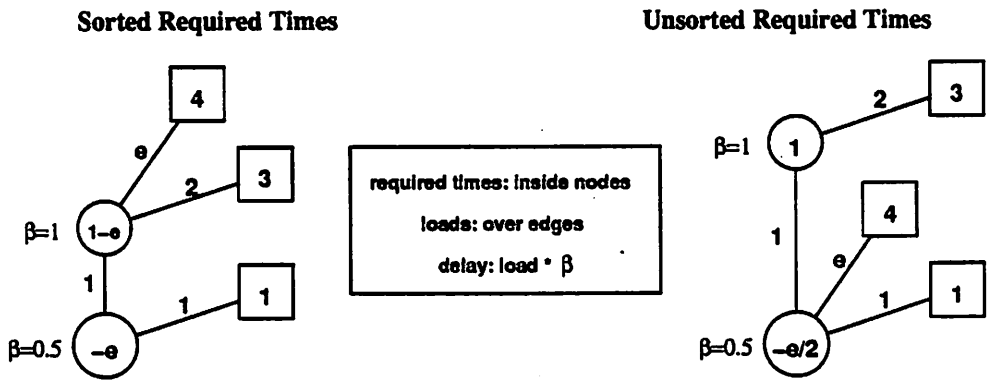


Figure 3.11: Suboptimality of $LT$-Trees with Consecutive Sink Ordering

other words, for every $LT$-tree of type 1, the algorithm only considers one $LT$-tree of type 2, for a total search space of size 2 $(d+1)^{n-2}$. If we ignore tree patterns, then the size of the search space is $2^{n-1}$. This is only a small fraction of the total number of rooted trees with $n$ leaves, which is at least of the order of the Catalan numbers $T(n) = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}$ (the Catalan numbers give the number of ways to fully parenthesize a string of n symbols). Using Stirling's formula, we can easily deduce that $T(n)$ is asymptotically equivalent to $\frac{2^{2n}}{\sqrt{\pi}\, n^{1.5}}$.

The $LT$-tree based algorithm only considers assignments of sinks to leaves of the $LT$-trees that are such that sinks with larger required times are placed further from the root of the tree. This is partially justified by the following fact:

**lemma 3.4.5** *If sink loads are all equal, there is an optimal $LT$-tree such that the sinks with larger required times are placed further from the root.*

**Proof** When loads are equal, exchanging two sinks that are out of order in a $LT$-tree can only increase the required time at the source of the tree. ■

Unfortunately, for arbitrary load values, the optimal $LT$-tree may require that sinks are placed out of order, as can be checked by inspection in the example of Figure 3.11.

**Selection of $LT$-Trees with Dynamic Programming** The $LT$-tree based algorithm selects the best $LT$-tree for a fanout problem by dynamic programming, under the restrictions presented in the previous paragraph, namely that the two-level subtrees of $LT$-trees of type 2 are restricted to be trees generated by the two-level fanout algorithm of section 3.4.4 and the sinks appear in the tree in order of increasing required times. From now on, we suppose that these restrictions are enforced, and we will only speak about $LT$-trees without additional qualifications.

The $LT$-tree based algorithm works as follows. As with previous algorithms, we first preprocess the sinks by sorting them in order of increasing required times and compute the quantities $\gamma_{i,n} = \sum_{i \leq j \leq n} \gamma_j$. Then we precompute all the two-level trees that will be considered as subtrees of candidate $LT$-trees. This precomputation is done by calling the two-level fanout algorithm of section 3.4.4 on a fanout problem composed of a source and $n - k + 1$ sinks. The source is a buffer $b$ from the library, or, if $k = 1$, the source $s$ of the original fanout problem. The sinks are the $n - k + 1$ sinks with largest required times, $(s_k, \ldots, s_n)$. This computation is done for all values of $k$ between 1 and $n$, and, for $k > 1$, all buffers in the library. For each pair $(k, g)$ in $\{(1, s)\} \cup ([2, \ldots, n] \times \mathcal{L})$, we keep in a table the required time $required_{two-level}[k, g]$ achievable by the two-level fanout algorithm. The tree itself does not need to be stored at this point: it can be recomputed if needed.

Each pair $(k, g)$ in $\{(1, s)\} \cup ([2, \ldots, n] \times \mathcal{L})$ specifies a fanout subproblem, of source $g$ and sinks $(s_k, \ldots, s_n)$. The algorithm relies on dynamic programming to compute by induction on $k$, $k$ varying from $n$ to 1, an $LT$-tree that achieves the maximum required time for each fanout subproblem $(k, g)$. For a given pair $(k, g)$, an optimal $LT$-tree $T_{(k,g)}$ can be obtained by selecting the best of the following $(n - k)d + 1$ configurations:

(1) for some sink index $l > k$ and buffer type $b$, the root of $T_{(k,g)}$ is directly connected to sinks $(s_k, \ldots, s_{l-1})$ and to a buffer of type $b$. The subtree connected to the buffer $b$ is an optimal $LT$-tree $T_{(l,b)}$ for the subproblem $(l, b)$. Since the algorithm proceeds by induction from $n$ to 1, $T_{(l,b)}$ has already been computed and is available.

(2) $T_{(k,g)}$ is a the two-level tree precomputed for pair $(k, g)$.

The algorithm is detailed in Figure 3.12. The best required time achievable for a pair $(k, g)$ is stored in the table $required[k, g]$. To keep track of the optimal configuration selected for $(k, g)$, we simply need to store a flag, $use\_two\_level[k, g]$, to decide whether a two level tree is used or not; and, in the case a two level tree is not used, an index, $next[k, g]$, which is

```
algorithm lt_tree_computation
    sort the sinks s_i by increasing required times (r_1, ..., r_n)
    foreach 1 ≤ i ≤ n compute γ_{i,n} = Σ_{i≤j≤n} γ_j
    required[n + 1, ∅] = +∞;
    for k = n to 1 {
        foreach g such that (k, g) ∈ {(1, s)} ∪ ([2, ..., n] × L) {
            required[k, g] = required_{two-level}[k, g];
            use_two_level[k, g] = true;
            foreach (l, b) ∈ {(n + 1, ∅)} ∪ [2, ..., n] × L such that l > k {
                required = min(r_k, required[l, b] − α_b) − β_g(γ_b + γ_{k,n} − γ_{l,n});
                if (required > required[k, g]) {
                    required[k, g] = required;
                    use_two_level[k, g] = false;
                    next[k, g] = l;
                    gate[k, g] = b;
                }
            }
        }
    }
end lt_tree_computation
```

Figure 3.12: Fanout Optimization with $LT$-Trees

the index of the first sink not directly attached to the root of $T_{(k,g)}$, and, if $next[k,g] < n$, a buffer type, $gate[k,g]$, which is the type of the unique buffer attached to the root of $T_{(k,g)}$. An example of computation of these entries is given in Figure 3.13. To compute the required time of a configuration other than a precomputed two-level tree, we start with the required time of the selected subproblem $(l,b)$, $required[l,b]$. This required time is not exactly the required time at the input of $T_{(l,b)}$: it does not take into account the intrinsic delay of buffer $b$. To obtain the required time at the input of $T_{(l,b)}$, we need to subtract from $required[l,b]$ the intrinsic delay $\alpha_b$ of buffer $b$. The required time at the output of the root of $T_{(k,g)}$ is then the minimum between the earliest required time of a sink connected to the root of $T_{(k,g)}$, which is $r_k$, and the required time at the input of $T_{(l,b)}$, which is $required[l,b] - \alpha_b$. To

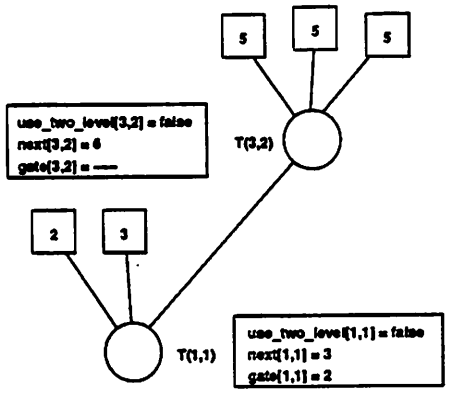


Figure 3.13: Illustration of $LT$-Tree Algorithm

obtain the required time $required[k,g]$, we need to subtract from $\min(\tau_k, required[l,b] - \alpha_b)$ the load dependent delay $\beta_g(\gamma_b + \gamma_{k,n} - \gamma_{l,n})$ but we do not include the intrinsic delay of gate $g$. $\gamma_b$ is the load of buffer $b$ while $\gamma_{k,n} - \gamma_{l,n}$ is the sum of the loads of the sinks attached to the root of $T_{(k,g)}$. The required time $required[k,g]$ is actually the best required time achieved for any possible choice of $(l,g)$, with $l > g$.

**Complexity of the Algorithm** The precomputation of two-level trees requires no more than $d \times n$ calls to the two-level fanout algorithm, for a total cost of $O(d^2\ n^{2.5})$. The computation, in the main algorithm, of a entry for a pair $(k,g)$ requires $O(d \times n)$ operations. Since there are $O(d \times n)$ such pairs, the total cost of the main part of the algorithm is $O(d^2\ n^2)$. Overall, the complexity of the $LT$-tree based algorithm is thus $O(d^2\ n^{2.5})$.

**Allowing Nodes with No Leaves** It can be helpful in practice to allow some intermediate nodes in a $LT$-tree to bear no leaves. For example, this gives the algorithm the freedom to generate a sequence of buffers of increasing sizes to drive large loads when needed, as for example at the root of a two-level tree. This can be implemented as a direct extension of the dynamic programming algorithm of Figure 3.12, by computing, for each triplet $(k,g,l)$, with $0 \leq l < L$ the optimal $LT$-tree $T_{k,g,l}$ for sinks $k$ to $n$ with source $g$ and $l$ or less intermediate nodes with no leaves connected to them. $T_{k,g,l}$ can only be composed of a buffer attached to the tree $T_{k,g,l-1}$ or a buffer attached to the tree $T_{k',g,l}$ for some $k' > k$. If

we allow up to $L-1$ intermediate nodes to bear no leaves, the complexity of the algorithm becomes $O(L\ d^2\ n^{2.5})$.

**Minimization of Area under a Delay Constraint** The $LT$-tree based algorithm can be modified to minimize area under a delay constraint. To do so, we use the same technique as the one we used on tree covering in chapter 2. The only important difference between $LT$-tree based fanout optimization and tree covering is that the role of required times and arrival times is reversed. The modified algorithm works as follows. Instead of selecting a minimum delay $LT$ tree for every pair $(k,g)$, we select a minimum cost $LT$-tree $T_{k,g,a}$ for every triplet $(k,g,a)$, where $a$ is an arrival time. The cost of each tree evaluated by the algorithm is of the form $(A,r)$, where $A$ is the area of the tree and $r$ the required time at the source of the tree. When two trees are compared, the tree with smaller area is selected, unless its required time is smaller than the arrival time $a$, in which case the tree with larger required time is selected.

To make this algorithm practical, arrival times need to be discretized. If $\tau$ is the number of discretization intervals used for arrival times, the complexity of the algorithm becomes $O(\tau\ d^2\ n^{2.5})$. Better results can be obtained by discretizing the values of $a$ in $(k,g,a)$ within bounds dependent on the value of $k$ and $g$. That technique was also described in chapter 2 and can be applied here without modification.

### 3.4.7 Other Fanout Algorithms

Other fanout optimizations have been proposed, by Berman *et al.* in [5] and Singh *et al.* in [40]. None of these algorithms are optimal, since they all rely, as the $LT$-tree algorithm does for $LT$-trees of type 1, on ordering sinks by required times. All of these algorithms produce trees that have the following property: there exists a depth-first search order of the nodes of the tree that visits the leaves in order of increasing required times. We give in Figure 3.14 an example of a fanout problem that cannot be solved optimally with such trees, even with the simple unit-fanout delay model of section 3.3.2. It is to be noted that the suboptimality of these algorithms has nothing to do with the fact that the fanout problem is NP-complete for some delay models. It is actually not known whether the fanout problem is suboptimal for the unit-delay model.

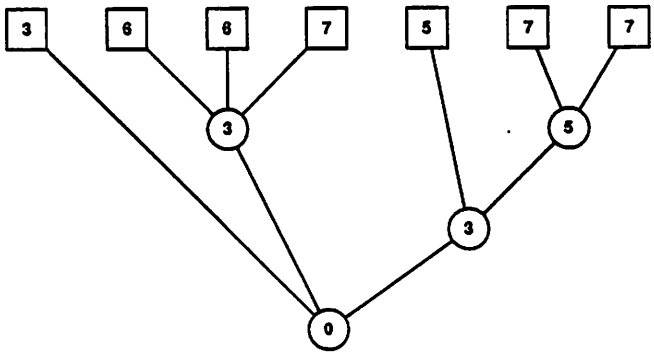


Figure 3.14: Fanout Problem Unsolvable with Consecutive Sink Ordering

**Berman's Algorithms** Berman *et al.* presented two algorithms for the fanout problem. One complex algorithm, also based on dynamic programming, which can be seen as a generalization of our $LT$-tree based algorithm, and one much simpler algorithm, called the *two-group* algorithm.

The $LT$-tree algorithm only allows trees that have at most one buffer in the fanout of any buffer. Since this condition is too restrictive to produce good quality results, we have relaxed it somewhat by allowing fanout trees to contain one balanced fanout tree as a subtree if it helps reducing delay. Berman's algorithm relaxes the restriction of having at most one buffer in the fanout of any buffer using a different technique. The algorithm allows any number of buffers in the fanout of a buffer, from 1 to some limit $k$, and uses dynamic programming to select this number optimally. By restricting the sinks to be ordered by increasing required times, the dynamic programming algorithm can be made polynomial, though with a large exponent: $O(kn^3)$, if we ignore buffer selection. If we want to take buffer selection into account with our delay model, a direct modification of this algorithm yields a complexity $O(n^2\ d^k\ (n+d))$. The $d^k$ term comes from the fact that we have $d^k$ possible ways of assigning buffers to $k$ inputs. If $k = 1$, we obtain the bound $O(n^2\ d^2)$ because the term $n^2\ d^k\ n$ disappears in that case. This is the complexity of the $LT$-tree based algorithm if we do not use two-level trees.

The two-group algorithm algorithm was introduced by Berman *et al.* as a more practical alternative to fanout optimization. This very simple algorithm tries all possible

decomposition of the sinks into two groups of sinks with consecutive required times, and, for each group, supposes that all required times and all loads are equal to select a balanced tree out of a set of precomputed trees. The precomputation can be done once and for all for a given library, so the run time of the two-group algorithm is essentially $O(n)$. This simple algorithm is fast but unlikely to generate results of the same quality as the algorithms proposed earlier.

**Singh's Algorithm** Singh's algorithm [40] uses a divide-and-conquer strategy in which the set of fanout signals is partitioned into subsets, and the process is recursively applied on the smaller problems. It can be seen as the recursive application of the two-group algorithm of Berman *et al.*, and bears some similarity with the technique used by Paulin *et al.* to decompose gates with large fanins [35]. Singh's algorithm builds a fanout tree in a top-down fashion, in contrast with combinational merging, which works bottom-up. The partitioning of the fanout signals is based on a greedy procedure that determines the kind of re-distribution of the fanout signals (based on the required times and load distributions) that results in the greatest saving at the current step. The algorithm is able to generate a balanced fanout tree if the signals are required at similar times and a skewed tree if the signals are required at widely different times. The complexity of this algorithm is $O(d^2\ n^2)$.

## 3.5 Handling Differing Polarities

**Handling Inverting Buffers** In some technologies, like CMOS standard cells, noninverting buffers are made of a juxtaposition of two inverting buffers. For delay optimization, it is thus always preferable to use inverting buffers. Inverting buffers offer more possibilities for optimization, and in the worst case can always be combined to reproduce the delay characteristics of noninverting buffers. There is no major difficulty in handling inverting buffers. We simply need to keep track of the polarity of the signal at the output of a buffer and accept a connection with a sink only if the polarities match.

**Sinks Required under Both Polarities** In real circuits, a signal is often needed under both polarities. It is possible to extend all previous algorithms to handle this situation simply by separating the sinks into two groups, one for each polarity, and using the source of the signal to distribute the signal to one group, and an inverter connected to the source to

distribute the signal to the other group. When inverting buffers are used, we do not specify which group of sinks is directly connected to the source, since connecting the source to the sinks with inverted polarities may yield a faster solution. We actually try both assignments and keep the best solution.

**Treating Both Polarities Simultaneously** The *LT*-tree based algorithm and Singh's fanout algorithm can handle signals of differing polarities directly, at an increased computational cost. This approach yields in general better results that the technique suggested in the previous section but is too expensive to be applied to large fanout problems.

The *LT*-tree algorithm only needs an extra index variable to keep track of positive and negative sinks independently. Since the two-level algorithms cannot handle sinks of different polarities, the precomputation of two-level trees is done independently, for a total cost of $O(d^2 \max(n,p)^{2.5})$, where $p$ is the number of sinks of positive phase and $n$ the number of sinks of negative phase. The cost of the rest of the computation is $O(d^2\ np \max(n,p))$, for a total cost of $O(d^2\ \max(n,p) \max(np, \max(n,p)^{1.5}))$. When treating sinks of different polarities simultaneously, Singh's algorithm becomes $O(d^3\ n^2\ p^2)$ [40].

## 3.6 Peephole Optimizations for Area and Delay

### 3.6.1 Motivation

Many local optimizations on fanout trees can be performed independently of the fanout algorithm used to produce them. Implementing these optimizations as a postprocessing phase on fanout trees has several advantages:

- the local optimizations only have to be implemented once, instead of once per fanout algorithm.
- the fanout algorithms can be made simpler. This is particularly true for area minimization under a delay constraint, which can be done quite effectively by local optimization. That way, the fanout algorithms can simply be implemented for delay minimization only.
- good quality results can be obtained efficiently by the combined effect of a simple and fast fanout algorithm and a simple and fast local optimization algorithm.

This corresponds to a general approach for solving NP-complete problems approximately that has often be found effective in practice: first computing an initial solution with a global greedy algorithm, and then improving this solution with local optimizations. This organization is commonly used in optimizing compilers. It is also at the basis of the best known heuristics for combinational problems such as the Traveling Salesman Problem.

In the rest of this section we first present an optimal algorithm to perform buffer selection on a fanout tree, under the constraint that the topological structure of the tree remains unchanged. This optimization is useful after all our fanout optimization algorithms, since none guarantee optimal gate selection. Then we present several local optimizations to decrease the area of a fanout tree under a delay constraint.

### 3.6.2 Optimal Buffer Selection

Fanout algorithms do not guarantee in general that the buffers in the fanout trees they produce are selected optimally. For example, the two-level algorithm enforces all intermediate buffers to be of the same type which is not necessarily the best possible solution; the combinational merging algorithm selects buffers heuristically, without complete knowledge of the local structure of the tree in which the buffer is inserted. By performing optimal buffer selection on the fanout trees resulting from these algorithms, we can decrease delay at times significantly. In addition, the computational cost of performing optimal gate selection is very low: $O(d^2\ m)$, where $m$ is the number of edges in the tree, and $d$ the number of buffers in the library. The cost of performing optimal gate selection is thus only a fraction of the cost of building a fanout tree in the first place. Thus there is no reason not to perform optimal gate selection on every fanout tree.

Optimal buffer selection can be implemented with a simple algorithm that proceeds from the sinks to the root of a fanout tree, and selects, at each intermediate node, a buffer that maximizes the required time at the parent of that node. In contrast with tree covering, it is not enough to simply select a buffer that maximizes the required time at a node, because a later required time for a subtree usually means a higher load to drive for that subtree. Selecting the largest possible required time for a subtree can slow down the signal going to the other subtrees sharing the same parent. It is possible, using dynamic programming, to compute for each subtree the required time $required[b]$ achievable by using a buffer of type $b$ at the root of the subtree. To do so, we proceed as follows.

Let $v$ be an intermediate node of the tree, with nodes $(v_1, \ldots, v_n)$ as children. For each buffer $b \in \mathcal{L}$, we have already computed by induction the best required times $required[v_i, b]$ achievable for the subtrees rooted at $v_i$, $1 \leq i \leq n$ if $b$ is selected at node $v_i$. To compute $required[v, b]$, we simply need to find an assignment of buffers to nodes $(v_1, \ldots, v_n)$, i.e. a function $f : \{1, \ldots, n\} \rightarrow \{1, \ldots, d\}$, that maximizes the required time at $v$, given by the following expression:

$$required[v, b] = \min_{1 \leq i \leq n} required[v_i, b_f(i)] - \beta_b \sum_{1 \leq i \leq n} \gamma_{b_f(i)} - \alpha_b \qquad (3.27)$$

To compute $f$ by exhaustive enumeration would require $O(d^n)$ operations. It is not uncommon to have libraries with $d$ of the order of 10, so this brute force approach is not practical. In the next subsection, we present an $O(d\,n)$ algorithm to compute the optimal solution of equation 3.27. To compute gate selection for an entire tree, we need to apply this algorithm $d$ times to each intermediate node, for a total cost of $O(d^2 m)$, where $m$ is the number of edges in the fanout tree. The number of edges in a tree is equal to the number of nodes of the tree minus one.

### A Fast Buffer Assignment Algorithm

This assignment problem we are attempting to solve can be reformulated as follows: given two $n \times d$ matrix of numbers $r_{i,j}$ and $l_{i,j}$ such that, for any given $i$, $r_{i,j}$ and $l_{i,j}$ are monotonically non decreasing in $j$, find an assignment $f : \{1, \ldots, n\} \rightarrow \{1, \ldots, d\}$ that maximizes the quantity:

$$gain(f) = r_f - l_f \quad \text{where} \qquad (3.28)$$

$$r_f = \min_{1 \leq i \leq n} r_{i,f(i)} \qquad (3.29)$$

$$l_f = \sum_{1 \leq i \leq n} l_{i,f(i)} \qquad (3.30)$$

The $r_{i,j}$ represent the required times $required[v_i, b_j]$ and the $l_{i,j}$ the load values $\gamma_{b_j}$. The load values are actually independent of $i$, and, without loss of generality, we can suppose that they are sorted in increasing order. Thus the $l_{i,j}$ are monotonically non decreasing in $j$. If the $r_{i,j}$ are not monotonically non decreasing in $j$, there would be a pair $(j_1, j_2)$ of indices such that, for a some $i$, $l_{i,j_1} \leq l_{i,j_2}$ and $r_{i,j_1} > r_{i,j_2}$. For $i$, $j_1$ would always be a superior choice than $j_2$. We can therefore replace $r_{i,j_2}$ by $r_{i,j_1}$ and $l_{i,j_2}$ by $l_{i,j_1}$ without

affecting at least one optimal assignment to the problem. We can iterate this replacement until the monotonicity condition is satisfied by the $r_{i,j}$.

We define a distance between two assignments $f$ and $g$ as follows:

$$d(f,g) = \sum_{1 \leq i \leq n} |f(i) - g(i)| \tag{3.31}$$

In particular, $d(f,g) = 1$ if and only if $f$ and $g$ differ on only one index, and the difference of value on this index is only one.

The optimum assignment algorithm is outlined in Figure 3.15. The algorithm computes a sequence of assignments $(f_k)_{1 \leq k \leq K}$, for some $K \leq dn$, such that $d(f_{k+1}, f_k) = 1$, and such that there is a $k_0$, $0 \leq k_0 \leq K$ for which $f_{k_0}$ is optimal. $f_0$ is initialized to be such that $f_0(i) = 1$ for all $1 \leq i \leq n$. Given $f_k$, the computation of $f_{k+1}$ only takes constant time. To compute $f_{k+1}$ from $f_k$, the algorithm finds an index $i_k$ that is critical for the current assignment, i.e. an index that minimizes the quantity $r_{i,f_k(i)}$ for $1 \leq i \leq n$. $f_{k+1}$ is then defined as being equal to $f_k$ for all indices different from $i_k$ and equal to $f_k(i_k) + 1$ on $i_k$. If $f_k(i_k) = d$ and thus cannot be incremented, the algorithm terminates the computation of the sequence $(f_k)$. The total number of incrementing steps cannot exceed $dn$.

To find the optimal assignment $f_{k_0}$, the algorithm works backwards, from $f_K$ to $f_0$, in order to exploit the fact that the cost of assignment $f_k$, $gain(f_k)$, can be computed in constant time from the cost of $f_{k+1}$, but not conversely. To compute $gain(f_k)$ in constant time from $gain(f_{k+1})$, the algorithm keeps track of the intermediate quantities $r_{f_k} = \min_{1 \leq i \leq n} r_{i,f_k(i)}$ and $l_{f_k} = \sum_{1 \leq i \leq n} l_{i,f_k(i)}$, and use the fact that $r_{f_k} = r_{i_k,f_k(i_k)}$ and $l_{f_k} = l_{f_{k+1}} - r_{i_k,f_k(i_k)+1} + r_{i_k,f_k(i_k)}$. This guarantees that $k_0$ can be computed in $O(nd)$ time. Finally constructing $f_{k_0}$ from $f_0$ given $k_0$ takes $O(nd)$ time.

**Theorem 3.6.1** *The algorithm of Figure 3.15 is optimum.*

The proof of the theorem relies on the following lemma:

**lemma 3.6.2** *Let $f$ and $g$ be two assignments, such that:*

$$f(i) \leq g(i) \qquad \text{for all } 1 \leq i \leq n \tag{3.32}$$

$$f(i_0) = g(i_0) \qquad \text{for } i_0 = \arg \min_{1 \leq i \leq n} r_{i,f(i)} \tag{3.33}$$

*Then $gain(f) \geq gain(g)$.*

```
algorithm optimal_buffer_assignment
    set f to be such that: f(i) = 1 for all 1 ≤ i ≤ n.
    k = 1;
    do {
        i_0 = arg min_{1≤i≤n} r_{i,f(i)}
        if f(i_0) = d break
        f(i_0) = f(i_0) + 1;  i[k] = i_0;  k = k + 1;
    }
    K = k − 1;  k_0 = K;
    r = min_{1≤i≤n} r_{i,f(i)};
    l = Σ_{1≤i≤n} l_{i,f(i)};
    gain = r − l;
    for k = K − 1 to 0 {
        f(i[k]) = f(i[k]) − 1;
        r = r_{i[k],f(i[k])};  l = l − l_{i[k],f(i[k])+1} + l_{i[k],f(i[k])};
        if (r − l > gain) {
            k_0 = k;  gain = r − l;
        }
    }
    for k = 1 to k_0    f(i_k) = f(i_k) + 1;
end optimal_buffer_assignment
```

Figure 3.15: An Optimum Assignment Algorithm

**Proof** By the monotonicity hypothesis on the arrays $r_{i,j}$ and $l_{i,j}$, (3.32) implies that $r_f \leq r_g$ and $l_f \leq l_g$, and (3.33) implies that $r_f = r_{i_0,f(i_0)} = r_{i_0,g(i_0)} \geq r_g$. Thus $r_f = r_g$ and $l_f \leq l_g$, which implies that $gain(f) \geq gain(g)$. ∎

**Proof** [of theorem 3.6.1]

We only have to prove that one of the assignments $(f_k)_{0 \leq k \leq K}$ produced by the algorithm is optimal. Let $g$ be an optimal assignment, and let $I_k = \{i | 1 \leq i \leq n, g(i) < f_k(i)\}$. Let $k_0$ be the largest index for which $I_k = \emptyset$. We will use the previous lemma to prove that $gain(f_{k_0}) \geq gain(g)$.

Since $I_{k_0} = \emptyset$, $f_{k_0}(i) \leq g(i)$ for all $1 \leq i \leq n$, thus condition (3.32) is satisfied. If

$k_0 = K$ we know that there is an index $i_0$ such that $f_K(i_0) = d$ and $r_{i_0,d} = r_{f_K}$, since that is the termination condition of the first phase of the algorithm. Thus $i_0 = \arg\min_{1 \le i \le n} r_{i,f_K(i)}$. Moreover, since $f_K(i_0) \le g(i_0)$ and $f_K(i_0) = d$, we have $f_K(i_0) = g(i_0)$, which proves that condition (3.33) is satisfied. If $k_0 < K$, we know that there is an index $i_0$ such that $i_0 = \arg\min_{1 \le i \le n} r_{i,f_{k_0}}(i)$ and such that $f_{k_0+1}(i) = f_{k_0}(i)$ if $i \neq i_0$ and $f_{k_0+1}(i_0) = f_{k_0}(i_0) + 1$. Since $I_{k_0}$ is empty, $I_{k_0+1}$ contains at most one element: $i_0$. Since $k_0$ is the largest index for which $I_k$ is empty, we have $I_{k_0+1} \neq \emptyset$. Thus $I_{k_0+1} = \{i_0\}$. Consequently $f_{k_0}(i_0) + 1 = f_{k_0+1}(i_0) > g(i_0) \ge f_{k_0}(i_0)$. This proves that $f_{k_0}(i_0) = g(i_0)$, and thus that condition (3.33) of the lemma is satisfied. ∎

### 3.6.3 Area Recovery under a Delay Constraint

If the fanout problem is given with a delay constraint, our objective is to find a fanout tree with minimum area that meets the delay constraint. The fanout algorithms we have presented so far can only minimize delay. We have suggested in section 3.4.6 a way to extend the *LT*-tree based algorithm to minimize area under a delay constraint. Other fanout algorithms could also be extended to support this optimization, but each algorithm will have to be modified separately (e.g. the *LT*-tree algorithm is the only one to be based on dynamic programming). In addition, it is likely to be difficult to extend the fanout algorithms to minimize area under a delay constraint in an efficient and accurate way. Fortunately, there are simpler ways to recover area. Though not optimal, the techniques we present in the following paragraphs are very effective in practice and straightforward to implement.

**Selecting the Best Fanout Tree** We have at our disposal several fanout optimization algorithms, based on inverter selection, two-level trees and *LT*-trees. Even if used for minimizing delays, these algorithms are going to produce fanout trees of differing area and delay. Our first area recovery technique simply consists in computing the fanout trees generated by all fanout algorithms at our disposal, and selecting the minimum area tree that meets the delay requirement. This technique is effective because the two-level algorithm often generates trees that are fast enough and usually smaller than the trees produced by the *LT*-tree algorithm. In addition, this technique also detects the case where the delay constraint can be met with no buffering at all.

**Partial Collapse** The second heuristic we use, which is also very effective at recovering unnecessary area, consists in partially collapsing minimum delay fanout trees. This technique is particularly useful on $LT$-trees or trees generated by combinational merging. The algorithm performs partial collapses as follows. Given a fanout tree and a delay constraint, i.e. an arrival time at the root of the fanout tree, the tree is visited from the root to the sinks in order to compute the arrival times at every intermediate node. Then the tree is visited in reverse order, from the sinks to the root. At each intermediate node $v$, the algorithm computes the required time at $v$ that would be obtained if $v$ were connected directly to all the sinks driven by the subtree $T_v$ rooted at $v$. If this required time is larger than the arrival time at $v$, $T_v$ is collapsed into $v$: all buffers contained in $T_v$ are eliminated and the sinks of $T_v$ are directly connected to $v$.

**Buffer Selection** We also use a modified version of the optimal buffer selection algorithm of section 3.6.2. This algorithm is not optimal: it does not find the buffer assignment that would minimize area under a delay constraint, but is fast, simple and easy to implement. The modification is done as follows: given an arrival time at the root of the tree, for each intermediate node $v$ and for each buffer type $b$, we want to compute an achievable arrival time at $v$ if node $v$ is assigned a buffer of type $b$. This arrival time is taken to be the arrival time obtained on the fanout tree after the buffer at node $v$ has been replaced by $b$, without changing the rest of the tree; it is clearly an achievable, but not necessarily an optimal arrival time at node $v$ with buffer $b$. The suboptimality comes from the fact that we do not know what is the optimal buffer assignment for the siblings of $v$ given that $v$ is assigned $b$, and it would be too time consuming to compute it for all values of $b$. Given this achievable arrival time at $v$ for a given choice of buffer $b$, to perform buffer selection we use the algorithm of section 3.6.2, modified to select the minimum index $k$ for which $f_k$ guarantees a nonnegative slack at $v$. This computation is done for each value of $b$ and the result is stored at node $v$. The buffer selection for node $v$ is done when the parent node of $v$ is visited.

## 3.7 Global Fanout Optimization

We have only discussed so far algorithms to optimize a given fanout problem. It is time to introduce a technique that can be used to apply a fanout algorithm to an entire

```
algorithm one_pass_global_fanout_optimization
    foreach node v visited in topological order from outputs to inputs {
    if v is the root of a tree {
        apply fanout optimization to the fanout problem rooted at v
    } else {
        propagate the required time at the output of v to the inputs of v
    }
end one_pass_global_fanout_optimization
```

Figure 3.16: A One-Pass Optimal Global Fanout Algorithm

circuit. In section 3.7.1 we present such a technique and show that it is optimal with respect to delay minimization. In section 3.7.2 we extend this technique to recover area under a delay constraint. For area minimization under a delay constraint, this technique is not optimal but is very efficient in practice.

### 3.7.1 A One Pass Approach

To apply a fanout algorithm to an entire circuit, we use a simple procedure introduced by Hoover *et al.* [22]. This procedure, described in Figure 3.16 and illustrated in Figure 3.17, consists in visiting the nodes of a circuit in topological order, and at each node $v$ doing the following:

- if $v$ is the root of a tree, $v$ is also the root of a fanout problem. In that case, we replace the existing fanout tree rooted at $v$ by the result of the fanout optimization algorithm applied at the fanout problem rooted at $v$.

- otherwise, we simply propagate the required times of the outputs of $v$ to the input of $v$. More specifically, if $(v_1, \ldots, v_n)$ are the fanouts of $v$, the required time $r$ at the output of $v$ is the minimum of the required times of $(v_1, \ldots, v_n)$; the load $\gamma$ at the output of $v$ is the sum of the loads of $(v_1, \ldots, v_n)$. The required time $r_i$ at input pin $i$ of node $v$ is then given by the following equation: $r_i = r - \alpha_i - \beta_i\gamma$, where $\alpha_i$ is the intrinsic delay and $\beta_i$ the load dependent delay of the gate at node $v$ for input pin $i$.

The algorithm of Figure 3.16 is optimal in the following sense. Let $N$ be a combinational network, $L = (v_1, \ldots, v_m)$ be an arbitrary list of possibly repeated tree roots of

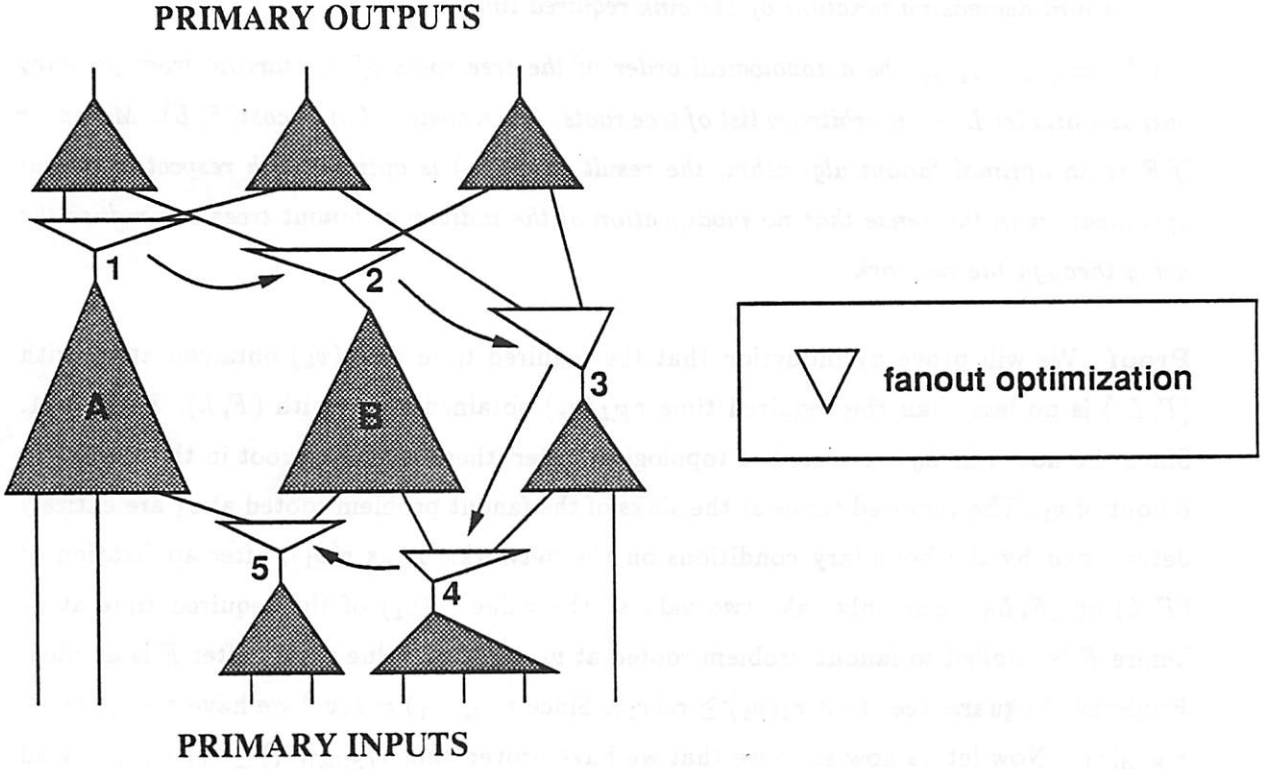


Figure 3.17: Applying Fanout Algorithms in One Pass From Outputs to Inputs

$N$ and $F$ a fanout algorithm. The pair $(F, L)$ defines a global fanout algorithm as follows: first, compute the required times at all nodes in $N$. Then, for $k = 1$ to $k = m$, apply the algorithm $F$ to the fanout problem rooted at $v_k$ and recompute the required times at all nodes in $N$ wherever necessary. We define the cost of $(F, L)$ as a n-dimensional vector:

$$cost(L, F) \quad = \quad \{r(p_i), 1 \leq i \leq n\} \tag{3.34}$$

where the nodes $p_i$, $1 \leq i \leq n$, are the primary inputs of $N$, and $r(v)$ designates the required time at node $v$. We say that the n-dimensional vectors $x$ and $y$ satisfy the inequality $x \leq y$ if and only if $x_i \leq y_i$ for all $1 \leq i \leq n$.

**Theorem 3.7.1** *Let $F$ be a fanout algorithm with the following two properties:*

1. *it never replaces a fanout tree by one with worse delay.*

2. *the fanout trees it produces are such that the required times at the root of the trees are a non decreasing function of the sink required times.*

*Let $L_0 = (v_1, \ldots, v_m)$ be a topological order of the tree roots of $N$ starting from primary outputs, and let $L$ be an arbitrary list of tree roots. Then $cost(F, L_0) \geq cost(F, L)$. Moreover if $F$ is an optimal fanout algorithm, the result of $(F, L_0)$ is optimal with respect to fanout optimization in the sense that no modification of the individual fanout trees can reduce the delay through the network.*

**Proof** We will prove by induction that the required time $r_{F,L_0}(v_k)$ obtained at $v_k$ with $(F, L_0)$ is no less than the required time $r_{F,L}(v_k)$ obtained at $v_k$ with $(F, L)$. Let $k = 1$. Since the nodes in $L_0$ are sorted in topological order, there is no tree root in the transitive fanout of $v_1$. The required times at the sinks of the fanout problem rooted at $v_1$ are entirely determined by the boundary conditions on the network. Thus $r(v_1)$, after application of $(F, L)$ or $(F, L_0)$, can only take two values: the value $r_0(v_1)$ of the required time at $v_1$ before $F$ is applied to fanout problem rooted at $v_1$, and the value $r_1(v_1)$ after $F$ is applied. Property (1) guarantees that $r_1(v_1) \geq r_0(v_1)$. Since $r_{F,L_0}(v_1) = r(v_1)$ we have $r_{(F,L_0)}(v_1) \geq r_{(F,L)}(v_1)$. Now let us now suppose that we have proved that $r_{(F,L_0)}(v_i) \geq r_{(F,L)}(v_i)$ for all $1 \leq i \leq k$. Since $F$ can only increase the required time at a node, $r_{F,L_0}(v_i)$ is the largest required time at node $v_i$, $1 \leq i \leq k$ observed at any intermediate step of computation of $(F, L)$ or $(F, L_0)$. Since the required times at the sinks of node $v_{k+1}$ are monotonic non decreasing functions of the required times $r(v_i)$ for $1 \leq i \leq k$, and do not depend on the required times at nodes $v_i$ for $i > k + 1$, the fanout problem rooted at node $v_{k+1}$ is given with the largest required times observed during the execution of $(F, L)$ or $(F, L_0)$ when $r(v_i) = r_{F,L_0}(v_i)$ for $1 \leq i \leq k$. From this fact and property (2), we deduce that $r_{(F,L_0)}(v_{k+1}) \geq r_{(F,L)}(v_{k+1})$, which proves the first part of the theorem.

The second part of the theorem is proved in a similar fashion. Let $T$ be a circuit, $T_{opt}$ a version of $T$ that is optimal with respect to fanout optimization, and let $T_F$ be the result of applying to $T$ an optimal fanout algorithm $F$ to all tree roots in topological order. Since only fanout optimization has been performed in both cases, $T_{opt}$ and $T_F$ have the same tree roots. We will prove by induction that $r_{T_{opt}}(v_k) \leq r_{T_F}(v_k)$ for all $1 \leq k \leq m$, where $(v_1, \ldots, v_m)_{1 \leq i \leq m}$ is a topological ordering of the tree roots. The result is true for $k = 1$, since the required times at the sinks of $v_1$ are the same in both trees, and $F$ is an optimal fanout algorithm. Let us suppose that the result holds for $1 \leq i \leq k$. By induction

hypothesis, the required times at the sinks of tree root $v_{k+1}$ are no worse in $T_F$ than in $T_{opt}$. Since $F$ is optimal, $r_{T_{opt}}(v_{k+1}) \leq r_{T_F}(v_{k+1})$. ■

The importance of theorem 3.7.1 is to prove that any lack of optimality in fanout optimization is due to the lack of optimality of the fanout optimization algorithms, not to the procedure we use to apply the fanout algorithms to the entire circuit. In particular, there is no need to develop fast, incremental techniques to extract critical path information that would be used to guide the fanout optimization. This simple algorithm based on one topological traversal does as well in terms of delay as any other more complicated technique.

**Duality with Tree Covering** An interesting question to ask at this point is why a similar algorithm of applying tree covering in topological order, this time from inputs to outputs, is not optimal for tree covering. The reason is that tree covers for trees that are on disjoint paths from primary inputs to primary outputs do interact, which is not the case for fanout trees. For example, in Figure 3.17, the choice of a cover for tree A influences the required times at the sinks of fanout tree 5, and thus the delay through fanout tree 5, and thus the arrival times at tree B. There is no such coupling between the solutions of fanout problems that are not on a common path from inputs to outputs. This why theorem 3.7.1 holds for fanout optimization and the equivalent theorem does not hold for tree covering.

### 3.7.2 Global Area Recovery under a Delay Constraint

The main weakness of the optimal procedure described in the previous section is that it can be too wasteful in area. It optimizes all tree roots for minimum delay, with no consideration for the necessity of such an optimization. We now introduce a technique to recover area at no cost in delay. This technique works with arbitrary required times at the primary outputs and arbitrary arrival times at the primary inputs of a network.

To recover area at no delay cost after fanout optimization, we proceed as follows. We first save at each tree root the arrival time achievable after fanout optimization. We then reapply the fanout optimizer to each fanout problem, visited in topological order. This time we call the fanout optimizer to minimize fanout tree area under the constraint that the required time at the root of the tree is no less than the arrival time at the root of the tree. To perform this optimization, we use the techniques described in section 3.6.3.

With this simple algorithm, we can recover, at no delay cost, most of the area wasted by the first phase of fanout optimization. Unfortunately this algorithm is not opti-

mal, for two reasons: first because it relies on a fanout algorithm that itself is not optimal; second because by visiting nodes in topological order, it uses the slack available on any given path as early as possible. A more equilibrated use of the slack can lead, at least in some cases, to a smaller circuit with the same delay. Despite these limitations, this technique is very effective at recovering area, as can be seen from the results of section 3.8.3.

## 3.8 Experimental Results

To provide some experimental evidence of the efficiency of fanout optimization, we gathered a set of 25 benchmark circuits from several origins. These circuits are relatively large, ranging from 119 gates to 2557 gates, and are described in more detail in section 3.8.1. We present overall performance results in section 3.8.2 and a more detailed analysis of the effect of various optimizations in section 3.8.3.

### 3.8.1 Circuit Descriptions

Our set of benchmark circuits come from four sources: MCNC, ISCAS, Intel and AT&T. The MCNC benchmarks were put together during the International Logic Synthesis Workshop 1989 [32] and are publicly available from MCNC. They are themselves from several origins, though complete information was not always available on each of them. The ISCAS benchmarks were originally testing benchmarks. The Intel and AT&T circuits come from these companies. No details concerning their functionality were provided. Table 3.2 contains some general information on the 25 benchmark circuits, including the number of primary inputs and primary outputs, the number of literals in factored form, and the number of gates needed to implement the circuits when technology mapped for minimum area using the MCNC library `lib2`. We also indicate briefly the function of the circuit if known. If it is not known, we simply used the word *logic* to characterize the circuit.

### 3.8.2 Overall Performance of Fanout Optimization

We measured the effect of applying fanout optimization to our set of benchmark circuits after the circuits were mapped for minimum area. The effect of combining minimum delay tree covering with fanout optimization will be discussed in the next chapter.

| circuit | circuit function | origin | # pis | # pos | # lits | # gates |
|---|---|---|---|---|---|---|
| C1355 | error correcting | ISCAS | 41 | 32 | 1064 | 510 |
| C1908 | error correcting | ISCAS | 33 | 25 | 1497 | 349 |
| C2670 | ALU and control | ISCAS | 233 | 140 | 2075 | 505 |
| C3540 | ALU and control | ISCAS | 50 | 22 | 2936 | 740 |
| C5315 | ALU and selector | ISCAS | 178 | 123 | 4386 | 1080 |
| C6288 | 16-bit multiplier | ISCAS | 32 | 32 | 4800 | 2371 |
| C7552 | ALU and control | ISCAS | 207 | 108 | 6144 | 1688 |
| alu4 | logic | MCNC | 14 | 8 | 1278 | 418 |
| ampbpreg | logic | AT&T | 117 | 88 | 3318 | 1137 |
| ampbsm | logic | AT&T | 75 | 66 | 2578 | 795 |
| amppint2 | logic | AT&T | 85 | 66 | 3372 | 513 |
| ampxhdl | logic | AT&T | 62 | 40 | 3742 | 365 |
| apex6 | logic | MCNC | 135 | 99 | 904 | 438 |
| des | data encryption | MCNC | 256 | 245 | 6346 | 2557 |
| dflgrcb1 | logic | Intel | 108 | 65 | 623 | 179 |
| fconrcb1 | logic | Intel | 62 | 35 | 459 | 129 |
| frg2 | logic | MCNC | 143 | 139 | 2014 | 727 |
| k2 | logic | MCNC | 45 | 45 | 2930 | 1172 |
| kcctlcb3 | logic | Intel | 81 | 44 | 415 | 137 |
| pair | logic | MCNC | 173 | 137 | 2426 | 944 |
| rot | logic | MCNC | 135 | 107 | 869 | 403 |
| sbiucb1 | logic | Intel | 40 | 35 | 591 | 144 |
| tfaultcb1 | logic | Intel | 77 | 35 | 659 | 119 |
| vda | logic | MCNC | 17 | 39 | 1423 | 586 |
| x3 | logic | MCNC | 135 | 99 | 1345 | 486 |

Table 3.2: General Information on the Benchmark Set

**circuit function:** simple description of the logic function of the circuit, if available
**origin:** origin of the circuit
**# pis:** number of primary inputs
**# pos:** number of primary outputs
**# lits:** number of literals in factored form as computed by misII
**# gates:** number of gates to implement the circuit using misII technology mapper in minimum area mode with the MCNC library lib2

| circuit | min area | | fanout opt | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 990 | 27.16 | 1119 | 24.25 | 1.13 | 0.89 |
| C1908 | 1086 | 35.04 | 1236 | 29.55 | 1.14 | 0.84 |
| C2670 | 1420 | 28.42 | 1478 | 22.14 | 1.04 | 0.78 |
| C3540 | 2201 | 45.24 | 2421 | 34.27 | 1.10 | 0.76 |
| C5315 | 3171 | 39.94 | 3352 | 30.78 | 1.06 | 0.77 |
| C6288 | 6777 | 120.37 | 7340 | 101.08 | 1.08 | 0.84 |
| C7552 | 4660 | 69.47 | 5084 | 28.55 | 1.09 | 0.41 |
| alu4 | 1486 | 47.17 | 1724 | 31.84 | 1.16 | 0.68 |
| ampbpreg | 2741 | 59.85 | 2891 | 39.40 | 1.05 | 0.66 |
| ampbsm | 1478 | 25.78 | 1615 | 18.05 | 1.09 | 0.70 |
| amppint2 | 1021 | 22.45 | 1136 | 12.31 | 1.11 | 0.55 |
| ampxhdl | 751 | 24.73 | 865 | 13.38 | 1.15 | 0.54 |
| apex6 | 1505 | 17.74 | 1565 | 13.40 | 1.04 | 0.76 |
| des | 6452 | 107.12 | 7358 | 17.84 | 1.14 | 0.17 |
| dflgrcb1 | 615 | 12.83 | 630 | 11.01 | 1.02 | 0.86 |
| fconrcb1 | 467 | 15.18 | 481 | 13.82 | 1.03 | 0.91 |
| frg2 | 1738 | 37.91 | 1893 | 14.95 | 1.09 | 0.39 |
| k2 | 1972 | 32.80 | 2189 | 20.56 | 1.11 | 0.63 |
| kcctlcb3 | 449 | 11.50 | 469 | 10.20 | 1.04 | 0.89 |
| pair | 3200 | 25.85 | 3482 | 17.80 | 1.09 | 0.69 |
| rot | 1387 | 30.18 | 1444 | 21.49 | 1.04 | 0.71 |
| sbiucb1 | 471 | 22.18 | 514 | 18.89 | 1.09 | 0.85 |
| tfaultcb1 | 375 | 9.91 | 400 | 8.41 | 1.07 | 0.85 |
| vda | 1118 | 23.76 | 1324 | 16.75 | 1.18 | 0.70 |
| x3 | 1653 | 22.40 | 1754 | 11.45 | 1.06 | 0.51 |
| aver | – | – | – | – | 1.09 | 0.66 |

Table 3.3: Effect of Fanout Optimization on Circuits Optimized by misII

**min area:** minimum area technology mapping with MCNC library *lib2*
**fanout opt:** minimum area technology mapping followed by fanout optimization
**gain:** gain in area or in delay obtained by using fanout optimization
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

**Effect on Optimized Circuits** Before technology mapping, we optimized each circuit using `misII` technology independent simplification and factorization algorithms, as is normally done in logic synthesis. We then compared the area and delay of the circuits after technology mapping for minimum area, with and without fanout optimization. The results are reported in Table 3.3. We can observe a wide variation in delay reductions, ranging from 9% to a factor of 6, while the area increases ranged from 3% to 18%. On average, fanout optimization reduces delay by 34% for a area increase of 9%.

**Effect on Unoptimized Circuits** Logic simplification is usually beneficial both in terms of area and delay, while logic factorization can decrease circuit area by sharing common subexpressions at the cost of larger fanouts and extra levels of logic. Because we ran our experiments on circuits that were optimized by `misII`, there is the possibility that fanout optimization was effective on these circuits simply because it corrects the fanouts introduced by factorization. To check for this possibility, we ran the same experiments on the same circuits, but this time before the circuits were optimized by `misII`. The results are reported in Table 3.4. Though for some circuits, such as `C1355` and `C6288`, fanout optimization was more effective on optimized circuits, on average fanout optimization was more effective on unoptimized circuits, reducing delay by 44% for an area increase of 10% instead of 34% and 9% respectively. From this experiment we can conclude that factorization is not the main factor contributing to large fanouts. Actually, technology independent optimization can have the overall effect of reducing the impact of fanout optimization.

### 3.8.3 Detailed Performance Analysis

In this section, we perform a more detailed analysis of our fanout optimization algorithms. We first describe the effect of inverter sizing, then the effect of buffering with no critical signal isolation, the effect of peephole optimization and the effect of area recovery. Finally we compare our fanout optimization algorithm to Singh's algorithm [40]. All experiments use the optimized version of our benchmark circuits and minimum area technology mapping with the MCNC library `lib2`.

**The Effect of Inverter Optimization** Inverter optimization is a simple optimization that consists in solving a fanout problem by introducing one inverter if there is a sink that needs the signal in a different polarity than provided by the source, and sizing this inverter

| circuit | min area | | fanout opt | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1662 | 30.59 | 1757 | 27.65 | 1.06 | 0.90 |
| C1908 | 1405 | 35.76 | 1633 | 27.10 | 1.16 | 0.76 |
| C2670 | 1928 | 43.78 | 2052 | 25.19 | 1.06 | 0.58 |
| C3540 | 2719 | 51.69 | 3106 | 36.85 | 1.14 | 0.71 |
| C5315 | 4369 | 40.60 | 4587 | 32.05 | 1.05 | 0.79 |
| C6288 | 7094 | 123.87 | 7801 | 113.28 | 1.10 | 0.91 |
| C7552 | 6386 | 72.45 | 6816 | 28.16 | 1.07 | 0.39 |
| alu4 | 1543 | 47.17 | 1807 | 30.79 | 1.17 | 0.65 |
| ampbpreg | 3901 | 101.72 | 4112 | 47.11 | 1.05 | 0.46 |
| ampbsm | 3017 | 43.40 | 3504 | 20.29 | 1.16 | 0.47 |
| amppint2 | 2232 | 31.23 | 2562 | 10.86 | 1.15 | 0.35 |
| ampxhdl | 1471 | 32.95 | 1698 | 13.02 | 1.15 | 0.40 |
| apex6 | 1548 | 15.55 | 1607 | 12.95 | 1.04 | 0.83 |
| des | 11023 | 114.56 | 12288 | 16.70 | 1.11 | 0.15 |
| dflgrcb1 | 604 | 13.05 | 634 | 9.89 | 1.05 | 0.76 |
| fconrcb1 | 467 | 15.18 | 481 | 13.82 | 1.03 | 0.91 |
| frg2 | 2949 | 52.10 | 3246 | 12.27 | 1.10 | 0.24 |
| k2 | 5055 | 48.25 | 6264 | 14.28 | 1.24 | 0.30 |
| kcctlcb3 | 457 | 11.50 | 477 | 10.20 | 1.04 | 0.89 |
| pair | 3636 | 30.93 | 3982 | 17.81 | 1.10 | 0.58 |
| rot | 1438 | 30.17 | 1495 | 21.90 | 1.04 | 0.73 |
| sbiucb1 | 505 | 22.46 | 576 | 18.90 | 1.14 | 0.84 |
| tfaultcb1 | 367 | 9.94 | 394 | 8.41 | 1.07 | 0.85 |
| vda | 2474 | 29.79 | 3185 | 11.95 | 1.29 | 0.40 |
| x3 | 2056 | 20.76 | 2138 | 11.48 | 1.04 | 0.55 |
| aver | - | - | - | - | 1.10 | 0.56 |

Table 3.4: Effect of Fanout Optimization on Unoptimized Circuits

**min area:** minimum area technology mapping with MCNC library *lib2*
**fanout opt:** minimum area technology mapping followed by fanout optimization
**gain:** gain in area or in delay obtained by using fanout optimization
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

| circuit | min area | | inv opt | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 990.00 | 27.16 | 992.00 | 26.87 | 1.00 | 0.99 |
| C1908 | 1086.00 | 35.04 | 1090.00 | 34.51 | 1.00 | 0.98 |
| C2670 | 1420.00 | 28.42 | 1422.00 | 27.30 | 1.00 | 0.96 |
| C3540 | 2201.00 | 45.24 | 2205.00 | 42.81 | 1.00 | 0.95 |
| C5315 | 3171.00 | 39.94 | 3177.00 | 38.45 | 1.00 | 0.96 |
| C6288 | 6777.00 | 120.37 | 6780.00 | 118.02 | 1.00 | 0.98 |
| C7552 | 4660.00 | 69.47 | 4661.00 | 62.83 | 1.00 | 0.90 |
| alu4 | 1486.00 | 47.17 | 1491.00 | 44.19 | 1.00 | 0.94 |
| ampbpreg | 2741.00 | 59.85 | 2743.00 | 58.88 | 1.00 | 0.98 |
| ampbsm | 1478.00 | 25.78 | 1482.00 | 23.69 | 1.00 | 0.92 |
| amppint2 | 1021.00 | 22.45 | 1023.00 | 21.69 | 1.00 | 0.97 |
| ampxhdl | 751.00 | 24.73 | 754.00 | 23.59 | 1.00 | 0.95 |
| apex6 | 1505.00 | 17.74 | 1507.00 | 15.79 | 1.00 | 0.89 |
| des | 6452.00 | 107.12 | 6453.00 | 94.97 | 1.00 | 0.89 |
| dflgrcb1 | 615.00 | 12.83 | 616.00 | 12.15 | 1.00 | 0.95 |
| fconrcb1 | 467.00 | 15.18 | 467.00 | 14.79 | 1.00 | 0.97 |
| frg2 | 1738.00 | 37.91 | 1741.00 | 36.83 | 1.00 | 0.97 |
| k2 | 1972.00 | 32.80 | 1973.00 | 31.88 | 1.00 | 0.97 |
| kcctlcb3 | 449.00 | 11.50 | 449.00 | 11.04 | 1.00 | 0.96 |
| pair | 3200.00 | 25.85 | 3207.00 | 23.58 | 1.00 | 0.91 |
| rot | 1387.00 | 30.18 | 1390.00 | 29.66 | 1.00 | 0.98 |
| sbiucb1 | 471.00 | 22.18 | 471.00 | 21.99 | 1.00 | 0.99 |
| tfaultcb1 | 375.00 | 9.91 | 376.00 | 9.70 | 1.00 | 0.98 |
| vda | 1118.00 | 23.76 | 1121.00 | 21.86 | 1.00 | 0.92 |
| x3 | 1653.00 | 22.40 | 1654.00 | 22.02 | 1.00 | 0.98 |
| aver | - | - | - | - | 1.00 | 0.95 |

Table 3.5: Effect of Inverter Optimization on Circuits Optimized by misII

**min area:** minimum area technology mapping with MCNC library *lib2*
**inv opt:** minimum area technology mapping followed by inverter optimization
**gain:** gain in area or in delay obtained by using inverter optimization
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

optimally for delay. In the area recovery phase, if the inverter is not critical for delay, it is replaced by a smaller inverter. The effect of this simple optimization is reported in Table 3.5. Inverter optimization can achieve a 5% decrease in delay at negligible cost in area. Inverter optimization can achieve 15% of the total delay decrease obtainable with fanout optimization.

**Fanout Optimization with No Critical Signal Isolation** The two main fanout optimizations combined in our algorithms are buffering and critical signal isolation. To determine what is the relative importance of these two optimizations, we measured the effect of fanout optimization limited to buffering. For the buffering algorithm, we use the algorithm of section 3.4.3, which restricts itself to two-level fanout trees. This algorithm ignores required times but takes load values into account. The results obtained with this algorithm are reported in Table 3.6. Using this simple algorithm, we obtained a delay reduction of 23% for a cost in area of only 3% in average. Buffering alone accounts for 68% of the total delay reduction we can obtain with fanout optimization, for only a third of the cost in area.

**A Lower Bound on the Effect of Fanout Optimization** Fanout optimization can reduce delay only at multiple fanout points. Using critical signal isolation, it is possible to deliver a signal at a multiple fanout point with no overhead, provided that the other signals are required sufficiently late, which is not always the case. Thus a lower bound on the effect of fanout optimization can be determined by computing the arrival times of all signals of a network, ignoring the effect of multiple fanouts. To perform this computation at a multiple fanout point, we simply replace the sum of the sink loads by their average. The results of this computation are reported in Table 3.7 and compared with the delay values obtained with fanout optimization. Only delay values are reported. The data indicate that on average, fanout optimization operates within 12% of this lower bound. This result also provides a lower bound on the effect of source duplication, since source duplication can only decrease the delay through fanout nodes. Source duplication can still be a helpful technique, but its overall effect can only be secondary in comparison to the effect of fanout optimization.

**Peephole Optimization** We presented in section 3.6 several algorithms to improve the quality of a fanout tree after it has been built. To determine the effect of these peephole

| circuit | min area | | buffering | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 990 | 27.16 | 1054 | 26.29 | 1.06 | 0.97 |
| C1908 | 1086 | 35.04 | 1119 | 33.34 | 1.03 | 0.95 |
| C2670 | 1420 | 28.42 | 1441 | 25.15 | 1.01 | 0.88 |
| C3540 | 2201 | 45.24 | 2247 | 39.60 | 1.02 | 0.88 |
| C5315 | 3171 | 39.94 | 3220 | 36.26 | 1.02 | 0.91 |
| C6288 | 6777 | 120.37 | 6870 | 111.65 | 1.01 | 0.93 |
| C7552 | 4660 | 69.47 | 4758 | 35.64 | 1.02 | 0.51 |
| alu4 | 1486 | 47.17 | 1565 | 40.53 | 1.05 | 0.86 |
| ampbpreg | 2741 | 59.85 | 2815 | 49.92 | 1.03 | 0.83 |
| ampbsm | 1478 | 25.78 | 1524 | 20.62 | 1.03 | 0.80 |
| amppint2 | 1021 | 22.45 | 1052 | 15.62 | 1.03 | 0.70 |
| ampxhdl | 751 | 24.73 | 806 | 15.33 | 1.07 | 0.62 |
| apex6 | 1505 | 17.74 | 1519 | 15.72 | 1.01 | 0.89 |
| des | 6452 | 107.12 | 6818 | 21.27 | 1.06 | 0.20 |
| dflgrcb1 | 615 | 12.83 | 621 | 11.50 | 1.01 | 0.90 |
| fconrcb1 | 467 | 15.18 | 477 | 14.52 | 1.02 | 0.96 |
| frg2 | 1738 | 37.91 | 1834 | 21.05 | 1.06 | 0.56 |
| k2 | 1972 | 32.80 | 2017 | 26.75 | 1.02 | 0.82 |
| kcctlcb3 | 449 | 11.50 | 449 | 11.04 | 1.00 | 0.96 |
| pair | 3200 | 25.85 | 3341 | 20.48 | 1.04 | 0.79 |
| rot | 1387 | 30.18 | 1427 | 24.98 | 1.03 | 0.83 |
| sbiucb1 | 471 | 22.18 | 482 | 21.31 | 1.02 | 0.96 |
| tfaultcb1 | 375 | 9.91 | 381 | .9.33 | 1.02 | 0.94 |
| vda | 1118 | 23.76 | 1159 | 19.59 | 1.04 | 0.82 |
| x3 | 1653 | 22.40 | 1690 | 13.23 | 1.02 | 0.59 |
| aver | 1.00 | 1.00 | 1.03 | 0.77 | 1.03 | 0.77 |

Table 3.6: Effect of Buffering on Circuits Optimized by misII

**min area:** minimum area technology mapping with MCNC library *lib2*
**buffering:** minimum area technology mapping followed by buffering alone
**gain:** gain in area or in delay obtained by using buffering
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

| circuit | fanout opt delay | lower bound delay | ratio delay |
|---|---|---|---|
| C1355 | 24.25 | 20.53 | 0.85 |
| C1908 | 29.55 | 24.56 | 0.83 |
| C2670 | 22.14 | 19.74 | 0.89 |
| C3540 | 34.27 | 28.86 | 0.84 |
| C5315 | 30.78 | 27.01 | 0.88 |
| C6288 | 101.08 | 83.74 | 0.83 |
| C7552 | 28.55 | 24.48 | 0.86 |
| alu4 | 31.84 | 27.08 | 0.85 |
| ampbpreg | 39.40 | 34.24 | 0.87 |
| ampbsm | 18.05 | 15.97 | 0.88 |
| amppint2 | 12.31 | 11.00 | 0.89 |
| ampxhdl | 13.38 | 11.43 | 0.85 |
| apex6 | 13.40 | 12.55 | 0.94 |
| des | 17.84 | 15.71 | 0.88 |
| dflgrcb1 | 11.01 | 9.71 | 0.88 |
| fconrcb1 | 13.82 | 13.54 | 0.98 |
| frg2 | 14.95 | 12.06 | 0.81 |
| k2 | 20.56 | 18.46 | 0.90 |
| kcctlcb3 | 10.20 | 8.75 | 0.86 |
| pair | 17.80 | 16.00 | 0.90 |
| rot | 21.49 | 18.22 | 0.85 |
| sbiucb1 | 18.89 | 17.77 | 0.94 |
| tfaultcb1 | 8.41 | 7.75 | 0.92 |
| vda | 16.75 | 15.45 | 0.92 |
| x3 | 11.45 | 10.40 | 0.91 |
| aver | – | – | 0.88 |

Table 3.7: A Lower Bound on the Effect of Fanout Optimization

**fanout opt:** minimum area technology mapping with fanout optimization
**lower bound:** minimum area technology mapping ignoring fanout to compute delay
**ratio:** ratio between lower bound and delay realized with fanout optimization
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

| circuit | min area | | no peephole | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 990 | 27.16 | 1209 | 24.25 | 1.22 | 0.89 |
| C1908 | 1086 | 35.04 | 1324 | 29.64 | 1.22 | 0.85 |
| C2670 | 1420 | 28.42 | 1500 | 22.14 | 1.06 | 0.78 |
| C3540 | 2201 | 45.24 | 2587 | 33.97 | 1.18 | 0.75 |
| C5315 | 3171 | 39.94 | 3565 | 30.78 | 1.12 | 0.77 |
| C6288 | 6777 | 120.37 | 7512 | 101.72 | 1.11 | 0.85 |
| C7552 | 4660 | 69.47 | 5388 | 29.13 | 1.16 | 0.42 |
| alu4 | 1486 | 47.17 | 2010 | 31.90 | 1.35 | 0.68 |
| ampbpreg | 2741 | 59.85 | 3003 | 39.44 | 1.10 | 0.66 |
| ampbsm | 1478 | 25.78 | 1694 | 18.18 | 1.15 | 0.71 |
| amppint2 | 1021 | 22.45 | 1233 | 12.31 | 1.21 | 0.55 |
| ampxhdl | 751 | 24.73 | 921 | 13.53 | 1.23 | 0.55 |
| apex6 | 1505 | 17.74 | 1610 | 13.44 | 1.07 | 0.76 |
| des | 6452 | 107.12 | 8013 | 17.84 | 1.24 | 0.17 |
| dflgrcb1 | 615 | 12.83 | 647 | 11.01 | 1.05 | 0.86 |
| fconrcb1 | 467 | 15.18 | 489 | 13.82 | 1.05 | 0.91 |
| frg2 | 1738 | 37.91 | 1973 | 15.03 | 1.14 | 0.40 |
| k2 | 1972 | 32.80 | 2325 | 20.56 | 1.18 | 0.63 |
| kcctlcb3 | 449 | 11.50 | 479 | 10.20 | 1.07 | 0.89 |
| pair | 3200 | 25.85 | 3638 | 17.86 | 1.14 | 0.69 |
| rot | 1387 | 30.18 | 1484 | 21.47 | 1.07 | 0.71 |
| sbiucb1 | 471 | 22.18 | 543 | 18.98 | 1.15 | 0.86 |
| tfaultcb1 | 375 | 9.91 | 405 | 8.41 | 1.08 | 0.85 |
| vda | 1118 | 23.76 | 1466 | 16.77 | 1.31 | 0.71 |
| x3 | 1653 | 22.40 | 1815 | 11.51 | 1.10 | 0.51 |
| aver | – | – | – | – | 1.15 | 0.66 |

Table 3.8: Effect of Fanout Optimization without Peephole Optimization

**min area:** minimum area technology mapping with MCNC library *lib2*
**no peephole:** minimum area technology mapping followed by fanout optimization with no peephole optimization
**gain:** gain in area or in delay obtained when using no peephole optimization
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

optimizations, we ran the fanout optimizer after having deactivated them. The results of this experiment are reported in Table 3.8. In terms of delay, the impact of peephole optimization is negligible. This was to be expected, since our most powerful fanout optimization algorithm does buffer selection optimally. However, in terms of area, peephole optimization reduces the cost of fanout optimization from 15% down to 9%, which is a valuable contribution to the overall performance of the fanout optimizer.

**Effect of Area Recovery** After using fanout optimization for delay, we use a second pass on the network to recover area in fanout trees that are not critical for delay. The overall result is an average increase in area of only 9%, but it is not clear how much of this limited increase in area is due to area recovery or to the fact that fanout optimization for delay itself does not increase area significantly. To evaluate the effect of area recovery, we ran the fanout optimizer without area recovery. The results are reported in Table 3.9. Area recovery reduces the average cost in area of fanout optimization dramatically, from 51% to 9%. In addition, as predicted, area recovery does not increase circuit delay.

**Comparison with Singh's Algorithm** We compared the results obtained with our fanout optimizer to the results obtained with Singh's algorithm. To perform a fair comparison, we interfaced Singh's algorithm to our fanout optimizer, in such a way that it is called in the same order and on the same data than our fanout algorithms. The results are reported in Table 3.10. On most examples, the results are quite similar. Our fanout algorithm does consistently better than his in terms of delay, with an average reduction of 12%; but it does consistently worse than his in terms of area, with an average increase of 4%.

## 3.9 Conclusion

Fanout optimization is an important delay optimization technique, that can reduce delay often quite dramatically for a very moderate cost in area. It is an essential component of any logic synthesis system that claims to optimize delay. It is an important optimization for other reasons as well. In particular it can be directly adapted to make sure that fanout constraints imposed by a technology are satisfied.

Optimizing a fanout problem is, for most delay models, a difficult problem. In-

| circuit | min area | | no area recov | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 990 | 27.16 | 1362 | 24.25 | 1.38 | 0.89 |
| C1908 | 1086 | 35.04 | 1647 | 29.59 | 1.52 | 0.84 |
| C2670 | 1420 | 28.42 | 1995 | 22.08 | 1.40 | 0.78 |
| C3540 | 2201 | 45.24 | 3572 | 34.27 | 1.62 | 0.76 |
| C5315 | 3171 | 39.94 | 4929 | 30.78 | 1.55 | 0.77 |
| C6288 | 6777 | 120.37 | 10887 | 101.09 | 1.61 | 0.84 |
| C7552 | 4660 | 69.47 | 6959 | 28.68 | 1.49 | 0.41 |
| alu4 | 1486 | 47.17 | 2573 | 31.85 | 1.73 | 0.68 |
| ampbpreg | 2741 | 59.85 | 4209 | 39.46 | 1.54 | 0.66 |
| ampbsm | 1478 | 25.78 | 2219 | 18.05 | 1.50 | 0.70 |
| amppint2 | 1021 | 22.45 | 1483 | 12.31 | 1.45 | 0.55 |
| ampxhdl | 751 | 24.73 | 1116 | 13.45 | 1.49 | 0.54 |
| apex6 | 1505 | 17.74 | 2347 | 13.40 | 1.56 | 0.76 |
| des | 6452 | 107.12 | 9360 | 17.75 | 1.45 | 0.17 |
| dflgrcb1 | 615 | 12.83 | 815 | 11.01 | 1.33 | 0.86 |
| fconrcb1 | 467 | 15.18 | 653 | 13.82 | 1.40 | 0.91 |
| frg2 | 1738 | 37.91 | 2465 | 14.95 | 1.42 | 0.39 |
| k2 | 1972 | 32.80 | 3412 | 20.56 | 1.73 | 0.63 |
| kcctlcb3 | 449 | 11.50 | 605 | 10.20 | 1.35 | 0.89 |
| pair | 3200 | 25.85 | 5002 | 17.80 | 1.56 | 0.69 |
| rot | 1387 | 30.18 | 2151 | 21.52 | 1.55 | 0.71 |
| sbiucb1 | 471 | 22.18 | 806 | 18.89 | 1.71 | 0.85 |
| tfaultcb1 | 375 | 9.91 | 508 | 8.41 | 1.35 | 0.85 |
| vda | 1118 | 23.76 | 1846 | 16.75 | 1.65 | 0.70 |
| x3 | 1653 | 22.40 | 2344 | 11.54 | 1.42 | 0.52 |
| aver | 1.00 | 1.00 | 1.51 | 0.66 | 1.51 | 0.66 |

Table 3.9: Effect of Fanout Optimization without Area Recovery

**min area:** minimum area technology mapping with MCNC library *lib2*
**no area recov:** minimum area technology mapping followed by fanout optimization without area recovery
**gain:** gain in area or in delay obtained when using no area recovery
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

| circuit | fanout opt | | Singh's alg | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1119 | 24.25 | 1012 | 26.37 | 0.90 | 1.09 |
| C1908 | 1236 | 29.55 | 1182 | 31.48 | 0.96 | 1.07 |
| C2670 | 1478 | 22.14 | 1460 | 23.05 | 0.99 | 1.04 |
| C3540 | 2421 | 34.27 | 2301 | 37.95 | 0.95 | 1.11 |
| C5315 | 3352 | 30.78 | 3251 | 33.12 | 0.97 | 1.08 |
| C6288 | 7340 | 101.08 | 7003 | 110.29 | 0.95 | 1.09 |
| C7552 | 5084 | 28.55 | 4713 | 57.88 | 0.93 | 2.03 |
| alu4 | 1724 | 31.84 | 1592 | 35.82 | 0.92 | 1.12 |
| ampbpreg | 2891 | 39.40 | 2824 | 44.71 | 0.98 | 1.13 |
| ampbsm | 1615 | 18.05 | 1539 | 20.18 | 0.95 | 1.12 |
| amppint2 | 1136 | 12.31 | 1083 | 14.12 | 0.95 | 1.15 |
| ampxhdl | 865 | 13.38 | 818 | 14.09 | 0.95 | 1.05 |
| apex6 | 1565 | 13.40 | 1537 | 14.27 | 0.98 | 1.06 |
| des | 7358 | 17.84 | 6784 | 20.01 | 0.92 | 1.12 |
| dflgrcb1 | 630 | 11.01 | 623 | 11.33 | 0.99 | 1.03 |
| fconrcb1 | 481 | 13.82 | 472 | 13.89 | 0.98 | 1.01 |
| frg2 | 1893 | 14.95 | 1841 | 16.75 | 0.97 | 1.12 |
| k2 | 2189 | 20.56 | 2029 | 24.20 | 0.93 | 1.18 |
| kcctlcb3 | 469 | 10.20 | 456 | 10.30 | 0.97 | 1.01 |
| pair | 3482 | 17.80 | 3357 | 19.94 | 0.96 | 1.12 |
| rot | 1444 | 21.49 | 1427 | 22.11 | 0.99 | 1.03 |
| sbiucb1 | 514 | 18.89 | 487 | 20.59 | 0.95 | 1.09 |
| tfaultcb1 | 400 | 8.41 | 385 | 8.90 | 0.96 | 1.06 |
| vda | 1324 | 16.75 | 1217 | 17.80 | 0.92 | 1.06 |
| x3 | 1754 | 11.45 | 1699 | 14.86 | 0.97 | 1.30 |
| aver | – | – | – | – | 0.96 | 1.12 |

Table 3.10: Comparison with Singh's Algorithm

**fanout opt:** minimum area technology mapping followed by our fanout algorithm
**Singh's alg:** minimum area technology mapping followed by Singh's algorithm without area recovery
**gain:** gain or loss in area or in delay obtained by using Singh's algorithm
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

stead of trying to find one fanout algorithm that would be applicable in all contexts, we recommend the approach we have been following, of developing a spectrum of simple but efficient fanout optimization algorithms based on different approaches: balanced trees, *LT*-trees, combinational merging, top-down traversal. These algorithms are efficient enough to make it practical to try them all on every fanout problem to retain the best solution in all cases. In addition, some optimizations can be shared by all fanout algorithms if done during a postprocessing optimization phase on fanout trees. These optimizations do not affect delay, but can contribute significantly to area reduction.

The most important contribution of this chapter is to have demonstrated that, at least in the context of fanout optimization, there is a simple way of applying a fanout optimization algorithm to an entire network that is optimum in terms of delay and very efficient in terms of area. This is a significant improvement over past methods which rely on the identification and incremental improvement of critical paths. Our method is guaranteed to produce the best delay achievable with a given fanout optimization algorithm and requires only two passes on the network. In addition it achieves significant delay reduction for a very moderate cost in area, observed in our experiments to be no more than 10% on average.

To minimize area under a delay constraint, we did not modify any of the fanout algorithms to do so. Rather, we simply apply every fanout algorithm to each fanout problem, and selected among the fanout trees so obtained that met the delay constraint, one with minimum area. Area reduction is achieved simply by using a spectrum of fanout algorithms, including some that have can only produce fairly simple fanout trees.

# Chapter 4

# Combining Tree Covering and Fanout Optimization

It is not necessary to hope to undertake
nor to succeed to persevere.
— GUILLAUME d'ORANGE

## 4.1 Introduction

In the previous two chapters, we covered separately two important delay optimization techniques: tree covering and fanout optimization. The purpose of this chapter is to study the problem of integrating these two optimizations. Figure 4.1 illustrates how we have applied so far these optimizations. In gray are *fanin trees*, the parts of a circuit where we apply tree covering to perform gate selection. In white are *fanout trees*, where we apply fanout optimization. We can always combine tree covering and fanout optimization as follows: we first use tree covering to implement the fanin trees, in one pass from primary inputs to primary outputs. We then apply fanout optimization, in one pass from primary outputs to primary inputs. We can run an additional fanout optimization pass to recover area, but we will ignore area recovery for the moment. The question we would like to answer in this chapter is: are there better ways to apply tree covering and fanout optimization to a network that would lead to significant speed improvements?

We first introduce some definitions and terminology used in the rest of this chapter. We partition a Boolean network into fanin trees and fanout trees. We group each tree into

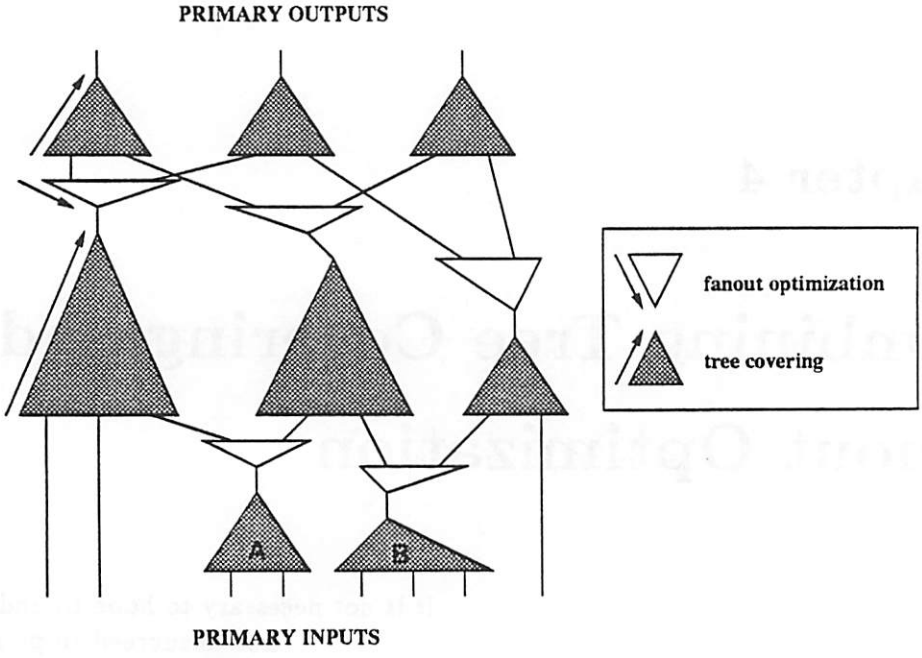


Figure 4.1: Combining Tree Covering and Fanout Optimization

a node. To each fanin tree corresponds a *fanin node*, and to each fanout tree corresponds a *fanout node*. Fanin nodes are specified by a Boolean function that can be implemented as a tree. Fanout nodes are simply specified by a source and a set of sinks. The polarity of these sinks is usually not known before the fanin trees are implemented. Fanin nodes can be *implemented* by tree covering, or by some form of restructuring followed by tree covering. In this work, we only use tree covering, but the theoretical part of this chapter remains valid if we use restructuring in addition to tree covering. Fanout nodes are implemented using fanout optimization. In each case, we suppose that the implementation attempts to minimize delay for a given set of arrival times for fanin nodes or required times for fanout nodes. An implementation that is such that all delays from the leaves of the tree to the root of the tree are equal is called a *balanced* implementation. We use *unbalanced* implementations to compensate for the imbalance in arrival times or required times. The problem we study in this chapter is the problem of finding a good order of application of tree covering (possibly with restructuring) and fanout optimization to minimize delay.

We call this problem the *tree-based delay optimization problem*, since it is the problem of minimizing delay while respecting the tree boundaries of a Boolean network.

In section 4.2 we formulate the tree-based delay optimization problem as a convex optimization problem. This formulation is only valid for a continuous version of the constant delay model of section 3.3.1. By abstracting away the discrete nature of delay optimization in our setting, this formulation allows us to compute analytically the minimum delay implementation of a few simple circuits. We can then use these examples to detect potential sources of suboptimality in tree-based delay optimization algorithms. In this section we exhibit a circuit that has the following property: for any constant $\alpha > 0$, there is an implementation of this circuit whose delay is $\alpha$ delay units worse than an optimal implementation and is such that all paths are critical and all fanin nodes and all fanout nodes are implemented optimally. Such an implementation cannot be optimized by any greedy application of tree covering or fanout optimization in any order. Due to physical constraints, this example is only realistic for a limited range of values of $\alpha$. Nevertheless it indicates clearly the limitations of greedy delay improvement strategies.

This example is based on an initial implementation where arbitrarily unbalanced implementations of fanin and fanout nodes compensate each other exactly. We can easily avoid this problem by starting with a balanced initial implementation. To do so, we implement all fanout trees using a balanced configuration prior to the first application of tree covering. This technique is described in section 4.3. The main difficulty in building these balanced fanout trees is to handle sink polarities properly. Since the fanout trees are built before an implementation of fanin trees is available, sink polarities are not known. This phase assignment problem has an important impact on the quality of the final implementation.

Once we have built an initial implementation, we can iterate tree covering and fanout optimization until we reach a local minimum. In section 4.4, we present a simple iterative scheme that we can use to perform this iteration. This scheme consists in iterating tree covering and fanout optimization passes on the network, using at the $i^{th}$ iteration the delay information computed at the $(i-1)^{th}$ iteration. Our experimental results indicate that there is almost no advantage in performing more than one iteration with this method. To estimate the optimality of the final result, we applied this optimization scheme to another simple circuit for which we can compute the optimal solution for a simple delay model. On this circuit also the iterative algorithm converges almost immediately, and the result

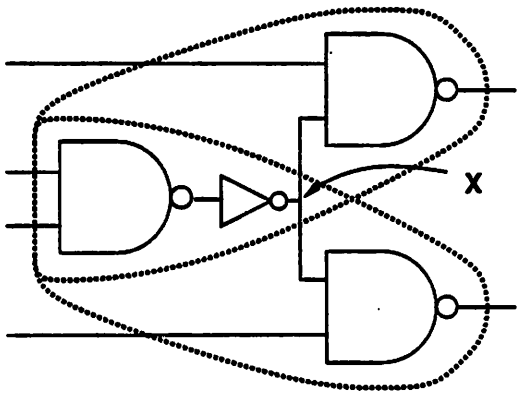


Figure 4.2: Suboptimality of Tree-Based Optimization

obtained after one iteration is within a few percent of the optimum solution. This example also suggests an improvement of the iterative algorithm, which converges more slowly but reaches a solution that is within a fraction of a percent of the optimum. We conclude section 4.4 by a brief overview of a new approach to tree-based minimization proposed by Yoshikawa *et al.* [44].

The results of the section 4.4 are a strong indication that we are reaching the limit of what can be achieved with tree-based delay minimization. The purpose of section 4.5 is to propose two techniques to minimize delay that do not preserve tree boundaries but are nevertheless simple variations of the tree covering and fanout optimization algorithms proposed so far. The first of these techniques, `tree duplication`, allows the duplication of a fanin node to implement the node both in positive and negative phases. In tree covering, one phase is selected, and the other phase is provided by an inverter. With tree duplication, both phases are implemented as separate trees, possibly with partial overlap. The rationale behind this heuristic is to make available to the fanout optimizer the earliest possible source for each polarity, possibly reducing by one the number of buffers on a critical path. Another factor makes this optimization attractive: the possibility of avoiding unnecessary duplication during the area recovery phase of fanout optimization. The second of these techniques, *tree overlap*, is more radical. It allows tree covering to ignore tree boundaries. The example of Figure 4.2 illustrates why allowing overlaps between trees can help reducing circuit delay. The circuit shown in this example can be implemented as shown in solid lines, with three

2-input NAND gates and one inverter. Or it can be implemented as suggested by the dotted lines, with two 3-input NAND gates. The second implementation is significantly faster, and in that case happens to use less area. In general, however, allowing overlaps tend to increase area. If the example of Figure 4.2 is modified to have a fanout of 10, the implementation with ten 3-input NAND gates will still be faster, but would use more area than an implementation with eleven 2-input NAND gates and one inverter. This optimization has the effect of moving logic across fanout points in a manner reminiscent of retiming [30].

The results obtained in this chapter indicate that we are reaching a limit to what we can expect from tree-based technology mapping algorithms in terms of delay optimization. This opens the way to the next chapter, where we examine the effect of technology-independent optimizations that can modify the global structure of a network.

## 4.2 Theoretical Analysis of Tree-Based Delay Minimization

In this section, we present an abstract formulation of the tree-based delay minimization problem, as a convex optimization problem. This formulation is obtained by abstracting away the discrete nature of our algorithms, but uses delay equations directly derived from the exact solution of fanout problems using the constant delay model of section 3.3.1 and in that sense represents a reasonable continuous approximation of the discrete problem we are seeking to solve.

We first show how we model the effect of tree covering (with restructuring) using a convex function, derived from combinational merging [18]. Then using symmetry we apply this model to cover fanout optimization. Using these models, we can formulate the tree-based covering problem as a convex optimization problem. We use this formulation on a simple circuit to compute the optimum implementation for that circuit, and exhibit a class of implementations of that circuit that cannot be improved using greedy tree-based optimization algorithms.

### 4.2.1 Modeling Tree Covering

To model the effect of tree covering and restructuring on a fanin node, we use a function $f(a_1, \ldots, a_n)$ that represents the arrival time achieved by an optimal implementation of a fanin node $v$ at the output of $v$ given arrival times $(a_1, \ldots, a_n)$ at the inputs of

$v$. If the Boolean function computed by fanin node $v$ is a Boolean and of $n$ inputs, and if the library is composed of and gates of constant delay equal to 1 and fanin $k$, where $k$ lies between 2 and some limit $t$, the optimal implementation of $v$ can be obtained using combinational merging. Moreover, the function $f$ can be computed exactly in that case, and is given by the following formula [18, 22]:

$$f(a_1, \ldots, a_n) = \left\lceil \log_t \sum_{1 \le i \le n} t^{a_i} \right\rceil \tag{4.1}$$

To be able to use optimization techniques from real analysis, we need to approximate $f$ by a differentiable function. We use the following approximation, obtained by dropping the ceiling function and setting for convenience $t = e$. A different choice of $k$ would only have for effect of scaling the delay numbers by a multiplicative constant.

$$f(a_1, \ldots, a_n) = \log \sum_{1 \le i \le n} e^{a_i} \tag{4.2}$$

This approximation ignores the discrete nature of tree covering and restructuring, and the added irregularity of fanin nodes with asymmetric logic functions. However it captures the essence of fanin tree balancing. It produces a delay of $\log n$ for balanced arrival times, and, for unbalanced arrival times, allows late signals to traverse a fanin node for less delay at an extra cost for the other signals. Moreover, the model has not been chosen arbitrarily, but derived directly from the optimal solution of a fanin problem for a simple delay model. In addition, this model has the following important property:

**lemma 4.2.1** *The function $f$ given by equation 4.2 is convex.*

**Proof** The function $f$ is obtained by composition of infinitely differentiable functions, and is thus infinitely differentiable over $\mathcal{R}^n$. To prove that it is strictly convex, we will show that its Laplacian is strictly positive at any point of the space. We first define the following coefficients:

$$a_{ij} = \frac{e^{x_i + x_j}}{(\sum_{k=1}^{n} e^{x_k})^2} \tag{4.3}$$

A simple computation yields the following equalities:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = -a_{ij} \quad \text{if} \quad i \neq j$$

$$\frac{\partial^2 f}{\partial {x_i}^2} = -a_{ii} + \sum_{k=1}^{n} a_{ik}$$

We deduce from these equations, and from the fact that $a_{ij} = a_{ji}$, the following inequality, which proves our result:

$$\begin{aligned}\sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} y_i y_j &= -\sum_{i,j} a_{ij} y_i y_j + \sum_{i,j} a_{ij} y_i^2 \\ &= \sum_{i<j} a_{ij}(y_i^2 + y_j^2 - 2y_i y_j) \\ &= \sum_{i<j} a_{ij}(y_i - y_j)^2 \quad \geq 0\end{aligned}$$

■

### 4.2.2 Modeling Fanout Optimization

To model the effect of fanout optimization, we use a function $g(r_1, \ldots, r_n)$ that represents the required time achieved by an optimal implementation of a fanout node $v$ at the input of $v$ given required times $(r_1, \ldots, r_n)$ at the outputs of $v$. If the fanout node $v$ has $n$ outputs, and the library is composed of buffers of constant delay equal to 1 and fanout $k$, where $k$ lies between 2 and some limit $t$, the fanout problem is the exact analog to the simplified fanin problem used in the previous section. One problem can be deduced from the other by changing the direction of propagation of signals. As a consequence, the optimum solution can be computed for this problem as follows:

$$g(r_1, \ldots, r_n) = -\left\lceil \log_t \sum_{1 \leq i \leq n} t^{-r_i} \right\rceil \tag{4.4}$$

The continuous approximation of $g$ is obtained in the same fashion as the continuous approximation of $f$ in the previous section:

$$g(r_1, \ldots, r_n) = -\log \sum_{1 \leq i \leq n} e^{-r_i} \tag{4.5}$$

It is easy to check that $g$ is concave, since $f$ is convex and $g(x) = -f(-x)$.

### 4.2.3 Formulation as a Convex Optimization Problem

Given a continuous model of the delay through fanin nodes and fanout nodes as the result of tree covering and fanout optimization, we can proceed to formulate the tree-based delay optimization problem as a global optimization problem. We assume that the Boolean network to be optimized is decomposed into fanin nodes and fanout nodes. Without loss of

generality, we can suppose that the fanin nodes and fanout nodes are maximal, in the sense that a fanin node is never connected to another fanin node, nor a fanout node to another fanout node. If it were not the case, i.e. for example if two fanin nodes were connected, the input node could always be collapsed into the output node.

We divide the edges or wires of the network into four sets: $PI$, $PO$, $LEAF$ and $ROOT$. Each wire only connects two nodes. Multiple fanouts are handled exclusively through fanout nodes. The $PI$ wires are wires directly connected to primary inputs. Associated with each of the $PI$ wires is an arrival time. Arrival times are represented as a vector $a$ of arrival times, of dimension $|PI|$. Similarly, the $PO$ wires are the wires directly connected to primary outputs. Associated with each $PO$ wire is a required times. Required times are represented as a vector $r$ of dimension $|PO|$. The $LEAF$ wires are the wires that connect an output of a fanout node to an input of a fanin node. Arrival times on these wires are represented by a vector $x$ of dimension $|LEAF|$. Finally the $ROOT$ wires are the wires connecting the output of a fanin node to the input of a fanout node. Figure 4.3 illustrates these definitions.

For each fanin node, we suppose that we have at our disposal a function that computes the best achievable arrival time at the output of the node, for any feasible implementation of the node. This function only depends on $x$ and $a$. To simplify the notation, we designate by $f$ the vector of such functions, and we keep implicit the dependency on the vector of arrival times $a$, considered constant. Thus $f : \mathcal{R}^{|LEAF|} \rightarrow \mathcal{R}^{|PO|} \times \mathcal{R}^{|ROOT|}$. All the components of $f$ are convex functions.

We note $\pi_{PO}$ the orthogonal projection of a vector of $\mathcal{R}^{|PO|} \times \mathcal{R}^{|ROOT|}$ onto $\mathcal{R}^{|PO|}$, and similarly $\pi_{ROOT}$ designates the projection onto $\mathcal{R}^{|ROOT|}$. For a given assignment of arrival times $x$ at the $LEAF$ wires, $\pi_{PO}(f(x))$ designates the best achievable arrival times at the primary outputs of the network. For $1 \leq p \leq +\infty$, $|x|_p$ designates the quantity $(\sum_{k=1}^{n} x_k^p)^{\frac{1}{p}}$, and $x_+$ designates the vector of components $(\max(x_k, 0))_{1 \leq k \leq n}$. The total amount by which an implementation fails to meet its timing requirements is given by $|(\pi_{PO}(f(x)) - r)_+|_1$, and the maximum amount by which a requirement fails is given by $|(\pi_{PO}(f(x)) - r)_+|_\infty$. In both cases, the convexity of the components of $f$ implies the convexity of the cost function.

Similarly, for each fanout node, we suppose that we have at our disposal a function that computes the best achievable required time at the input of the node, for a given set of required times at the outputs. This function only depends on $x$ and the constant vector $r$.

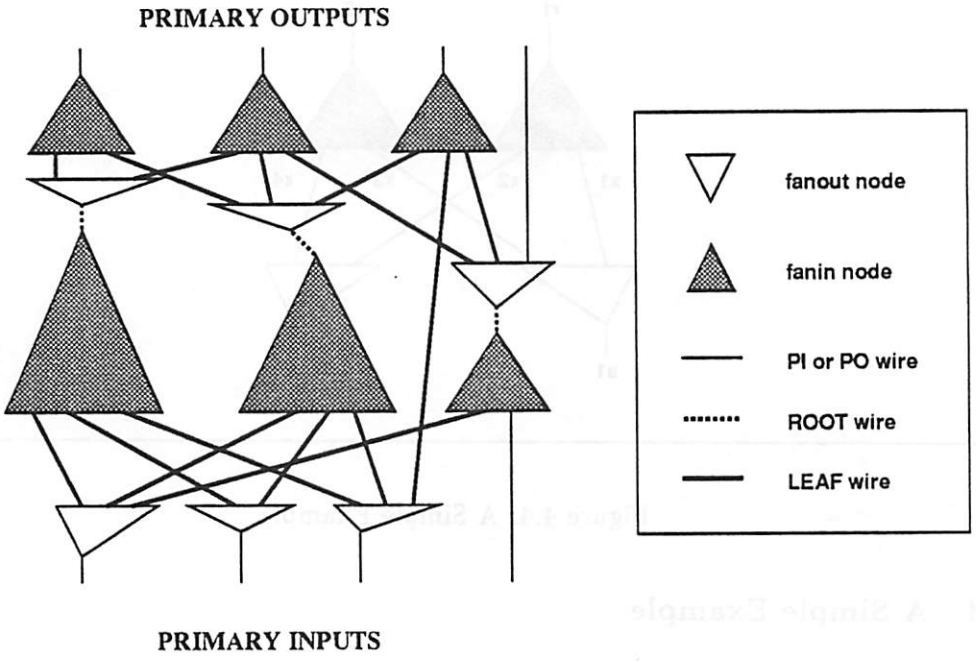


Figure 4.3: Partition of Edges

As for primary input arrival times, to simplify notation, we keep implicit the dependency on the primary output required times. We designate by $g$ the vector of these functions: $g : \mathcal{R}^{|LEAF|} \rightarrow \mathcal{R}^{|PI|} \times \mathcal{R}^{|ROOT|}$. A $LEAF$ wire assignment $x$ is realizable if it satisfies the following inequalities: $\pi_{|PI|}(g(x)) \geq a$ and $\pi_{|ROOT|}(g(x)) \geq \pi_{|ROOT|}(f(x))$.

In total, we have formulated the tree-based delay minimization problem as the following optimization problem:

$$\begin{aligned} \min_{x \in \mathcal{R}^{|LEAF|}} \quad & |(\pi_{|PO|}(f(x)) - r)_+|_\infty \\ \pi_{|PI|}(g(x)) &\geq a \\ \pi_{|ROOT|}(g(x)) &\geq \pi_{|ROOT|}(f(x)) \end{aligned}$$

Each constraint in the problem if of the form $g \geq f$, where $g$ is a concave function and $f$ a convex function. The set of points satisfying the constraints is thus convex, and the problem has been expressed as the problem if minimizing a convex function over a convex set.

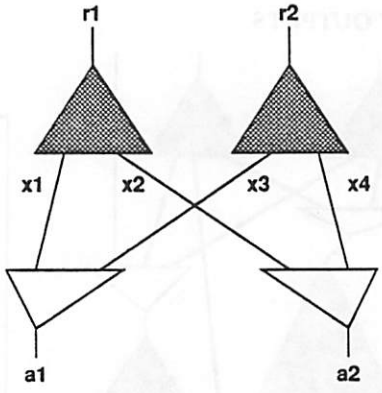


Figure 4.4: A Simple Example

### 4.2.4 A Simple Example

We now proceed to use this formulation in order to compute an analytical solution to the tree-based delay minimization problem for the simple circuit of Figure 4.4. The delay function associated at each fanin node is given by equation 4.2 and by equation 4.5 for the two fanout nodes.

We are interested in computing the best pairs of arrival times realizable at the outputs of this circuit, i.e. in finding all the sets of points $(r_1, r_2)$ that are realizable arrival times for the circuit of Figure 4.4, and minimal for the partial ordering on vectors defined by: $x \leq y$ iff $x_i \leq y_i$ for all $1 \leq i \leq n$. In other words, we are interested in the pairs of realizable arrival times $(r_1, r_2)$ such that $r_1$ and $r_2$ cannot be improved simultaneously. Such sets of optimal points are often studied in microeconomics and called `Pareto optimal` points. These points must satisfy the following equations:

$$\begin{aligned} r_1 &= \log(e^{x_1} + e^{x_2}) \\ r_2 &= \log(e^{x_3} + e^{x_4}) \\ a_1 &= -\log(e^{-x_1} + e^{-x_3}) \\ a_2 &= -\log(e^{-x_2} + e^{-x_4}) \end{aligned}$$

where $(a_1, a_2)$ are given arrival times at the inputs of the network. To simplify the compu-

tation, we perform the following substitution of variables:

$$\begin{aligned} R_i &= e^{r_i} \\ A_i &= e^{a_i} \\ X_i &= e^{x_i} \end{aligned}$$

The problem is then to reduced to finding the minimal pairs $(R_1, R_2)$ satisfying the following two equations:

$$\begin{aligned} R_1 &= X_1 + X_2 \\ R_2 &= \frac{A_1 X_1}{X_1 - A_1} + \frac{A_2 X_2}{X_2 - A_2} \end{aligned}$$

The minimal points must be such that $(\frac{\partial R_1}{\partial X_1}, \frac{\partial R_1}{\partial X_2})$ and $(\frac{\partial R_2}{\partial X_1}, \frac{\partial R_2}{\partial X_2})$ are colinear, otherwise it would be possible to decrease both $R_1$ and $R_2$ and still stay in the feasibility region. The first vector is equal to $(1,1)$ and the second vector to $(\frac{A_1^2}{(X_1-A_1)^2}, \frac{A_2^2}{(X_2-A_2)^2})$. Thus the minimal points are characterized by the equation:

$$\frac{X_1}{A_1} = \frac{X_2}{A_2}$$

A parametric representation of the set of minimum points is then readily available, using $T = \frac{X_1}{A_1}$ as parameter. The range of $T$ is $1 < T < \infty$.

$$\begin{aligned} R_1 &= T(A_1 + A_2) \\ R_2 &= \frac{T}{T-1}(A_1 + A_2) \end{aligned}$$

An equivalent closed from is given by:

$$-\log(e^{-r_1} + e^{-r_2}) = \log(e^{a_1} + e^{a_2})$$

This closed form indicates that the behavior of this circuit is equivalent to the behavior of a single fanout node with an input arrival time of $\log(e^{a_1} + e^{a_2})$.

### 4.2.5 Suboptimal Local Minima

With the example of the previous section, we can exhibit a family of circuit implementations for the same network that are arbitrarily far from the optimum solution, but yet have the property that every node, taken in isolation, is configured optimally with respect

to the rest of the network. In other words, any algorithm that works greedily by changing the circuit implementation one node at a time or a group composed of a fanin node and its fanout node at a time, can enter a global implementation where every single node or group appears optimal with respect to the rest of the circuit, but the overall delay through the circuit is arbitrarily far from the optimal solution.

This family of implementations is parametrized by one parameter, designated as $\alpha$. Given $\alpha$, we consider the following wire assignment to the circuit of Figure 4.4, where we now suppose that the arrival times and the required times are all equal to 0:

$$x_1 = \log(1 + e^{\alpha}) \tag{4.6}$$

$$x_2 = \log(1 + e^{-\alpha}) \tag{4.7}$$

$$x_3 = \log(1 + e^{-\alpha}) \tag{4.8}$$

$$x_4 = \log(1 + e^{\alpha}) \tag{4.9}$$

Since we have:

$$\begin{aligned} -\log(e^{-x_1} + e^{-x_3}) &= -\log(e^{-x_2} + e^{-x_4}) \\ &= -\log\left(\frac{1}{1+e^{\alpha}} + \frac{e^{\alpha}}{1+e^{\alpha}}\right) \\ &= 0 \end{aligned}$$

this leaf wire assignment is optimal with respect to the two fanout nodes. The best primary output arrival times achievable with this wire assignment are equal, for both primary outputs, to the value:

$$\begin{aligned} \log(e^{x_1} + e^{x_2}) &= \log(e^{x_3} + e^{x_4}) \\ &= \log(1 + e^{\alpha} + 1 + e^{-\alpha}) \\ &= \log 2 + \log(\cosh(\alpha)) \end{aligned}$$

which can be made arbitrarily far from an optimal solution. Solutions arbitrarily far from the optimum are not realized in practice due to physical limitations. This result simply indicates that greedy tree-based delay optimization algorithms are unable to recover from certain initial implementations, no matter how suboptimal these implementations are.

In addition, if we take as initial implementation an implementation where the fanout nodes are implemented optimally relative to the wire values given by equations 4.6

to 4.9, and the fanin trees are implemented arbitrarily, a one pass application of tree covering to this implementation will produce a configuration that cannot be optimized any further and whose distance to the optimal solution is given by equation 4.10. Only if the initial implementation is balanced would the result be optimal.

## 4.3 Selecting the Initial Implementation

Before improving a network implementation, we need to generate one. Since our incremental improvement algorithms are not very powerful, the quality of the final result is very sensitive to the quality of the initial point. To compute an initial implementation, we perform tree covering on the entire network, using a heuristic to estimate the arrival times at the output of fanout nodes. The quality of the initial implementation is very sensitive to the quality of this heuristic. From the previous section, we know that it is important not to introduce artificial imbalances in the initial implementation, because they are a source of suboptimality that cannot always be eliminated by incremental improvement algorithms. Therefore a heuristic to estimate arrival times at the output of fanout nodes should estimate these arrival times to be equal for all the outputs of a given fanout node.

There are still many ways to estimate balanced arrival times, and the following questions remain to be answered:

- what should be the delay through a fanout node?
- should the delay be sensitive to signal polarity?
- should the delay be sensitive to variations in sink loads?

These questions arise from the fact that the heuristic is to be used during tree covering. Since the implementation of fanin nodes is not known at this point, neither the polarity nor the load at the outputs of fanout nodes is known when the delay estimation heuristic is called.

**A Simple Fanout Delay Estimation Heuristic** We present in this paragraph a simple delay estimation heuristic that performs consistently better than its variants. With this heuristic, we estimate the arrival time at the output of a fanout node $v$ as follows. We first compute the best arrival time achievable at the input of $v$ by tree covering. This information is available because tree covering proceeds in topological order from primary

inputs to primary outputs. Then, if $n$ is the number of fanouts of $v$, we build a balanced fanout tree with $n$ sinks to implement $v$. The fanout tree is selected by the balanced tree algorithm presented in section 3.4.3. The loads of the sinks are taken to be all equal to some generic value (we use the input load of a minimum size 2-input NAND gate).

The computation of this fanout tree is done twice, once by supposing that all sinks are of positive polarity, once by supposing that all sinks are of negative polarity. Among these two trees, the fanout tree with the earlier output arrival time is stored at node $v$, and the other tree is discarded. The fanout tree stored at node $v$ in this fashion is called $T_v$. We use $T_v$ to compute the arrival time at any output of $v$.

When tree covering is applied to a fanin node in the fanout of $v$, we need to compute the arrival time at gate inputs $p$ connected to $v$. To do so, we compute the delay through the fanout tree $T_v$ stored at $v$, adding to the load driven by the buffer connected to $p$ the difference between the load at $p$ and the generic sink load value we used to build $T_v$. This computation is done irrespective of the polarity of $p$.

**Taking Sink Polarity into Account** We modified the heuristic described in the previous paragraph to take sink polarities into account. This modification was done as follows: we stored with fanout tree $T_v$ at each fanout node $v$ the polarity of the signals available at its outputs. Since $T_v$ is a balanced tree, all outputs have the same polarity. When the input pin $p$ of a gate requires the signal under the polarity provided by $T_v$, we compute the arrival time at $p$ as before. When the signal is required under the opposite polarity, we add an inverter delay to the arrival time obtained from $T_v$. We measured the effect of this modified fanout delay heuristic on the quality of circuit implementations obtained after one pass of tree covering followed by two passes of fanout optimization, one for delay and one for area recovery. As show in Table 4.1, taking sink polarity into account yields circuits that are on average 11% slower and 2% larger.

By taking polarities into account at this stage, we introduce an artificial bias in favor of polarity $X$ over polarity $\overline{X}$. In doing so we tend to eliminate tree covers that would have chosen polarity $\overline{X}$ if $X$ and $\overline{X}$ were available with the same arrival time, before we can determine whether later passes of fanout optimization could provide a signal of $\overline{X}$ at less cost than $X$ plus an extra inverter delay. In the absence of such information, it is more important not to introduce any bias than to try to be consistent with a feasible implementation of a fanout tree.

| circuit | no polarity | | polarity | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1204 | 25.99 | 1.01 | 1.07 |
| C1908 | 1358 | 28.24 | 1381 | 29.62 | 1.02 | 1.05 |
| C2670 | 1796 | 20.86 | 1809 | 21.66 | 1.01 | 1.04 |
| C3540 | 2857 | 32.88 | 2906 | 37.28 | 1.02 | 1.13 |
| C5315 | 3693 | 29.31 | 3789 | 32.08 | 1.03 | 1.09 |
| C6288 | 8272 | 95.47 | 8932 | 109.69 | 1.08 | 1.15 |
| C7552 | 5322 | 27.83 | 5303 | 29.98 | 1.00 | 1.08 |
| alu4 | 1977 | 29.24 | 1984 | 31.31 | 1.00 | 1.07 |
| ampbpreg | 3289 | 35.56 | 3422 | 42.43 | 1.04 | 1.19 |
| ampbsm | 1875 | 16.55 | 1930 | 19.51 | 1.03 | 1.18 |
| amppint2 | 1353 | 11.29 | 1390 | 11.92 | 1.03 | 1.06 |
| ampxhdl | 1059 | 12.90 | 1022 | 13.85 | 0.97 | 1.07 |
| apex6 | 1912 | 11.29 | 1905 | 12.41 | 1.00 | 1.10 |
| des | 8632 | 16.08 | 8598 | 17.40 | 1.00 | 1.08 |
| dflgrcb1 | 730 | 10.39 | 725 | 11.03 | 0.99 | 1.06 |
| fconrcb1 | 537 | 11.00 | 578 | 13.88 | 1.08 | 1.26 |
| frg2 | 2367 | 13.17 | 2204 | 14.85 | 0.93 | 1.13 |
| k2 | 2755 | 16.87 | 2887 | 19.69 | 1.05 | 1.17 |
| kcctlcb3 | 557 | 9.26 | 597 | 9.75 | 1.07 | 1.05 |
| pair | 3956 | 16.40 | 3978 | 19.41 | 1.01 | 1.18 |
| rot | 1651 | 19.17 | 1724 | 20.67 | 1.04 | 1.08 |
| sbiucb1 | 599 | 15.66 | 641 | 17.47 | 1.07 | 1.12 |
| tfaultcb1 | 439 | 6.80 | 483 | 7.92 | 1.10 | 1.16 |
| vda | 1522 | 13.31 | 1620 | 15.06 | 1.06 | 1.13 |
| x3 | 2033 | 10.97 | 2075 | 12.65 | 1.02 | 1.15 |
| aver | - | - | - | - | 1.02 | 1.11 |

Table 4.1: Effect of Taking Polarities Into Account in Fanout Delay Heuristic

**no polarity:** fanout delay heuristics ignores polarities
**polarity:** fanout delay heuristics takes polarities into account
**gain:** increase in area or in delay obtained by taking polarities into account
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

**Using a Wire as a Balanced Fanout Tree** A wire is the simplest balanced fanout tree we could use to estimate the arrival time at the output of fanout nodes. Unfortunately, the use of a wire as an delay estimator at a fanout node yields implementations of poorer quality, as shown in Table 4.2, with a degradation of 9% in delay and 3% in area. As in the previous paragraph, these results were obtained by using the fanout delay heuristic during an initial pass of tree covering, and completing the implementation of the circuits by two passes of fanout optimization, one for delay and one for area. By using a linear delay model at a fanout point, we introduce artificial imbalances in the implementation, in particular in the presence of nodes with large fanouts.

**Conclusion** In summary, we found that the best delay heuristic estimates the delay through a fanout node using a balanced fanout tree and ignores sink polarities. This heuristic also takes into account variations in sink loads, though we have not assessed the effectiveness of this technique, on the ground that it is straightforward to implement and we expect it to have only a second order effect on delay. In the results presented in the remainder of this thesis, we apply this heuristic during the initial tree covering phase.

## 4.4 Global Optimization Schemes

In this section we present two iterative delay optimization algorithms. The first algorithm, presented in section 4.4.1, simply iterates tree covering and fanout optimization. We call it the *simple iterative improvement algorithm.* Experimentally this algorithm converges very rapidly, and the result obtained after one iteration is almost as good as the final result. Unfortunately these experiments do not provide any information about the optimality of the final result. To gain some insight into possible sources of suboptimality, we introduce in section 4.4.2 a simple network for which we can compute an optimal implementation under the continuous delay model of section 4.2. We simulate the effect of the simple iterative improvement algorithm on this network under the continuous delay model. The solution obtained by this algorithm is suboptimal, but only within a few percent of the optimum solution. Finally, in section 4.4.3 we discuss briefly a new technique proposed by Yoshikawa *et al.* [44] to perform iterative improvements that uses an different approach.

### 4.4.1 Iterative Improvement

| circuit | logarithmic | | linear | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1283 | 26.55 | 1.08 | 1.10 |
| C1908 | 1358 | 28.24 | 1374 | 30.84 | 1.01 | 1.09 |
| C2670 | 1796 | 20.86 | 1810 | 21.28 | 1.01 | 1.02 |
| C3540 | 2857 | 32.88 | 2824 | 37.97 | 0.99 | 1.15 |
| C5315 | 3693 | 29.31 | 3907 | 30.85 | 1.06 | 1.05 |
| C6288 | 8272 | 95.47 | 8723 | 107.51 | 1.05 | 1.13 |
| C7552 | 5322 | 27.83 | 5754 | 29.85 | 1.08 | 1.07 |
| alu4 | 1977 | 29.24 | 2017 | 30.93 | 1.02 | 1.06 |
| ampbpreg | 3289 | 35.56 | 3330 | 38.52 | 1.01 | 1.08 |
| ampbsm | 1875 | 16.55 | 1995 | 17.52 | 1.06 | 1.06 |
| amppint2 | 1353 | 11.29 | 1388 | 12.16 | 1.03 | 1.08 |
| ampxhdl | 1059 | 12.90 | 1064 | 13.46 | 1.00 | 1.04 |
| apex6 | 1912 | 11.29 | 2029 | 13.46 | 1.06 | 1.19 |
| des | 8632 | 16.08 | 8659 | 17.92 | 1.00 | 1.11 |
| dflgrcb1 | 730 | 10.39 | 725 | 10.39 | 0.99 | 1.00 |
| fconrcb1 | 537 | 11.00 | 564 | 12.38 | 1.05 | 1.13 |
| frg2 | 2367 | 13.17 | 2440 | 13.42 | 1.03 | 1.02 |
| k2 | 2755 | 16.87 | 2775 | 18.13 | 1.01 | 1.07 |
| kcctlcb3 | 557 | 9.26 | 567 | 10.77 | 1.02 | 1.16 |
| pair | 3956 | 16.40 | 3923 | 20.46 | 0.99 | 1.25 |
| rot | 1651 | 19.17 | 1722 | 19.26 | 1.04 | 1.00 |
| sbiucb1 | 599 | 15.66 | 616 | 16.90 | 1.03 | 1.08 |
| tfaultcb1 | 439 | 6.80 | 443 | 8.24 | 1.01 | 1.21 |
| vda | 1522 | 13.31 | 1571 | 14.66 | 1.03 | 1.10 |
| x3 | 2033 | 10.97 | 2207 | 11.42 | 1.09 | 1.04 |
| aver | – | – | – | – | 1.03 | 1.09 |

Table 4.2: Effect of Using a Logarithmic vs. Linear Delay Estimate

**logarithmic:** use a logarithmic model to estimate delay through fanout node
**linear:** use a linear delay to estimate delay through fanout node
**gain:** increase in area or in delay obtained by using a linear delay estimate
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

| circuit | one iter | | three iters | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1182 | 24.61 | 0.99 | 1.02 |
| C1908 | 1358 | 28.24 | 1342 | 27.81 | 0.99 | 0.98 |
| C2670 | 1796 | 20.86 | 1768 | 20.92 | 0.98 | 1.00 |
| C3540 | 2857 | 32.88 | 2835 | 32.49 | 0.99 | 0.99 |
| C5315 | 3693 | 29.31 | 3760 | 28.77 | 1.02 | 0.98 |
| C6288 | 8272 | 95.47 | 8413 | 94.20 | 1.02 | 0.99 |
| C7552 | 5322 | 27.83 | 5135 | 26.89 | 0.96 | 0.97 |
| alu4 | 1977 | 29.24 | 1974 | 29.06 | 1.00 | 0.99 |
| ampbpreg | 3289 | 35.56 | 3283 | 35.23 | 1.00 | 0.99 |
| ampbsm | 1875 | 16.55 | 1917 | 16.15 | 1.02 | 0.98 |
| amppint2 | 1353 | 11.29 | 1329 | 11.00 | 0.98 | 0.97 |
| ampxhdl | 1059 | 12.90 | 1069 | 12.54 | 1.01 | 0.97 |
| apex6 | 1912 | 11.29 | 1800 | 11.38 | 0.94 | 1.01 |
| des | 8632 | 16.08 | 8407 | 16.09 | 0.97 | 1.00 |
| dflgrcb1 | 730 | 10.39 | 727 | 10.39 | 1.00 | 1.00 |
| fconrcb1 | 537 | 11.00 | 531 | 11.00 | 0.99 | 1.00 |
| frg2 | 2367 | 13.17 | 2363 | 13.27 | 1.00 | 1.01 |
| k2 | 2755 | 16.87 | 2778 | 16.82 | 1.01 | 1.00 |
| kcctlcb3 | 557 | 9.26 | 553 | 9.28 | 0.99 | 1.00 |
| pair | 3956 | 16.40 | 3931 | 16.29 | 0.99 | 0.99 |
| rot | 1651 | 19.17 | 1671 | 18.79 | 1.01 | 0.98 |
| sbiucb1 | 599 | 15.66 | 634 | 14.93 | 1.06 | 0.95 |
| tfaultcb1 | 439 | 6.80 | 440 | 6.52 | 1.00 | 0.96 |
| vda | 1522 | 13.31 | 1542 | 13.50 | 1.01 | 1.01 |
| x3 | 2033 | 10.97 | 2011 | 10.99 | 0.99 | 1.00 |
| aver | - | - | - | - | 1.00 | 0.99 |

Table 4.3: Effect of Iterative Improvement

**one iter:** one iteration of tree covering and fanout optimization
**three iters:** three iterations of tree covering and fanout optimization
**gain:** gain in area or in delay obtained by using iterative improvement
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

```
algorithm iterative_improvement
    foreach node v visited in topological order from inputs to outputs do {
    if if node is fanin node {
        apply tree covering
    } else {
        apply fanout optimization, taking all required times to be equal
    }
    do {
        foreach fanout node v visited in topological order from outputs to inputs do {
            apply fanout optimization
        }
        foreach fanin node v visited in topological order from inputs to outputs do {
            apply tree covering
        }
    } until network delay does not decrease
    foreach fanout node v visited in topological order from outputs to inputs do {
        apply fanout optimization in area recovery mode
    }
end iterative_improvement
```

Figure 4.5: A Simple Iterative Improvement Algorithm

The simple iterative improvement algorithm is sketched Figure 4.5. After an initial implementation has been built with tree covering, using the heuristic of section 4.3 to estimate delay through fanout nodes, the algorithm iterates fanout optimization and tree covering. As long as fanout optimization is done in topological order from outputs to inputs, fanout problems do not interact with each other, and the optimal solution with respect to fanout optimization can be achieved. During tree covering however, we need to evaluate the delay through fanout nodes. For that purpose, we use the fanout trees built at the previous fanout optimization pass. In this case, we need to take into account the polarity at which a signal is needed at a gate input, otherwise we obtain worse results. The results obtained by this algorithm after three iterations are reported in Table 4.3 and compared with the results obtained with only one iteration. The advantage of iterating is negligible on average, with a decrease in delay of 1% for no cost in area. In some examples, the delay

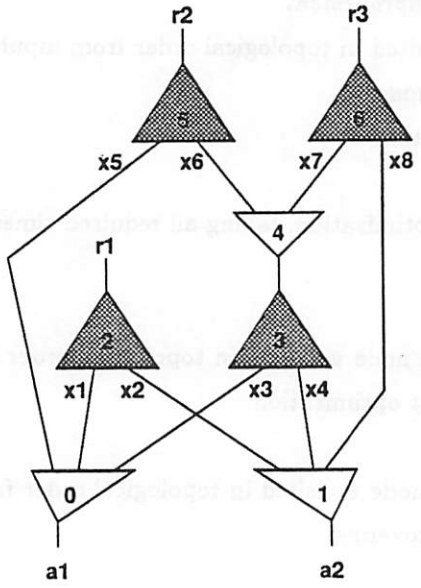


Figure 4.6: A More Complex Example

actually increases. This can be explained by the imprecision introduced by different rise and fall delays. There is no observable decrease in delay after the third iteration.

### 4.4.2 Optimality of Iterative Improvement

The example of section 4.2.5 shows a suboptimal implementation that cannot be improved by our iterative improvement algorithm. However, the heuristics we use to compute the initial implementation finds the optimal solution after the first iteration for this example. In other words, this example does not show any evidence that our approach is suboptimal. In this section, we study a similar but more complex example, and detail how the iterative improvement algorithm works on this example. Using the same delay model and hypotheses as in section 4.2.5, and solve the minimization problem analytically.

**A More Complex Example** As with the previous example, we are interested in computing the set of points $(r_1, r_2, r_3)$ that are realizable for given values $(a_1, a_2)$ of the arrival times, and dominated by no other solutions, i.e. the Pareto optimal points of this delay minimization problem. We use the circuit in Figure 4.6. To simplify the algebraic manipu-

lations, we perform the following substitution of variables:

$$\begin{aligned} R_i &= e^{r_i} \\ A_i &= e^{a_i} \\ X_i &= e^{x_i} \end{aligned}$$

The Pareto optimum points satisfy the following equations:

$$\begin{aligned} R_1 &= X_1 + X_2 \\ R_2 &= X_5 + X_6 \\ R_3 &= X_7 + X_8 \\ C_1 &= \frac{1}{X_1} + \frac{1}{X_3} + \frac{1}{X_5} - \frac{1}{A_1} = 0 \\ C_2 &= \frac{1}{X_2} + \frac{1}{X_4} + \frac{1}{X_8} - \frac{1}{A_2} = 0 \\ C_3 &= \frac{1}{X_6} + \frac{1}{X_7} - \frac{1}{X_3 + X_4} = 0 \end{aligned}$$

Since the optimization problem we are trying to solve in convex, the Pareto optimal points have a simple characterization. For each Pareto optimal point $p$, there is a hyperplane $H$ containing $p$ such that all points satisfying the constraints $(C_1, C_2, C_3)$ are on the same side of the hyperplane $H$. That is, for each Pareto optimal point $p$ there is a triplet $(a, b, c)$ such that $p$ is a solution of the following convex optimization problem:

$$\begin{aligned} &\min_X && aR_1(X) + bR_2(X) + cR_3(X) \\ &C_i(X) = 0, 1 \leq i \leq 3 \end{aligned}$$

Thus at a Pareto optimal point, there is a linear combination of $(dR_1, dR_2, dR_3)$ that is a linear combination of $(dC_1, dC_2, dC_3)$. In our example, this condition is equivalent to saying that the following matrix is of rank 2:

$$\begin{pmatrix} \frac{1}{X_1^2} & -\frac{1}{X_3^2} & 0 & \frac{1}{X_5^2} & 0 \\ -\frac{1}{X_2^2} & 0 & -\frac{1}{X_4^2} & 0 & -\frac{1}{X_8^2} \\ 0 & \frac{1}{(X_3+X_4)^2} & \frac{1}{(X_3+X_4)^2} & -\frac{1}{X_6^2} & \frac{1}{X_7^2} \end{pmatrix}$$

In turn, since the $X_i$ have a range limited to the interval $[0, +\infty)$ this condition is equivalent to the following set of equations:

$$X_1 X_6 X_8 = X_2 X_5 X_7$$

$$\begin{aligned} X_3X_6 &= X_5(X_3+X_4) \\ X_4X_7 &= X_8(X_3+X_4) \end{aligned}$$

which is equivalent to:

$$\begin{aligned} X_1X_4 &= X_2X_3 \\ X_3X_6 &= X_5(X_3+X_4) \\ X_4X_7 &= X_8(X_3+X_4) \end{aligned}$$

We can derive from these equations a parametric representation of the set of Pareto points for this example. We take as parameters the following quantities:

$$\begin{aligned} T &= \frac{X_3}{X_4} \\ U &= \frac{1}{X_2} \end{aligned}$$

The Pareto points are characterized by the following parametrized representation:

$$\begin{aligned} R_1 &= \frac{T+1}{U} \\ R_2 &= \frac{3(2T+1)}{\left(\frac{2T}{A_1}-\frac{1}{A_2}-U\right)} \\ R_3 &= \frac{3(T+2)}{\left(\frac{2}{A_2}-\frac{T}{A_1}-U\right)} \end{aligned}$$

The values of the intermediate variables can be rederived from the following equations:

$$\begin{aligned} \frac{1}{X_2} &= U \\ \frac{1}{X_4} &= \frac{1}{3}\left(\frac{T}{A_1}+\frac{1}{A_2}-2U\right) \\ \frac{1}{X_5} &= \frac{1}{3T}\left(\frac{2T}{A_1}-\frac{1}{A_2}-U\right) \\ \frac{1}{X_8} &= \frac{1}{3}\left(\frac{2}{A_2}-\frac{T}{A_1}-U\right) \\ X_1 &= TX_2 \\ X_3 &= TX_4 \\ X_6 &= (1+\frac{1}{T})X_5 \\ X_7 &= (1+T)X_8 \end{aligned}$$

| init | | optimum | $\rho = 0.0$ | | $\rho = 0.5$ | | $\rho = 0.9$ | |
|---|---|---|---|---|---|---|---|---|
| $a_1$ | $a_2$ | arrival | arrival | # iter | arrival | # iter | arrival | # iter |
| 0.0 | 0.0 | 2.398 | 2.485 | 1 | 2.402 | 12 | 2.398 | 79 |
| 2.0 | 0.0 | 3.885 | 3.968 | 3 | 3.899 | 13 | 3.892 | 84 |
| 4.0 | 0.0 | 5.805 | 5.885 | 2 | 5.821 | 17 | 5.813 | 79 |

Table 4.4: Iterative Improvement vs. Optimal Assignment

| | |
|---|---|
| **init:** | arrival times at the primary inputs |
| **optimum:** | optimum solution (derived analytically) |
| $\rho = x$**:** | iterative algorithm with rate $r_i = 1 - \rho^i$ |
| **arrival:** | worst case arrival time at the primary outputs |
| **# iter:** | number of iterations to reach convergence within $10^{-4}$ |

In particular, the Pareto optimal point satisfying $R_1 = R_2 = R_3$ is characterized by the following two equations:

$$T = \frac{A_1}{5}\left(3\left(\frac{1}{A_2} - \frac{1}{A_1}\right) + \sqrt{9\left(\frac{1}{A_2} - \frac{1}{A_1}\right)^2 + \frac{25}{A_1 A_2}}\right)$$

$$U = \frac{(T+1)}{(7T+4)}\left(\frac{2T}{A_1} - \frac{1}{A_2}\right)$$

**Application of the Iterative Algorithm** We applied our iterative algorithm to this example with three different pairs of values for the arrival times: $(0,0)$, $(2,0)$ and $(4,0)$. The results are reported in Table 4.4, under the column $\rho = 0$. In all cases, the iterative algorithm converges very rapidly, but the solution is not optimal.

Closer inspection of the solution produced by the algorithm reveals a reason why the algorithm does not converge towards an optimal solution. When arrival times at the primary inputs are equal, the iterative algorithm compensates the imbalances introduced by the long path through nodes 3 and 5 entirely at node 5, while the optimal solution distributes the compensation of the imbalance between node 0 and node 5. A similar phenomenon occurs with unbalanced arrival times at the primary inputs. The iterative algorithm attempts to correct imbalances in a greedy fashion, which is suboptimal in general.

To determine whether the greedy compensation of imbalances is the only source of suboptimality in the iterative algorithm, we performed the following experiment. We modified the iterative algorithm to limit the rate $r$ at which imbalances are corrected during a given iteration. A rate of $r = 50\%$ would mean that only half of the imbalance is corrected

by the algorithm. We modified the iterative algorithm to use a rate of $r_i = (1 - \rho^i)$ at the $i^{th}$ iteration, where $0 \leq \rho < 1$. The original algorithm corresponds to the special case of $\rho = 0$. By decreasing $r$ towards 0, the iterative algorithm is given more opportunity to distribute imbalance corrections properly throughout the network, at the cost of more iterations. The results for $\rho = 0.0$, $\rho = 0.5$ and $\rho = 0.9$ are given in Table 4.4.

The results obtained with $\rho = 0.0$ confirm our earlier experimental results that the simple iterative improvement algorithm converges very rapidly. In all cases, the results for $\rho = 0.0$ after only one iteration (our default iterative improvement algorithm) are within 2.5% of the optimum. By increasing $\rho$, we were able to improve further the quality of the final result at the cost of more iterations to reach a good quality result. For $\rho = 0.9$, the results after one iteration are only within 27% of the optimum, while the results after 100 iterations are within 0.2% of the optimum in all three examples.

These results provide, in a limited way, some solid evidence of the effectiveness of our simple iterative improvement algorithm, and confirmation that one iteration is sufficient to obtain good quality results.

### 4.4.3 Criticality Based Iteration

Yoshikawa *et al.* [44] have suggested a different technique to perform iterative improvement, based on the criticality of nodes. Their approach borrows the notion of $\epsilon$-critical subnetwork from Singh *et al.* [41]. The $\epsilon$-critical subnetwork of a Boolean network, for a given value of $\epsilon \geq 0$, is the subnetwork composed of the nodes and edges whose slack is within $\epsilon$ of the slack on a critical path. If $\epsilon = 0$, the $\epsilon$-critical subnetwork is simply composed of the set of critical paths of the network. If all paths of the $\epsilon$-critical subnetwork are sped up by some constant $\delta$, we can only guarantee that the circuit is sped up by $\min(\epsilon, \delta)$. The choice of $\epsilon$ is a tradeoff between the amount of computation to be done to optimize the network and the amount of improvement to be expected at each iteration.

Their algorithm alternates between two optimizations. The first optimization computes a minimum weight node cutset through an $\epsilon$-critical subnetwork, where the weights are used to direct the cutset on nodes with higher potential for delay reduction. The second optimization computes a minimum weight cutset across a region outside the $\epsilon$-critical subnetwork. In their approach, a node groups together a fanin node and the fanout node connected to its outputs.

Their main result is that by alternating between cutsets chosen within an $\epsilon$-critical subnetwork and cutsets taken outside an $\epsilon$-critical subnetwork, they obtain better results than by optimizing cutsets only chosen within an $\epsilon$-critical subnetwork. In their experiments, they obtained 36% improvement by alternating cutsets vs. 28% by using only critical cutsets. The reason why this phenomenon occurs can be explained as follows in the case of fanout optimization. A non-critical path may have a slack of 0.5 units of delay, which is enough to make it non-critical, but may be insufficient to allow a different selection of a fanout tree that would make the critical path faster. By optimizing the non-critical path, we may be able to increase the slack to 1 unit of delay, which may be sufficient to allow the selection of a different fanout tree that would make the critical path faster. A similar phenomenon can also occur in the case of fanin optimization.

Their results provide an additional justification of our global approach to iterative improvement. By isolating critical subnetworks and concentrating the effort of the local optimizers on critical nodes, global optimization algorithms produce lower quality results because they do not exploit fully the slack that could be made available on non critical parts of the circuit. Our approach, which does not distinguish at all between critical and non critical paths, has, in addition to its simplicity, the advantage of avoiding this problem. Its only drawback is that it may optimize more than necessary; but we have provided enough evidence in this work that area reclamation after delay optimization can be effective to maintain area increases within reasonable limits.

## 4.5 Beyond Tree-Based Optimization

In the previous section, we provided some evidence that we have reached the limits of the reductions we can obtain with tree-based delay optimization algorithms. In this section, we propose two additional delay optimization techniques that do not respect tree boundaries but are nevertheless simple modifications of tree-based delay optimization algorithms.

The first of these optimizations allows the duplication of fanin trees in order to provide the output of a fanin node in both polarities. This is achieved by implementing a fanin node with the fastest tree independently for each polarity. This optimization is discussed in section 4.5.1. We have implemented a simple version of this optimization and we provide experimental results.

The second of these optimizations, presented in section 4.5.2 allows the overlap of the implementation of fanin trees as shown in Figure 4.2 in order to reduce delay. This optimization can decrease delay significantly, but may lead to substantial area increases.

### 4.5.1 Phase Optimization by Tree Duplication

We showed in section 4.3 the role of phase assignment on the quality of tree-based delay optimization. By biasing tree covering in favor of a given polarity, we could slow down a circuit by an average 11%. By using a heuristic that attributes the same arrival times to signals of different polarities, we essentially make sure that the best phase is used for any given tree in the absence of any accurate information concerning the arrival times of signals. If all signals at the output of a fanout node are required with the same phase, there is no problem. This is not the case in general. If a signal is required under both polarities, at least one critical sink will receive the signal delayed by one inverter.

The problem occurs because we limit ourselves to one covering per fanin node. If we allow tree duplication, we can cover each fanin node twice, each cover producing the signal in a different polarity. These two trees may overlap and share some logic if there is no advantage in duplicating them further. This optimization has two advantages:

- in the case of small fanouts with signals needed in different polarities, it can remove one inverter delay if both trees can produce the signal with similar arrival times.
- for large fanouts, it decreases the need for deeper fanout trees by providing an additional source that can provide signals to one half of the sinks.

On the other hand, this optimization has two potential drawbacks: it may be wasteful in area and it does not preserve testability [38]. Unnecessary logic duplication can be controlled easily using the same technique that we use during fanout optimization. In the first pass of fanout optimization, we select the best solution at each node, which may require the use of tree duplication on the source node. In the second pass of fanout optimization, the area recovery pass, we can eliminate one cover of the source node if this transformation does not slow down the circuit. Removing redundancies is a more time consuming operation, but can only decrease delay and area, provided that there are no false paths in the circuit [33, 25].

We have implemented a simple version of this optimization. Our implementation has the following limitations: it does not take into account the cost of tree duplication in

| circuit | nodup | | dup | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1555 | 22.54 | 1.30 | 0.93 |
| C1908 | 1358 | 28.24 | 1553 | 26.40 | 1.14 | 0.93 |
| C2670 | 1796 | 20.86 | 1885 | 20.38 | 1.05 | 0.98 |
| C3540 | 2857 | 32.88 | 3024 | 33.22 | 1.06 | 1.01 |
| C5315 | 3693 | 29.31 | 4086 | 28.69 | 1.11 | 0.98 |
| C6288 | 8272 | 95.47 | 9485 | 92.66 | 1.15 | 0.97 |
| C7552 | 5322 | 27.83 | 5894 | 27.02 | 1.11 | 0.97 |
| alu4 | 1977 | 29.24 | 2052 | 30.55 | 1.04 | 1.04 |
| ampbpreg | 3289 | 35.56 | 3333 | 34.90 | 1.01 | 0.98 |
| ampbsm | 1875 | 16.55 | 2004 | 16.05 | 1.07 | 0.97 |
| amppint2 | 1353 | 11.29 | 1357 | 11.48 | 1.00 | 1.02 |
| ampxhdl | 1059 | 12.90 | 1128 | 12.35 | 1.07 | 0.96 |
| apex6 | 1912 | 11.29 | 1951 | 10.73 | 1.02 | 0.95 |
| des | 8632 | 16.08 | 9047 | 16.15 | 1.05 | 1.00 |
| dflgrcb1 | 730 | 10.39 | 755 | 9.19 | 1.03 | 0.88 |
| fconrcb1 | 537 | 11.00 | 587 | 10.62 | 1.09 | 0.97 |
| frg2 | 2367 | 13.17 | 2445 | 13.24 | 1.03 | 1.01 |
| k2 | 2755 | 16.87 | 2958 | 16.12 | 1.07 | 0.96 |
| kcctlcb3 | 557 | 9.26 | 563 | 8.47 | 1.01 | 0.91 |
| pair | 3956 | 16.40 | 4047 | 16.24 | 1.02 | 0.99 |
| rot | 1651 | 19.17 | 1651 | 18.73 | 1.00 | 0.98 |
| sbiucb1 | 599 | 15.66 | 609 | 15.19 | 1.02 | 0.97 |
| tfaultcb1 | 439 | 6.80 | 439 | 6.80 | 1.00 | 1.00 |
| vda | 1522 | 13.31 | 1618 | 13.28 | 1.06 | 1.00 |
| x3 | 2033 | 10.97 | 2035 | 10.57 | 1.00 | 0.96 |
| aver | - | - | - | - | 1.06 | 0.97 |

Table 4.5: Effect of Tree Duplication

**nodup:** tree covering and fanout optimization without tree duplication
**dup:** tree covering and fanout optimization with tree duplication
**gain:** gain in area or in delay obtained by using tree duplication
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

terms of extra fanout loads at the inputs of a fanin node; it uses a simple-minded allocation of sinks to the two sources made available by tree duplication; and it does not perform redundancy removal after tree duplication.

Given two sources, $S$ and $\overline{S}$, providing the same signal under differing polarities, for a fanout node, we perform fanout optimization as follows. We partition the sinks into two sets: the set $P$ of sinks with positive polarity, and the set $N$ of sinks with negative polarity. We try all 4 possible assignments of $P$ and $N$ to $S$ and $\overline{S}$ and perform fanout optimization in all 4 cases; i.e. we consider implementing the problem with $S$ alone, $\overline{S}$ alone, or both $S$ and $\overline{S}$. The best solution with smallest delay is retained. In case of equality, the solution which uses only one source is chosen and one tree is discarded.

We have implemented this optimization, and report the results of our experiments in Table 4.5. We achieved an average delay reduction of 3% for a average area increase of 6%. Additional delay reductions should be achievable by using a better sink assignment algorithm, and a more flexible tree duplication policy, allowing in particular the duplication of sources of the same polarity.

### 4.5.2 Allowing Overlaps between Trees

We gave in Figure 4.1 an example of a circuit that could be mapped more efficiently if overlaps between trees were allowed. However it is not clear how much delay improvement could be obtained in general by allowing fanin trees to overlap. We performed an experiment to answer this question.

Since `misII` tree covering algorithm is based on direct pattern matching, it is a simple matter to modify it to allow overlaps between trees. If overlaps between trees are allowed, the number and position of multiple fanout points can be modified arbitrarily by the covering algorithm. If a point $p$ is originally a multiple fanout point, we predict the arrival time at $p$ using the heuristics of section 4.3. Otherwise, the arrival time at $p$ is predicted as if $p$ had a fanout of 1, even if its fanout may increase due to an overlap between trees.

The effect of allowing tree overlaps is reported in Table 4.6. The reduction in delay obtained by allowing tree overlaps is significant: an average of 9%. However, as predicted, this reduction in delay comes with a heavy price in area: an average increase of 44%. These results indicate that better delays can be achieved by relaxing the constraints imposed by

| circuit | no overlap | | overlap | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1743 | 21.96 | 1.46 | 0.91 |
| C1908 | 1358 | 28.24 | 2042 | 25.69 | 1.50 | 0.91 |
| C2670 | 1796 | 20.86 | 2693 | 17.88 | 1.50 | 0.86 |
| C3540 | 2857 | 32.88 | 4461 | 31.04 | 1.56 | 0.94 |
| C5315 | 3693 | 29.31 | 6108 | 26.50 | 1.65 | 0.90 |
| C6288 | 8272 | 95.47 | 15083 | 86.18 | 1.82 | 0.90 |
| C7552 | 5322 | 27.83 | 8959 | 26.19 | 1.68 | 0.94 |
| alu4 | 1977 | 29.24 | 3289 | 29.10 | 1.66 | 1.00 |
| ampbpreg | 3289 | 35.56 | 5026 | 30.10 | 1.53 | 0.85 |
| ampbsm | 1875 | 16.55 | 2702 | 14.62 | 1.44 | 0.88 |
| amppint2 | 1353 | 11.29 | 1710 | 9.49 | 1.26 | 0.84 |
| ampxhdl | 1059 | 12.90 | 1489 | 11.46 | 1.41 | 0.89 |
| apex6 | 1912 | 11.29 | 2288 | 10.53 | 1.20 | 0.93 |
| des | 8632 | 16.08 | 14254 | 15.88 | 1.65 | 0.99 |
| dflgrcb1 | 730 | 10.39 | 901 | 9.53 | 1.23 | 0.92 |
| fconrcb1 | 537 | 11.00 | 669 | 8.77 | 1.25 | 0.80 |
| frg2 | 2367 | 13.17 | 3183 | 11.51 | 1.34 | 0.87 |
| k2 | 2755 | 16.87 | 4146 | 15.01 | 1.50 | 0.89 |
| kcctlcb3 | 557 | 9.26 | 740 | 8.17 | 1.33 | 0.88 |
| pair | 3956 | 16.40 | 5214 | 15.58 | 1.32 | 0.95 |
| rot | 1651 | 19.17 | 2168 | 17.30 | 1.31 | 0.90 |
| sbiucb1 | 599 | 15.66 | 873 | 15.35 | 1.46 | 0.98 |
| tfaultcb1 | 439 | 6.80 | 527 | 6.57 | 1.20 | 0.97 |
| vda | 1522 | 13.31 | 2390 | 11.96 | 1.57 | 0.90 |
| x3 | 2033 | 10.97 | 2650 | 10.72 | 1.30 | 0.98 |
| aver | - | - | - | - | 1.44 | 0.91 |

Table 4.6: Effect of Allowing Tree Overlaps

**no overlap:** tree covering and fanout optimization without tree overlaps
**overlap:** tree covering and fanout optimization allowing tree overlaps
**gain:** gain in area or in delay obtained by allowing tree overlap
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

tree-based delay optimization, but more work is required to control the penalty in area. Combining tree overlap with the tree duplication algorithm of section 4.5.1 has no effect, since allowing overlaps allows in particular the duplication of trees in different phases.

**Limiting Overlaps** It is possible to reduce the increase in area by limiting the overlap between trees. A simple way to enforce this limit is to allow overlaps to take place only over nodes that have a fanout of $K$ or less, for some constant $K$. This simple technique is an efficient way to reclaim area because most of the delay reduction can be obtained by allowing overlaps over nodes with small fanouts, and a large fraction of the area increase is due to overlaps allowed over nodes with large fanouts. The results of allowing tree overlaps only on nodes with five or fewer fanouts is reported in Table 4.7. By limiting overlaps, we achieved an average delay reduction of 8% for a cost in area of 28%.

## 4.6 Conclusion

We have provided an abstract framework for understanding tree-based delay optimization, and formulated the tree-based delay optimization problem as a convex optimization problem. Unfortunately, we do not have an analytic formula for the functions to be optimized. Also with the number of variables in the problem being usually large even for medium size circuits, it seems impractical to use this formulation directly. Instead we proposed a simple iterative technique that is guaranteed to produce solutions that cannot be improved by local transformations of the circuit. Unfortunately, we were able to exhibit a class of circuits showing that locally optimum implementations with respect to local circuit transformations can be arbitrarily far removed from the optimum solution. Even though the physical limitations of our model make this impossible in practice outside a fixed range of values, these examples strongly suggest that algorithms based on iterative improvement by local transformations are very limited in their optimization power. Finding an efficient optimization technique that would exploit directly the convexity of the search space remains an open problem.

On the practical side, we have shown experimentally the effect of heuristics to estimate the arrival time at a multiple fanout point; in particular, we have shown that these heuristics should give at the first application of tree covering the same delay for both signal polarities, while the actual delay value is less critical to the quality of the result. We

| circuit | no overlap | | overlap-5 | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1967 | 22.20 | 1.65 | 0.92 |
| C1908 | 1358 | 28.24 | 1890 | 26.59 | 1.39 | 0.94 |
| C2670 | 1796 | 20.86 | 2400 | 20.53 | 1.34 | 0.98 |
| C3540 | 2857 | 32.88 | 3799 | 31.51 | 1.33 | 0.96 |
| C5315 | 3693 | 29.31 | 5494 | 26.56 | 1.49 | 0.91 |
| C6288 | 8272 | 95.47 | 14534 | 87.28 | 1.76 | 0.91 |
| C7552 | 5322 | 27.83 | 8254 | 26.98 | 1.55 | 0.97 |
| alu4 | 1977 | 29.24 | 2542 | 28.24 | 1.29 | 0.97 |
| ampbpreg | 3289 | 35.56 | 4255 | 30.45 | 1.29 | 0.86 |
| ampbsm | 1875 | 16.55 | 2229 | 14.24 | 1.19 | 0.86 |
| amppint2 | 1353 | 11.29 | 1473 | 9.92 | 1.09 | 0.88 |
| ampxhdl | 1059 | 12.90 | 1244 | 11.82 | 1.17 | 0.92 |
| apex6 | 1912 | 11.29 | 2213 | 10.23 | 1.16 | 0.91 |
| des | 8632 | 16.08 | 9869 | 16.12 | 1.14 | 1.00 |
| dflgrcb1 | 730 | 10.39 | 879 | 9.41 | 1.20 | 0.91 |
| fconrcb1 | 537 | 11.00 | 667 | 9.55 | 1.24 | 0.87 |
| frg2 | 2367 | 13.17 | 2437 | 11.95 | 1.03 | 0.91 |
| k2 | 2755 | 16.87 | 3891 | 15.10 | 1.41 | 0.90 |
| kcctlcb3 | 557 | 9.26 | 684 | 8.09 | 1.23 | 0.87 |
| pair | 3956 | 16.40 | 4565 | 15.30 | 1.15 | 0.93 |
| rot | 1651 | 19.17 | 1997 | 17.17 | 1.21 | 0.90 |
| sbiucb1 | 599 | 15.66 | 820 | 15.00 | 1.37 | 0.96 |
| tfaultcb1 | 439 | 6.80 | 527 | 6.57 | 1.20 | 0.97 |
| vda | 1522 | 13.31 | 2057 | 12.20 | 1.35 | 0.92 |
| x3 | 2033 | 10.97 | 2312 | 10.62 | 1.14 | 0.97 |
| aver | – | – | – | – | 1.28 | 0.92 |

Table 4.7: Effect of Limiting Tree Overlaps

**no overlap:** tree covering and fanout optimization without tree overlaps
**overlap-5:** tree overlaps over nodes with five or fewer fanouts
**gain:** gain in area or in delay obtained by allowing limited tree overlaps
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

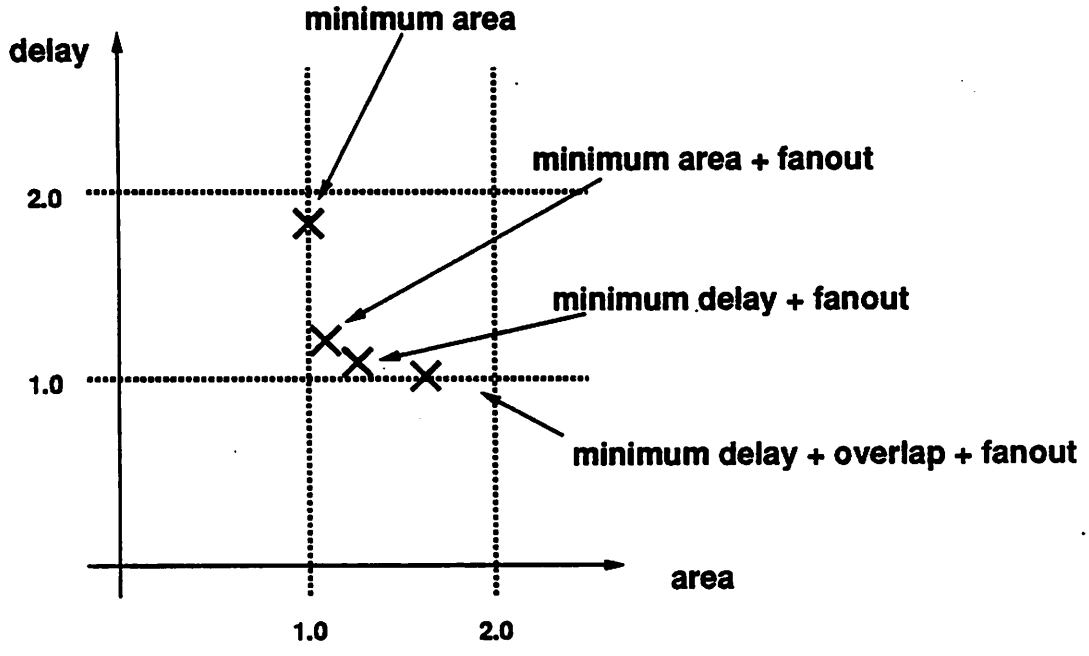


Figure 4.7: Area / Delay Tradeoff

have also proposed to use logic duplication to provide a signal in both phases at a multiple fanout point and shown that this technique can lead to additional delay reductions at little cost in area. More work needs to be done to preserve the testability of the circuit during this transformation.

In total, we have provided several methods to perform technology mapping, which provide a wide tradeoff between delay and area:

- minimum area tree covering with no fanout optimization.
- minimum area tree covering with fanout optimization.
- minimum delay tree covering with fanout optimization.
- minimum delay tree covering with limited overlaps and fanout optimization.

The average effect of these four methods in indicated in Figure 4.7. In area, all data are relative to the minimum area mapping. In delay, the data relative the minimum delay tree covering with overlaps and fanout optimization.

# Chapter 5

# Technology Independent Delay Optimizations

Monde nouveau, tu m'appartiens!
Sois donc à moi, ô beau pays!
Monde nouveau, tu m'appartiens!
Sois donc à moi!
— GIACOMO MEYERBEER, L'Africaine

## 5.1 Introduction

In the previous chapter, we presented several techniques for the efficient integration of tree covering and fanout optimization and provided empirical evidence of the efficiency of some of these methods. The purpose of this chapter is to examine the effect of *technology independent* logic transformations on circuit delay. We do not introduce any new technology independent algorithms to reduce delay. The originality and interest of this study comes from the fact that we now have at our disposal an efficient technology mapper on which we can rely to estimate delay. Similar data previously reported in the literature are usually of limited accuracy because they do not take into account the corrective effect of powerful optimization techniques such as fanout optimization.

The first step of this empirical study is to measure the effect of literal count minimization on circuit delay and area. Literal count minimization has been used as the main objective of technology independent optimization in logic synthesis because it correlates well with final circuit area [31]. The effect of literal count minimization is to simplify and

factorize the logic. By eliminating logic, simplification helps both in terms of delay and in terms of area. However factorization usually trades off delay for better area. We present in section 5.2 empirical data that confirms that literal count minimization has unpredictable effects on delay.

Since our objective is to minimize delay more than area, we should control the use of factorization, and concentrate our efforts on logic simplification instead. We have not explored fully the modification of the many technology independent transformations available in a logic synthesis tool such as misII, but we examine in section 5.3 the effect of a controlled use of a few of these transformations that could lead to a substantial area reduction at a much smaller cost in delay than uncontrolled literal count minimization.

However if delay is our principal objective, we should be able to produce fast circuits simply by flattening the logic. To flatten the logic, we collapse the Boolean network into a graph with only one level of nodes. Each of the nodes has a function associated with it that can be represented in sum-of-product form. In other words, collapsing to one level of nodes can be seen as collapsing to two levels of logic if no fanin or fanout limitation is enforced. Collapsing only helps in reducing delay for a certain set of circuits. For many circuits, collapsing introduces such a large amount of logic duplication that even delay increases. Nevertheless, when it applies, collapsing is a simple and very efficient technology independent delay optimization technique. We discuss the effect of collapsing in more detail in section 5.4.

Since network collapsing is such an effective technique at reducing circuit delay, it is worth investigating whether partial collapsing can be used when total collapsing is not practical. To decide which parts of a network should be collapsed, we use an algorithm developed by Lawler *et al.* [28]. Lawler's algorithm can be viewed as the technology independent analog of the extension of tree covering allowing overlaps between trees that we introduced in section 4.5.2. The main drawback of this algorithm is that it tends to increase area, but some of this area can be recovered by using the controlled literal count reduction techniques presented in section 5.3.

There has been some previous work in the area of logic restructuring for delay. The most notable effort in this direction was speedup, by Singh *et al.* [41], which performs local collapsing and factorization in order to reduce the number of levels of logic a signal has to traverse while controlling the increase in area incurred by collapsing. The work by Fishburn [15] is based on a similar idea, though the restructuring is performed differently. Another

technique, that is more efficient in terms of area but more computationally intensive, was proposed by Chen and Muroga [12]. This technique consists in exploiting the observability don't care set at a node to remove connections on the critical path. Berman *et al.* propose similar methods [6]. We present our adaptation of Lawler's algorithm in section 5.5 and compare it to speedup.

In the remainder of this section, we use the technology mapper to measure the area and delay of a circuit. For technology mapping, we use tree covering in delay minimization mode, followed by fanout optimization. We use the heuristic of section 4.3 but we do not allow overlaps between tree covers.

## 5.2 Effect on Delay of Minimizing Literal Count

We measured the effect of literal count minimization on circuit area and circuit delay after technology mapping. The results are reported in Table 5.1. All circuits were optimized using `misII` standard algebraic script [9] except `C2670`, which was optimized manually. As is apparent in the table, the circuits obtained from Intel (`dflgrcb1`, `fconrcb1`, `kcctlcb3`, `sbiucb1`, `tfaultcb1`) were already optimized, and little gain was achieved for this circuits. On average, minimizing literal count decreased area by 28% for no cost in delay. However, a more careful inspection of the data indicates that the effect on delay of literal count minimization is unpredictable, varying from a decrease of 22% to an increase of 26%, though the larger increases in delay correspond to significant decreases in area.

Obviously the techniques used in `misII` to reduce literal count are quite powerful. Unfortunately if they are used without discrimination, they may lead at times to substantial delay increases. This unpredictable behavior is undesirable and more work needs to be done to control the effect on delay of these optimizations.

## 5.3 Performance-Oriented Logic Simplification

A simple way to reduce circuit area without having to pay for an increase in delay is to reduce the literal count by using simplification only. A better way would be to allow, in addition to simplification, factorization along non critical paths. However, to perform this optimization reliably, we need a good technology independent delay estimator, and none is available at present. We have experimented with a simple `misII` script, shown in

| circuit | raw | | opt | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1764 | 26.14 | 1192 | 24.21 | 0.68 | 0.93 |
| C1908 | 1835 | 26.71 | 1358 | 28.24 | 0.74 | 1.06 |
| C2670 | 2399 | 24.76 | 1796 | 20.86 | 0.75 | 0.84 |
| C3540 | 3513 | 35.87 | 2857 | 32.88 | 0.81 | 0.92 |
| C5315 | 5285 | 31.13 | 3693 | 29.31 | 0.70 | 0.94 |
| C6288 | 8178 | 114.62 | 8272 | 95.47 | 1.01 | 0.83 |
| C7552 | 7326 | 28.75 | 5322 | 27.83 | 0.73 | 0.97 |
| alu4 | 2107 | 27.76 | 1977 | 29.24 | 0.94 | 1.05 |
| ampbpreg | 4607 | 45.36 | 3289 | 35.56 | 0.71 | 0.78 |
| ampbsm | 3840 | 19.32 | 1875 | 16.55 | 0.49 | 0.86 |
| amppint2 | 3347 | 9.48 | 1353 | 11.29 | 0.40 | 1.19 |
| ampxhdl | 1990 | 11.42 | 1059 | 12.90 | 0.53 | 1.13 |
| apex6 | 1963 | 10.98 | 1912 | 11.29 | 0.97 | 1.03 |
| des | 15166 | 16.15 | 8632 | 16.08 | 0.57 | 1.00 |
| dflgrcb1 | 732 | 9.08 | 730 | 10.39 | 1.00 | 1.14 |
| fconrcb1 | 537 | 11.00 | 537 | 11.00 | 1.00 | 1.00 |
| frg2 | 4265 | 10.90 | 2367 | 13.17 | 0.55 | 1.21 |
| k2 | 7616 | 13.34 | 2755 | 16.87 | 0.36 | 1.26 |
| kcctlcb3 | 555 | 9.26 | 557 | 9.26 | 1.00 | 1.00 |
| pair | 4347 | 18.03 | 3956 | 16.40 | 0.91 | 0.91 |
| rot | 1758 | 19.44 | 1651 | 19.17 | 0.94 | 0.99 |
| sbiucb1 | 634 | 15.97 | 599 | 15.66 | 0.94 | 0.98 |
| tfaultcb1 | 433 | 7.04 | 439 | 6.80 | 1.01 | 0.97 |
| vda | 3799 | 11.58 | 1522 | 13.31 | 0.40 | 1.15 |
| x3 | 2627 | 9.30 | 2033 | 10.97 | 0.77 | 1.18 |
| aver | - | - | - | - | 0.72 | 1.00 |

Table 5.1: Effect of Literal Count Minimization

**raw:** minimum delay technology mapping of unoptimized circuits
**opt:** minimum delay technology mapping of circuits optimized by misII
**gain:** gain in area or in delay obtained by literal count minimization
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

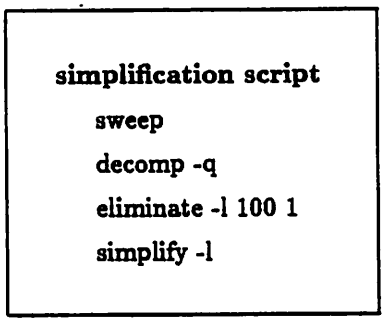


Figure 5.1: misII Logic Simplification Script

Figure 5.1. This script applies several commands [9, 8]:

- sweep: this command eliminates nodes with no fanin or no fanout. It simply removes from a network unnecessary nodes.
- decomp -q: this command factorizes nodes in a simple way with no concern for criticality. It is only used to break large nodes into smaller ones so that the other commands can run in a reasonable amount of cpu time.
- eliminate -l 100 -1: this command collapses a node into its fanout. The collapsing is only done if the node has a single fanout, and if the size of the resulting node does not exceed 100 cubes. The role of this command is to ensure that nodes are large enough for simplification to have some effect, but not too large so that simplification takes a reasonable amount of cpu time.
- simplify -l: this command runs espresso [7, 37], a two-level logic minimizer. The minimizer is given some information about the structure of the network, that allows it to simplify the logic function at a node in the context of the other nodes in the network. In particular, when simplifying a node $v$, the minimizer is allowed to change the inputs of $v$ if it simplifies the logic function at $v$. The -l option limits the minimizer to using as inputs of $v$ only nodes that are closer to the primary inputs than $v$. This restriction guarantees that the number of levels of logic in the network is not increased by the minimizer.

We measured the effect of the simple simplification script of Figure 5.1 on circuit area and delay after technology mapping. The results are reported in Table 5.2. The average effect of the simple simplification script is an area reduction of 9% and a delay reduction of 2%. The

| circuit | raw | | simplified | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1764 | 26.14 | 1274 | 20.45 | 0.72 | 0.78 |
| C1908 | 1835 | 26.71 | 1796 | 26.13 | 0.98 | 0.98 |
| C2670 | 2399 | 24.76 | 2079 | 23.78 | 0.87 | 0.96 |
| C3540 | 3513 | 35.87 | 3339 | 33.25 | 0.95 | 0.93 |
| C5315 | 5285 | 31.13 | 4471 | 30.65 | 0.85 | 0.98 |
| C6288 | 8178 | 114.62 | 7016 | 108.04 | 0.86 | 0.94 |
| C7552 | 7326 | 28.75 | 6294 | 27.06 | 0.86 | 0.94 |
| alu4 | 2107 | 27.76 | 1992 | 28.46 | 0.95 | 1.03 |
| ampbpreg | 4607 | 45.36 | 4412 | 43.89 | 0.96 | 0.97 |
| ampbsm | 3840 | 19.32 | 2945 | 16.97 | 0.77 | 0.88 |
| amppint2 | 3347 | 9.48 | 3101 | 10.68 | 0.93 | 1.13 |
| ampxhdl | 1990 | 11.42 | 1542 | 11.78 | 0.77 | 1.03 |
| apex6 | 1963 | 10.98 | 1956 | 11.25 | 1.00 | 1.02 |
| des | 15166 | 16.15 | 14430 | 16.19 | 0.95 | 1.00 |
| dflgrcb1 | 732 | 9.08 | 740 | 9.08 | 1.01 | 1.00 |
| fconrcb1 | 537 | 11.00 | 534 | 11.00 | 0.99 | 1.00 |
| frg2 | 4265 | 10.90 | 3388 | 12.40 | 0.79 | 1.14 |
| k2 | 7616 | 13.34 | 7620 | 13.62 | 1.00 | 1.02 |
| kcctlcb3 | 555 | 9.26 | 555 | 9.26 | 1.00 | 1.00 |
| pair | 4347 | 18.03 | 4002 | 16.91 | 0.92 | 0.94 |
| rot | 1758 | 19.44 | 1671 | 19.67 | 0.95 | 1.01 |
| sbiucb1 | 634 | 15.97 | 626 | 15.67 | 0.99 | 0.98 |
| tfaultcb1 | 433 | 7.04 | 433 | 7.04 | 1.00 | 1.00 |
| vda | 3799 | 11.58 | 3511 | 11.55 | 0.92 | 1.00 |
| x3 | 2627 | 9.30 | 2433 | 9.51 | 0.93 | 1.02 |
| aver | - | - | - | - | 0.91 | 0.98 |

Table 5.2: Effect of Simplification without Factorization

**raw:** minimum delay technology mapping of unoptimized circuits
**simplified:** minimum delay technology mapping of simplified circuits
**gain:** gain in area or in delay obtained by simplifying circuits
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

fluctuations in delay are less severe than with the standard script, varying from a reduction of 22% to an increase of 14%. The average reduction in area obtained by simplification alone is roughly a third of the reduction in area obtainable with the standard script. This is a heavy price to pay for a more controlled effect on delay.

## 5.4 Effect of Collapsing to Two Levels of Logic

A simple way to improve the performance of a circuit is to collapse it to two levels of logic. Unfortunately this technique has its limitations: for a large class of circuits, the area penalty is too large for collapsing to be practical. Nevertheless in some cases collapsing yields significant delay reductions for an acceptable increase in area. We were able to collapse 16 out of our 25 benchmark circuits. On each of the collapsed circuits we run `misII simplify -l` command to simplify the logic at each node after collapsing. The results after technology mapping are reported in Table 5.3.

## 5.5 Partial Collapsing for Delay Minimization

### 5.5.1 Lawler's Algorithm

Some circuits cannot be collapsed into two levels of logic without a large area penalty. However it is often possible to collapse these networks partially in order to reduce delay at a more moderate cost in area. To perform partial collapsing, we need an algorithm that determines which groups of nodes are to be collapsed into single nodes in order to decrease delay the through the network.

Unfortunately we do not have at out disposal a reasonably accurate technology independent delay model, such as the one proposed by Wallace *et al.* [43]. As a rough measure of delay, we use the number of logic levels a signal has to cross. To limit the inaccuracy of this delay model, we only apply it after having decomposed a network into simple gates. These simple gates are any of the four 2-input gates that can be represented as a 2-input NAND gate with possibly inverters at the inputs, representing one of the four following Boolean functions: $a + b$, $\overline{a} + b$, $a + \overline{b}$ or $\overline{a} + \overline{b}$.

To form the groups, we use a clustering algorithm due to Lawler that minimizes the number of levels of logic in the network after collapsing of the groups subject to the constraint that each group is formed of at most $K$ nodes. This algorithm generates possibly

| circuit | opt | | collapsed | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | * | * | – | – |
| C1908 | 1358 | 28.24 | * | * | – | – |
| C2670 | 1796 | 20.86 | * | * | – | – |
| C3540 | 2857 | 32.88 | * | * | – | – |
| C5315 | 3693 | 29.31 | * | * | – | – |
| C6288 | 8272 | 95.47 | * | * | – | – |
| C7552 | 5322 | 27.83 | * | * | – | – |
| alu4 | 1977 | 29.24 | 3595 | 14.22 | 1.82 | 0.49 |
| ampbpreg | 3289 | 35.56 | 4055 | 10.55 | 1.23 | 0.30 |
| ampbsm | 1875 | 16.55 | 3352 | 9.49 | 1.79 | 0.57 |
| amppint2 | 1353 | 11.29 | 3263 | 9.56 | 2.41 | 0.85 |
| ampxhdl | 1059 | 12.90 | 3770 | 11.39 | 3.56 | 0.88 |
| apex6 | 1912 | 11.29 | 3011 | 9.56 | 1.57 | 0.85 |
| des | 8632 | 16.08 | * | * | – | – |
| dflgrcb1 | 730 | 10.39 | 915 | 7.23 | 1.25 | 0.70 |
| fconrcb1 | 537 | 11.00 | 875 | 7.73 | 1.63 | 0.70 |
| frg2 | 2367 | 13.17 | 7128 | 10.64 | 3.01 | 0.81 |
| k2 | 2755 | 16.87 | 7902 | 12.08 | 2.87 | 0.72 |
| kcctlcb3 | 557 | 9.26 | 1126 | 6.39 | 2.02 | 0.69 |
| pair | 3956 | 16.40 | 19143 | 14.22 | 4.84 | 0.87 |
| rot | 1651 | 19.17 | * | * | – | – |
| sbiucb1 | 599 | 15.66 | 1830 | 11.26 | 3.06 | 0.72 |
| tfaultcb1 | 439 | 6.80 | 713 | 5.53 | 1.62 | 0.81 |
| vda | 1522 | 13.31 | 4544 | 9.95 | 2.99 | 0.75 |
| x3 | 2033 | 10.97 | 3027 | 9.56 | 1.49 | 0.87 |
| aver | – | – | – | – | 2.15 | 0.70 |

Table 5.3: Effect of Collapsing to Two Levels of Logic

**opt:** minimum delay technology mapping of circuits optimized by `misII`
**collapsed:** minimum delay technology mapping of the collapsed circuits
**gain:** gain in area or in delay obtained by collapsing circuits
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains
*: circuit not collapsible to two levels of logic

```
algorithm Lawler_clustering_algorithm
  { /* labeling step */
    foreach node v visited in topological order from inputs to outputs {
      if fanin(v) = ∅ then L = 0 else L = max_{u∈fanin(v)} label(u)
      k = |{u, u ∈ transitive_fanin(v), label(u) = L}|
      if k > K label(v) = L + 1 else label(v) = L
    }
  }
  { /* clustering step */
    foreach node v visited in topological order from outputs to inputs {
      if fanout(v) = ∅ then L = ∞ else L = min_{u∈fanout(v)} label(u)
      if label(v) < L {
        create a new cluster c
        c = {v} ∪ {u ∈ transitive_fanin(v), label(u) = label(v)}
      }
    }
  { /* collapsing step */
    foreach cluster c {
      root(c) = {v ∈ c, label(v) < max_{u∈fanout(v)} label(u)}
      foreach v ∈ root(c) {
        collapse into v all nodes in c ∩ transitive_fanin(v)
      }
    }
  }
end Lawler_clustering_algorithm
```

Figure 5.2: Lawler's Algorithm

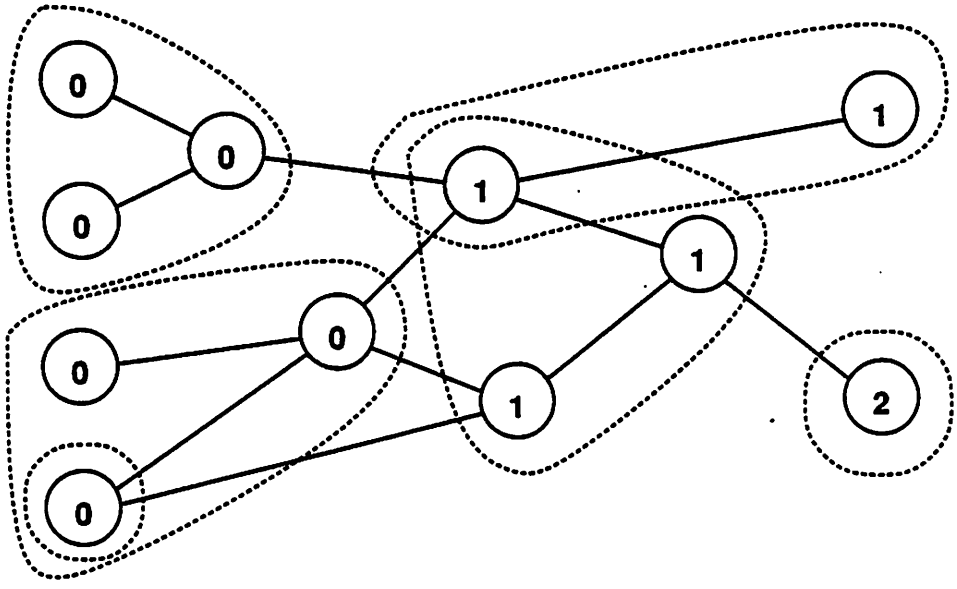


Figure 5.3: Example of Clustering

overlapping clusters, and clusters that have more than one output, requiring extra logic duplication during collapsing. For these reasons, this clustering algorithm usually increases circuit area. In the next subsection, we describe Lawler's algorithm in more detail.

Lawler's algorithm determines a minimum delay clustering of a network under the constraint that each cluster does not exceed a global capacity constraint $K$. The delay through a cluster is assumed to be the same for all clusters, and the size of a cluster is the number of nodes it contains. Lawler's algorithm can handle more general clustering problems, but the present formulation is sufficient for our purpose. Lawler's algorithm proceeds in two steps: a labeling step and a clustering step. We have added a third step to do the collapsing of the clusters. The algorithm is described in Figure 5.2.

The labeling step proceeds as follows. We visit the nodes in topological order, from inputs to outputs. For each node $v$, we compute the largest label $L$ of any of its fanins. If $v$ does not have any fanin, $L$ is taken to be equal to 0. We then compute the number $k$ of nodes in the transitive fanin of $v$ that are of label $L$. If $k$ exceeds $K$, the label of $v$ is set to be $L + 1$, otherwise it is set to be $L$. In the clustering step the nodes are visited in the reverse order, from outputs to inputs. If the label of a node $v$ is less than the labels of all the

nodes in its fanout, a new cluster is created, containing $v$ and all the nodes in the transitive fanin of $v$ with the same label as $v$. The collapsing step collapses the nodes of a cluster together. Some duplication is introduced within a cluster if a cluster has several outputs. One node is created per output, and every node of cluster contained in the transitive fanin of two more output nodes of a cluster is duplicated. Duplicating is also introduced across clusters, since a node may belong to several clusters as shown in Figure 5.3.

An example (from [28]) shows the application of the algorithm in Figure 5.3. In that example $K$, the cluster size limit, is set to 3. The labels are indicated inside the nodes and the clustering is shown by encirclings. As can be seen in this example, the algorithm may replicate some nodes. The labeling and clustering parts of the algorithm operates in $O(KN^2)$, where $N$ is the total number of nodes in the network and $K$ the maximum size of a cluster. The time complexity of the collapsing part is dependent on the logic function obtained at each node.

As can be observed in Figure 5.2 and Figure 5.3, Lawler's algorithm bears a strong similarity to tree covering with overlaps. The labeling step is the analog of the forward dynamic programming pass of tree covering. The clustering step is the analog of the gate selection pass of tree covering. In both algorithms, the nodes of the network are visited in the same order. In other words, the Lawler's clustering algorithm can be thought as a technology independent tree covering algorithm allowing overlaps, and in that sense is a natural extension of the algorithms we presented in the previous chapter. It suffers from the same problem as the tree covering algorithm allowing overlaps, as it often causes large area increases. More work needs to be done in this area to determine whether these area increases are necessary to reduce delay.

### 5.5.2 Effect of Clustering on Delay

In this section we examine the effect of the clustering algorithm on our set of benchmarks. We apply the clustering algorithm using the script of Figure 5.4. Most of the commands in this script have been introduced earlier. The new commands are:

- `tech_decomp -o 2`: this command decomposes the network into 2-input NAND gates possibly with inverters at one or both of the inputs.
- `resub -a -d`: applied to a network decomposed into 2-input NAND gates, this command detects if two copies of the same node are present in the network. If it is the

```
clustering script {
    sweep
    decomp -q
    tech_decomp -o 2
    resub -a -d
    sweep
    reduce_depth -S 8
    eliminate -1
    simplify -l
}
```

Figure 5.4: misII Clustering Script

```
speed-up script {
    sweep
    decomp -q
    speed_up -d 6 -m unit
}
```

Figure 5.5: misII Speed-up Script

case, one copy is removed and the fanout of the remaining node is increased by the fanout of the removed node.

- `reduce_depth -S 8`: this command performs a clustering and a collapse of the clusters using Lawler's algorithm. The maximum cluster size is set to 8. The actual cluster size used in the smallest cluster size with which the algorithm can get the same number of logic levels as with a cluster size limit of 8. Since the number of logic levels can only decrease as the cluster size limit is increased, we can find the smallest cluster size for a given number of logic levels by binary search. In addition, the search only needs to execute the labeling step of the clustering algorithm and is thus very fast.

The results of clustering on area and delay of circuits optimized for minimum

| circuit | opt | | clustered | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1597 | 20.19 | 1.34 | 0.83 |
| C1908 | 1358 | 28.24 | 1743 | 25.97 | 1.28 | 0.92 |
| C2670 | 1796 | 20.86 | 2517 | 19.40 | 1.40 | 0.93 |
| C3540 | 2857 | 32.88 | 4856 | 30.29 | 1.70 | 0.92 |
| C5315 | 3693 | 29.31 | 5703 | 28.02 | 1.54 | 0.96 |
| C6288 | 8272 | 95.47 | 10811 | 77.43 | 1.31 | 0.81 |
| C7552 | 5322 | 27.83 | 8000 | 24.19 | 1.50 | 0.87 |
| alu4 | 1977 | 29.24 | 2929 | 27.66 | 1.48 | 0.95 |
| ampbpreg | 3289 | 35.56 | 4703 | 20.67 | 1.43 | 0.58 |
| ampbsm | 1875 | 16.55 | 2693 | 13.25 | 1.44 | 0.80 |
| amppint2 | 1353 | 11.29 | 1704 | 9.48 | 1.26 | 0.84 |
| ampxhdl | 1059 | 12.90 | 1389 | 11.65 | 1.31 | 0.90 |
| apex6 | 1912 | 11.29 | 2559 | 10.44 | 1.34 | 0.92 |
| des | 8632 | 16.08 | 11992 | 17.39 | 1.39 | 1.08 |
| dflgrcb1 | 730 | 10.39 | 879 | 10.18 | 1.20 | 0.98 |
| fconrcb1 | 537 | 11.00 | 669 | 9.32 | 1.25 | 0.85 |
| frg2 | 2367 | 13.17 | 2955 | 10.56 | 1.25 | 0.80 |
| k2 | 2755 | 16.87 | 4857 | 14.48 | 1.76 | 0.86 |
| kcctlcb3 | 557 | 9.26 | 853 | 6.57 | 1.53 | 0.71 |
| pair | 3956 | 16.40 | 5396 | 15.17 | 1.36 | 0.92 |
| rot | 1651 | 19.17 | 2169 | 16.05 | 1.31 | 0.84 |
| sbiucb1 | 599 | 15.66 | 917 | 14.41 | 1.53 | 0.92 |
| tfaultcb1 | 439 | 6.80 | 562 | 5.65 | 1.28 | 0.83 |
| vda | 1522 | 13.31 | 2301 | 12.32 | 1.51 | 0.93 |
| x3 | 2033 | 10.97 | 2510 | 10.54 | 1.23 | 0.96 |
| aver | – | – | – | – | 1.39 | 0.87 |

Table 5.4: Effect of Clustering with a Maximum Cluster Size of 8

**opt:** minimum delay technology mapping of circuits optimized by `misII`
**clustered:** minimum delay technology mapping of the clustered circuits
**gain:** gain in area or in delay obtained by clustering circuits
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

| circuit | opt | | speedup | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 2203 | 21.21 | 1.85 | 0.88 |
| C1908 | 1358 | 28.24 | 1895 | 26.49 | 1.40 | 0.94 |
| C2670 | 1796 | 20.86 | 1836 | 19.98 | 1.02 | 0.96 |
| C3540 | 2857 | 32.88 | 3309 | 33.92 | 1.16 | 1.03 |
| C5315 | 3693 | 29.31 | 4712 | 23.52 | 1.28 | 0.80 |
| C6288 | 8272 | 95.47 | 9247 | 95.36 | 1.12 | 1.00 |
| C7552 | 5322 | 27.83 | 5849 | 26.19 | 1.10 | 0.94 |
| alu4 | 1977 | 29.24 | 2009 | 27.03 | 1.02 | 0.92 |
| ampbpreg | 3289 | 35.56 | 4338 | 22.46 | 1.32 | 0.63 |
| ampbsm | 1875 | 16.55 | 2041 | 15.29 | 1.09 | 0.92 |
| amppint2 | 1353 | 11.29 | 1341 | 11.37 | 0.99 | 1.01 |
| ampxhdl | 1059 | 12.90 | 1023 | 12.55 | 0.97 | 0.97 |
| apex6 | 1912 | 11.29 | 2068 | 9.83 | 1.08 | 0.87 |
| des | 8632 | 16.08 | 8767 | 16.40 | 1.02 | 1.02 |
| dflgrcb1 | 730 | 10.39 | 716 | 10.39 | 0.98 | 1.00 |
| fconrcb1 | 537 | 11.00 | 507 | 12.20 | 0.94 | 1.11 |
| frg2 | 2367 | 13.17 | 2555 | 11.83 | 1.08 | 0.90 |
| k2 | 2755 | 16.87 | 2696 | 16.30 | 0.98 | 0.97 |
| kcctlcb3 | 557 | 9.26 | 564 | 9.19 | 1.01 | 0.99 |
| pair | 3956 | 16.40 | 5655 | 16.94 | 1.43 | 1.03 |
| rot | 1651 | 19.17 | 2100 | 16.11 | 1.27 | 0.84 |
| sbiucb1 | 599 | 15.66 | 677 | 13.94 | 1.13 | 0.89 |
| tfaultcb1 | 439 | 6.80 | 453 | 6.97 | 1.03 | 1.03 |
| vda | 1522 | 13.31 | 1625 | 12.83 | 1.07 | 0.96 |
| x3 | 2033 | 10.97 | 2226 | 11.17 | 1.09 | 1.02 |
| aver | - | - | - | - | 1.12 | 0.94 |

Table 5.5: Effect of the speed_up Command

**opt:** minimum delay technology mapping of circuits optimized by misII
**speedup:** minimum delay technology mapping after speed_up
**gain:** gain in area or in delay obtained using speed_up
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

literal count is given in Table 5.4. Clustering achieves an average delay reduction of 13% for an average area increase of 39%. This technique performs better than tree covering with overlaps in both area and delay. For comparison, we also give in Table 5.5 the results obtained using misII speed_up routine. The script we used to run the speed_up routine is indicated in Figure 5.5. There is no need to use most of the commands in the previous script because the speed_up command performs its own decomposition into simple gates and area recovery. The speed_up command decreased delay by only 6% on average, for a moderate increase of 12% in area. In addition, speed_up does not perform very consistently: it actually increases the delay of 7 out of our set of 25 benchmarks. In contrast, the clustering algorithm increased delay in only one of the examples for a higher cost in area.

### 5.5.3 Area Recovery and Clustering

In this section we present two techniques to reduce the area increase due to clustering. The first technique is a modification of the labeling step of the clustering algorithm. The second technique is based on redundancy removal.

**Area Efficient Labeling Procedure** Lawler's clustering algorithm forces the duplication of nodes in two cases. First, when a node belongs to more than one cluster, it is duplicated in order to provide one copy per cluster. Secondly, when a cluster has several output nodes $(v_1, \ldots, v_k)$, all nodes in the cluster belonging to the transitive fanin of two or more of the nodes $(v_1, \ldots, v_k)$ need to be duplicated. Equivalently, we can consider than a cluster than has several output nodes is itself duplicated, one copy per output node. The duplicated nodes that, in a given copy, do not have any fanout can be removed.

Lawler's algorithm has the property of attributing to each node the smallest possible label that respects the cluster size constraint. In some cases a node can be attributed a larger label without violating the cluster size constraint and without causing the maximum node label to increase. An example of a situation where relabeling can occur is given in Figure 5.6. The effect of increasing the label of a node may be to remove an output node from a multiple output cluster, and so doing to reduce logic duplication.

We use a simple greedy heuristic to increase node labels. This heuristic is outlined in Figure 5.7. It is used after the labeling step and before the clustering step of the algorithm of Figure 5.2. The heuristic visits the nodes of the network in topological order from outputs to inputs. At each node $v$, the maximum label value is supposed to be available. If $v$ is a

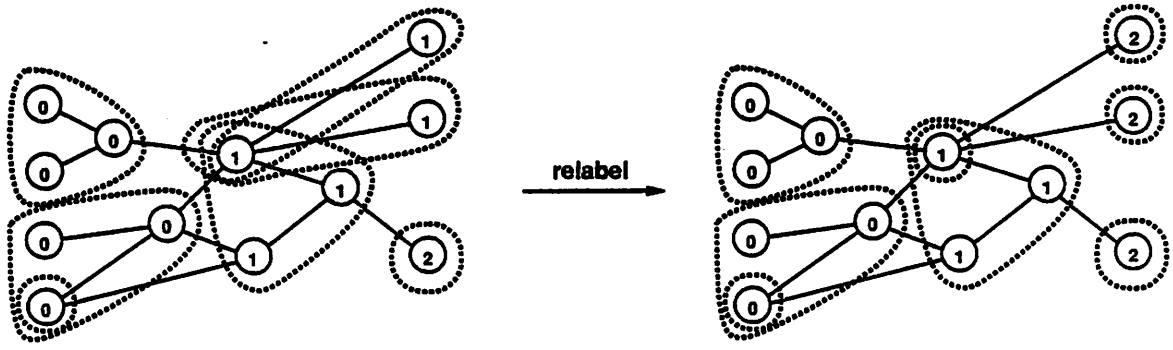


Figure 5.6: Example of Relabeling for Area

primary output, this value is simply the largest label value computed in the labeling step. If $v$ is not a primary output, this value is guaranteed to be available when $v$ is visited by the topological ordering of the nodes. The label of each node is given this maximum value. Then the largest label value the inputs of $v$ could have without forcing an increase of the label of $v$ is computed. This value is propagated to the inputs of $v$.

The effect of the relabeling heuristic is illustrated in Table 5.6. Relabeling reduces the average increase in area caused by clustering from 39% to 31%, and actually reduces delay by an additional percentage point, yielding an average delay reduction of 14%.

**Redundancy Removal** Clustering and collapsing may introduce a large number of redundancies. By removing these redundancies we can only reduce area and delay, assuming a static delay model. In the presence of *false paths*, i.e. paths in the circuit that cannot propagate any signal under any circumstances due to cancellation effects from side paths, redundancy removal may actually slow down the circuit by making a slow false path become active. The best known example of a circuit where this problem occurs is the carry-bypass adder. Removing logical redundancies from a carry-bypass adder has for effect to remove the bypass circuitry, transforming the fast carry-bypass adder into a slow ripple-carry adder. Redundancies can still be eliminated from circuits with false paths without slowing down the circuit, possibly at the cost of some logic duplication, using the algorithm of Keutzer Keutzer *et al.* [25]. In our experiments, we simply assume that all circuits have at least one

| circuit | opt | | relabel | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1597 | 20.19 | 1.34 | 0.83 |
| C1908 | 1358 | 28.24 | 1617 | 25.09 | 1.19 | 0.89 |
| C2670 | 1796 | 20.86 | 2243 | 19.91 | 1.25 | 0.95 |
| C3540 | 2857 | 32.88 | 4241 | 30.33 | 1.48 | 0.92 |
| C5315 | 3693 | 29.31 | 5079 | 27.89 | 1.38 | 0.95 |
| C6288 | 8272 | 95.47 | 10704 | 77.28 | 1.29 | 0.81 |
| C7552 | 5322 | 27.83 | 8302 | 24.34 | 1.56 | 0.87 |
| alu4 | 1977 | 29.24 | 2777 | 27.05 | 1.40 | 0.93 |
| ampbpreg | 3289 | 35.56 | 4162 | 20.66 | 1.27 | 0.58 |
| ampbsm | 1875 | 16.55 | 2390 | 12.69 | 1.27 | 0.77 |
| amppint2 | 1353 | 11.29 | 1731 | 9.29 | 1.28 | 0.82 |
| ampxhdl | 1059 | 12.90 | 1259 | 11.62 | 1.19 | 0.90 |
| apex6 | 1912 | 11.29 | 2429 | 10.12 | 1.27 | 0.90 |
| des | 8632 | 16.08 | 11540 | 17.40 | 1.34 | 1.08 |
| dflgrcb1 | 730 | 10.39 | 803 | 10.18 | 1.10 | 0.98 |
| fconrcb1 | 537 | 11.00 | 645 | 9.32 | 1.20 | 0.85 |
| frg2 | 2367 | 13.17 | 2825 | 10.57 | 1.19 | 0.80 |
| k2 | 2755 | 16.87 | 4784 | 14.36 | 1.74 | 0.85 |
| kcctlcb3 | 557 | 9.26 | 810 | 6.39 | 1.45 | 0.69 |
| pair | 3956 | 16.40 | 5274 | 14.95 | 1.33 | 0.91 |
| rot | 1651 | 19.17 | 1966 | 15.69 | 1.19 | 0.82 |
| sbiucb1 | 599 | 15.66 | 837 | 14.41 | 1.40 | 0.92 |
| tfaultcb1 | 439 | 6.80 | 513 | 5.65 | 1.17 | 0.83 |
| vda | 1522 | 13.31 | 2269 | 12.32 | 1.49 | 0.93 |
| x3 | 2033 | 10.97 | 2440 | 9.82 | 1.20 | 0.90 |
| aver | - | - | - | - | 1.31 | 0.86 |

Table 5.6: Effect of Relabeling Heuristic

**opt:** minimum delay technology mapping of circuits optimized by `misII`
**relabel:** minimum delay technology mapping of the relabeled, clustered circuits
**gain:** gain in area or in delay obtained by relabeled clustering
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains

```
algorithm relabeling_heuristic
    max_label = max_{v∈PO} label(v)
    foreach node v {
        max_label(v) = max_label
    }
    foreach node v visited in topological order from outputs to inputs {
        foreach node u ∈ FANIN(v) {
            if (label(v) < max_label(v) and label(u) == label(v)) {
                increment = max_label(v) − label(u)
            } else {
                increment = max(0, max_label(v) − label(u) − 1)
            }
            max_label(u) = min(max_label(u), label(u) + increment)
        }
        label(v) = max_label(v)
    }
end relabeling_heuristic
```

Figure 5.7: Relabeling Procedure for Reducing Logic Duplication

active critical path. This hypothesis is satisfied by most circuits. Moreover, all circuits can be made to satisfy this hypothesis by using Keutzer's algorithm.

We removed all redundancies from circuits that were partially collapsed using Lawler's algorithm. We applied the relabeling heuristic described in the previous section. The results are reported in Table 5.7. To identify redundancies, we used an improved version of the automatic test pattern generation program Socrates [39] developed by Jacoby *et al.* [23]. After redundancy removal, we ran the following `misII` commands: `sweep; eliminate -1; simplify -1`, except on C3540. For C3540, `eliminate -1` causes a large increase in the sum of product representation of the circuit which makes the use of Jacoby's redundancy removal program impractical. We were unable to complete the removal of all redundancies on one circuit, C6288, after 72 hours of cpu time on a DEC-3100. Jacoby's redundancy removal program performs some limited form of factorization, which may bias

| circuit | opt | | rr | | gain | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| C1355 | 1192 | 24.21 | 1417 | 19.98 | 1.19 | 0.83 |
| C1908 | 1358 | 28.24 | 1570 | 25.94 | 1.16 | 0.92 |
| C2670 | 1796 | 20.86 | 2216 | 17.36 | 1.23 | 0.83 |
| C3540 † | 2857 | 32.88 | 3836 | 28.68 | 1.34 | 0.87 |
| C5315 | 3693 | 29.31 | 4909 | 26.91 | 1.33 | 0.92 |
| C6288 | 8272 | 95.47 | * | * | – | – |
| C7552 | 5322 | 27.83 | 9550 | 25.34 | 1.79 | 0.91 |
| alu4 | 1977 | 29.24 | 2209 | 26.55 | 1.12 | 0.91 |
| ampbpreg | 3289 | 35.56 | 2541 | 13.92 | 0.77 | 0.39 |
| ampbsm | 1875 | 16.55 | 2191 | 11.06 | 1.17 | 0.67 |
| amppint2 | 1353 | 11.29 | 1761 | 9.30 | 1.30 | 0.82 |
| ampxhdl | 1059 | 12.90 | 1007 | 10.93 | 0.95 | 0.85 |
| apex6 | 1912 | 11.29 | 2397 | 10.03 | 1.25 | 0.89 |
| des | 8632 | 16.08 | 11313 | 17.38 | 1.31 | 1.08 |
| dflgrcb1 | 730 | 10.39 | 786 | 10.18 | 1.08 | 0.98 |
| fconrcb1 | 537 | 11.00 | 680 | 8.80 | 1.27 | 0.80 |
| frg2 | 2367 | 13.17 | 2420 | 10.30 | 1.02 | 0.78 |
| k2 | 2755 | 16.87 | 4505 | 13.99 | 1.64 | 0.83 |
| kcctlcb3 | 557 | 9.26 | 793 | 6.88 | 1.42 | 0.74 |
| pair | 3956 | 16.40 | 5145 | 14.10 | 1.30 | 0.86 |
| rot | 1651 | 19.17 | 1911 | 16.20 | 1.16 | 0.85 |
| sbiucb1 | 599 | 15.66 | 782 | 12.01 | 1.31 | 0.77 |
| tfaultcb1 | 439 | 6.80 | 486 | 6.38 | 1.11 | 0.94 |
| vda | 1522 | 13.31 | 2224 | 12.54 | 1.46 | 0.94 |
| x3 | 2033 | 10.97 | 2465 | 9.51 | 1.21 | 0.87 |
| aver | – | – | – | – | 1.23 | 0.83 |

Table 5.7: Effect of Redundancy Removal after Clustering

**opt:** minimum delay technology mapping of circuits optimized by `misII`
**rr:** minimum delay technology mapping of the relabeled, clustered circuits after redundancy removal
**gain:** gain in area or in delay obtained by clustering and redundancy removal
**area:** area of the circuit (MCNC *lib2* data divided by common divisor 464)
**delay:** delay of the circuit (MCNC *lib2* data in nanoseconds)
**aver:** geometric average of the gains
*: timeout after 72 hours of cpu time on a DEC 3100
†: `eliminate -1` was not applied before redundancy removal

our results slightly in favor of area. Redundancy removal was effective at reducing the area penalty incurred by clustering. On average, clustering followed by redundancy removal increased area by 23% for a reduction in delay of 17%.

## 5.6 Conclusion

We presented in this chapter several technology independent techniques to reduce circuit delay. The simplest of these techniques, which consists in collapsing a circuit to two levels of logic, is only applicable for a restricted class of circuits. For these circuits, collapsing may yield impressive delay reductions. However, for most circuits, the cost in area is too large for collapsing to be practical. As an alternative to full collapsing we introduced a simple clustering technique that allows partial collapsing, and realizes a compromise between area increase and delay reduction. This clustering technique fits naturally in this thesis, as it can be seen as a technology independent version of a tree covering algorithm allowing overlaps. Comparing clustering with speed_up, we saw that clustering was more wasteful in area, but gave better delays and was more consistent. Clustering can be rendered less costly in area and even more efficient in delay by reducing overlaps between clusters whenever it does not increase the number of levels of logic in the clustered network, and by using of redundancy removal. Using clustering and redundancy removal, we were able to obtain, in some cases, circuits that were faster than their collapsed versions for a fraction of the area.

We demonstrated that there is still a lot of potential for delay minimization beyond technology mapping. More work needs to be done to exploit this potential at a more moderate cost in area and cpu time than the small set of techniques presented in this chapter.

# Chapter 6

# Conclusion

Bornons ici cette carrière.
Les longs ouvrages me font peur.
Loin d'épuiser une matière,
On n'en doit prendre que la fleur.
— LA FONTAINE

The main results of this work are as follows. We provided an exact solution to the minimum delay tree covering problem based on piece-wise linear functions. We performed an extensive study of fanout optimization heuristics, presented new complexity results, and introduced a spectrum of fanout optimization algorithms. We developed a simple algorithm to apply fanout optimization throughout an entire network that reduces delay at a very moderate cost in area. To study the integration of tree covering and fanout optimization, we introduced a technology independent delay model that characterizes precisely suboptimalities due to imbalances in a network. This is the first technology independent delay model that models the delay through a node as a function of the arrival time distribution at a node. In addition, this delay model can be used to derive analytically optimal solutions in simple cases which can be used to assess the optimality of algorithms. We showed the importance of the technique used to evaluate the arrival times at the input of trees before fanout optimization, and presented an efficient heuristic to solve this problem. We also experimented with allowing tree covers to overlap, and showed significant delay reductions with this technique. Finally we investigated technology independent delay optimization techniques based on partial or total collapsing of logic, and showed that further delay reductions can be achieved with these techniques possibly at a higher cost in area.

A surprising conclusion of our work is that it is important to *ignore* critical paths

when performing delay optimization during logic synthesis. As confirmed by the experiments of Yoshikawa *et al.* [44], delay reduction on non-critical paths can create additional slacks on those paths that can be exploited to reduce delay through critical paths. By concentrating on critical paths only, delay optimization algorithms condemn themselves to suboptimal solutions.

We now have at our disposal a spectrum of delay optimization techniques. Fanout optimization is the cheapest technique in terms of area consumption, and should be given top priority. Tree covering for delay comes in second. The area-delay tradeoff potential of tree covering depends more heavily on the quality of the library used by the technology mapper. In some cases tree covering can outperform fanout optimization, though we were not able to demonstrate this fact in this thesis due to the confidentiality of some of our libraries. Allowing overlaps between tree covers as well as technology independent collapsing algorithms followed by redundancy removal add to the arsenal of delay minimization techniques.

More work needs to be done in technology independent delay optimization techniques. There are three main avenues of research: the development of more accurate technology independent delay models than the ones currently in use; the improvement of collapsing algorithms in terms of area; the improvement of techniques based on kernel extraction (speed_up) or observability don't-care sets in terms of cpu speed. In particular, it would be interesting to investigate the use of the technology independent delay model introduced in chapter 4 or a model derived on similar ideas to drive technology independent delay reduction algorithms. This delay model is the first to propose a way to take into account imbalances in arrival times as they occur in networks. A more fundamental issue would be to understand when logic duplication is needed for delay reduction (we know that redundancy is not needed, even in the presence of false paths [25]) in order to find more economical ways to perform partial collapsing or tree overlapping.

# Bibliography


[1] A. Aho, M. Ganapathi, and S. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[2] A. Aho, S. C. Johnson, and J. Ullman. Code generation for expressions with common subexpressions. *Journal of the Association for Computing Machinery*, 24(1):146–160, 1977.

[3] A. V. Aho and M. Ganapathi. Efficient Tree Pattern Matching: an Aid to Code Generation. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 334–340, January 1985.

[4] K. Bartlett, W. Cohen, A. De Geus, and G. Hacthel. Synthesis and Optimization of Multilevel Logic under Timing Constraints. *IEEE Transactions on CAD*, 5(4):582–596, October 1986.

[5] C. L. Berman, J. L. Carter, and K. F. Day. The Fanout Problem: From Theory to Practice. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference*, pages 69–99. MIT Press, March 1989.

[6] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, and L. H. Trevillyan. Efficient techniques for timing correction. In *ISCAS*, pages 415–419, 1990.

[7] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[8] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

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

[10] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[11] D. R. Chase. An Improvement to Bottom-up Tree Pattern Matching. In ACM, editor, *Symposium on Principles of Programming Languages*, pages 168–177, January 1987.

[12] K. C. Chen and S. Muroga. Timing Optimization for Multi-Level Combinational Networks. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 339–344, 1990.

[13] J. A. Darringer, D. Brand, W. H. Joyner, and L. Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Development*, 28(5):537–545, September 1984.

[14] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology Mapping in MIS. In *Proc. of the ICCAD-87*, pages 116–119, November 1987.

[15] J. P. Fishburn. A Depth-Decreasing Heuristic for Combinational Logic: or How to Convert a Ripple-Carry Adder into a Carry-Lookahead Adder or Anything In-Between. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 361–364, 1990.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Mathematical Sciences Series. Freeman, 1979.

[17] L. A. Glasser and L. P. J. Hoyte. Delay and Power Optimization in VLSI Circuits. In *21st ACM/IEEE Design Automation Conference*, pages 529–535, 1984.

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

[19] D. Gregory, K. Bartlett, A. de Geus, and Hachtel. G. SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic. In *23rd ACM/IEEE Design Automation Conference*, pages 79–85, 1986.

[20] C. M. Hoffmann and M. J. O'Donnell. Pattern Matching in Trees. *Journal of the Association for Computing Machinery*, 29(1):68–95, January 1982.

[21] M. Hofmann and J. K. Kim. Delay Optimization of Combinational Static CMOS Logic. In *24th ACM/IEEE Design Automation Conference*, pages 125–132, 1987.

[22] 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.

[23] R. Jacoby, P. Moceynas, H. Cho, and G. Hachtel. New ATPG Techniques for Logic Optimization. In *ICCAD*, pages 548–551, 1989.

[24] K. Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *Proceedings of the 24th Design Automation Conference*, pages 341–347. ACM/IEEE, June 1987.

[25] K. Keutzer, S. Malik, and A. Saldanha. Is redundancy necessary to reduce delay? In *Proceedings of the Design Automation Conference*, pages 228–234, June 1990. Accepted for publication, IEEE Transactions on Computer Aided Design.

[26] K. Keutzer and M. Vancura. Timing Optimization in a Logic Synthesis System. In G. Saucier, editor, *Proceedings of International Workshop on Logic and Arch. Synthesis for Silicon Compilers*, pages 1–13, Grenoble, France, May 1988. Inst. Nat. Polytechnique.

[27] K. Keutzer and W. Wolf. Anatomy of a Hardware Compiler. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 95–104. ACM, June 1988.

[28] E. L. Lawler, K. L. Levitt, and J. Turner. Module clustering to minimize delay in digital networks. *IEEE Transactions on Computers*, C-18(1):47–57, January 1969. 1969.

[29] Mario Lega. Private communication. AT&T Bell Laboratories, October 1990.

[30] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In R. Bryant, editor, *3rd Caltech Conference on Very Large Scale Integration*, pages 87–116, 1983.

[31] M. Lightner and W. Wolf. Experiments in Logic Optimization. In *ICCAD*, pages 286–289, 1990.

[32] R. Lisanke. Logic Synthesis and Optimization Benchmarks User Guide Version 2.0. Technical report, MCNC, P.O. Box 12889, Research Triangle Park, NC 27709, December 1988.

[33] P. McGeer. *On the Interaction of Functional and Timing Behavior of Combinational Logic Circuits.* PhD thesis, U.C. Berkeley, November 1989.

[34] F. W. Obermeier and R. H. Katz. Combining Circuit Level Changes with Electrical Optimization. In *ICCAD-88*, pages 218–221. IEEE, 1988.

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

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

[37] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):727–750, September 1987.

[38] A. Saldanha, R. K. Brayton, A. L. Sangiovanni-Vincentelli, and K. T. Cheng. Timing Optimization with Testability Considerations. In *ICCAD*, pages 460–463, 1990.

[39] M. Schulz, E. Trischler, and T. Sarfert. SOCRATES: a Highly Efficient ATPG System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(1):126–137, January 1988.

[40] K. J. Singh and A. Sangiovanni-Vincentelli. A Heuristic Algorithm for the Fanout Problem. In *DAC*, pages 357–360, June 1990.

[41] K. J. Singh, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing Optimization of Combinational Logic. In *ICCAD-88*, pages 282–285. IEEE, 1988.

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

[43] D. Wallace. High-Level Delay Estimation for Technology-Independent Logic Equations. In *ICCAD*, pages 188–191, 1990.

[44] K. Yoshikawa and H. Ichiryu. Timing Optimization by Technology Mapping and Fanout Adjustment. In *submitted to DAC*, 1991.