UNIVERSITY OF CALIFORNIA, SAN DIEGO

Algorithms for Performance Driven Design of Integrated Circuits

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science and Engineering

by

John Patrick Lillis

Committee in charge:

Professor Chung-Kuan Cheng, Chair
Professor Te C. Hu
Professor Russell Impagliazzo
Professor Walter H. Ku
Professor Ting-Ting Lin
Professor Alex Orailoğlu

1996

The dissertation of John Patrick Lillis is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

1996

To my wife Rebecca

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I must express my gratitude to my advisor Professor Chung-Kuan Cheng. Prior to joining Professor Cheng's group I had considered leaving the Ph.D. program; had it not been for the sheer persistence with which he encouraged me to continue my Ph.D. work and join his group, I surely would have left the program. I am truly indebted to him for this persistence and cannot overstate the impact it has had on my career and life. Of course I am also grateful to him for the patient guidance he provided during my time as his student; I learned a great deal from him. His affable personality and even-handed management style made working with him a pleasure.

I must also thank my committee members. Professor Ting-Ting Lin was always supportive of my work (both financially and intellectually); for this I thank her. I thank Professor T.C. Hu for sharing with me his research philosophies and for many helpful discussions (the depth of which I would often only realize well after the fact). Professor Russell Impagliazzo's theoretical insights often helped in my research. I also thank Professors Alex Orailoğlu and Walter Ku for their encouragement and advice.

I thank my colleagues associated with the CAD/VLSI group at UCSD. The group of students, postdocs and visiting scholars has grown so large, that I cannot hope to name them all, but people like Lung-Tien Liu, Ming-Ter Kuo, Ian Harris and Bob Carragher (to list some old-timers) were always friendly and made for a relaxed work environment.

Many fellow students in the CSE department helped make my stay at UCSD frequently enjoyable and always bearable. I'll not soon forget wandering discussions about everything from combinatorics to politics with Matt Clegg over beer and pool at the Hungry Stick. Lunches with Darren Atkinson, David Morgenthaler and Val Donaldson always made for a friendly and often amusing break from the lab.

Of course, I am grateful to my parents Jim and Shirley Lillis for their support and encouragement. It is comforting to know that, whatever decisions I make, they will always support me.

Finally, I thank my wife Rebecca Malley for her patience and support during our stay in San Diego. I cannot imagine traversing the ups and downs of life, let alone graduate school, without her.

VITA

August 19, 1967	Born, Seattle, Washington
August 1989	B.S. Computer Science, University of Washington
1990–1992	Teaching Assistant Department of Computer Science and Engineering University of California, San Diego
December 1992	M.S. Computer Science University of California, San Diego
1993–1996	Research Assistant University of California, San Diego
November 1994	Candidate of Philosophy University of California, San Diego
August 1996	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

J. Lillis, C.-K. Cheng, T.-T. Lin, C.-Y. Ho, "New Techniques for Performance Driven Routing with Explicit Area/Delay Tradeoff and Simultaneous Wire Sizing," *Proc. 33rd ACM/IEEE Design Automation Conference,* Las Vegas, Jun. 1996, pp. 395-400.

J. Li, J. Lillis, L.-T. Liu, C.-K. Cheng, "New Spectral Linear Placement and Clustering Approach," *Proc. 33rd ACM/IEEE Design Automation Conference,* Las Vegas, Jun. 1996, pp. 88-93.

J. Lillis, C.-K. Cheng, T.-T. Lin, "Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model," *Proc. IEEE International Conference on Computer-Aided Design,* Santa Clara, Nov. 1995, pp. 138-143.

J. Li, J. Lillis, C.-K. Cheng, "Linear Decomposition Algorithm for VLSI Design Applications," *Proc. IEEE International Conference on Computer-Aided Design,* Santa Clara, Nov. 1995, pp. 223-228.

J. Lillis, C.-K. Cheng, T.-T. Lin, "Simultaneous Routing and Buffer Insertion for High Performance Interconnect," *Proc. Sixth IEEE Great Lakes Symposium on VLSI,* Ames, Iowa, Mar. 1996, pp. 148-153.

J. Lillis, C.-K. Cheng, T.-T. Lin, "Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model," *IEEE Journal of Solid State Circuits*, vol. 31, no. 3, March 1996, pp. 437-447.

J. Lillis, C.-K. Cheng, T.-T. Lin, "Algorithms for Optimal Introduction of Redundant Logic for Timing and Area Optimization," *Proc. IEEE International Symposium on Circuits and Systems,* Atlanta, May 1996, v. 4, pp. 452-455.

J. Lillis, C.-K. Cheng, T.-T. Lin, "Optimal and Efficient Buffer Insertion and Wire Sizing," *Proc. IEEE 1995 Custom Integrated Circuits Conference,* Santa Clar, May 1995, pp. 259-262.

L.-T. Liu, M. Shi, J. Lillis, C.-K. Cheng, "Data Flow Partitioning for Clock Period and Latency Minimization," To appear in *IEEE Transactions on Circuits and Systems.*

FIELDS OF STUDY

Major Field: Computer Science and Engineering
Studies in Computer-Aided Design for VLSI.
Professor Chung-Kuan Cheng and Ting-Ting Lin
University of California, San Diego

ABSTRACT OF THE DISSERTATION

Algorithms for Performance Driven Design
of Integrated Circuits

by

John Patrick Lillis
Doctor of Philosophy in Computer Science and Engineering
University of California, San Diego, 1996
Professor Chung-Kuan Cheng, Chair

Recent scaling trends in fabrication of *Very Large Scale Integrated* (VLSI) circuits have resulted in interconnect delay being a significant bottleneck in system performance. This trend has resulted in a need to study new timing optimization problems in the physical design of VLSI circuits; additionally it has become important to re-visit many of the classical problems in the field. This thesis presents algorithmic techniques for three such problems: (1) Optimal *wire sizing* and *buffer insertion* for static routing topologies, (2) *Performance driven routing* (i.e., topology synthesis) and (3) *Logic replication.*

The proposed algorithms have been designed for realistic and practical problem formulations and delay models. For instance, the proposed techniques adopt a *"minimum cost subject to timing constraints"* formulation. While such a formulation is clearly valuable (perhaps imperative) from an engineering point of view, it has been frequently overlooked in the physical design literature, limiting the practicality of much previous work. Further, the proposed techniques for static topology optimization and topology synthesis explicitly capture a *complete* cost/performance tradeoff and, in fact, compute a *suite* of solutions which exhibit this tradeoff. One of the tenets of this thesis is that hte flexibility provided by such an approach is of significant practical utility.

Chapter 1

Introduction

1.1 Technology Trends and Background

Recent trends in fabrication technology for *Very Large Scale Integrated* (VLSI) circuits have resulted interconnect delay being a significant bottleneck in system performance. As device geometries enter the deep submicron range ($0.5\mu m$ and below), transistor switching speeds become proportionally faster, but wire resistance increases significantly. Consider the technology parameters in Table 3.1 (data taken from [2]; 2.0 and 1.2 micron data is from MOSIS and 0.5 micron data is from MCNC) [1] Of particular interest is the so-called *resistance ratio*; this figure represents the length of a wire with resistance equal to the output resistance of the driver. As we scale down into the submicron range, a wire which travels a relatively small distance with respect to the die size (say, $1cm^2$) to equal the driver resistance. This trend becomes more dramatic when one considers that more advanced fabrication technology invariably results in larger feasible die size. Additionally, technologies below 0.25 microns are due to appear in the near future.

As a result of this trend toward interconnect critical systems, timing opti-

[1] Note that this data is merely intended to be representative, using "typical" drivers etc.. Compared to some other technology data, the unit capacitance values in the table are somewhat small with values up to one order of magnitude larger appearing in some, otherwise comparable, technologies. Nevertheless, the general technological trend still holds.

tech. (CMOS)	$2.0\mu m$	$1.2\mu m$	$0.5\mu m$
driver res.	164.0Ω	212.1Ω	270.0Ω
wire res.	$0.033\Omega/\mu m$	$0.073\Omega/\mu m$	$0.122\Omega/\mu m$
wire cap.	$0.234fF/\mu m$	$0.083fF/\mu m$	$0.039fF/\mu m$
sink cap.	$5.7fF$	$7.06fF$	$1.0fF$
res. ratio (cm)	0.5	0.29	0.24

Table 1.1: Representative Technology Parameters

mization is no longer solely in the domain of logic synthesis. On the contrary, *physical layout* now must play a central role in designing circuits and systems which deliver the performance promised by the underlying technology. This change in emphasis has been well-known since the late 1980's and a great deal of research has been performed as a result. Many of the traditional phases of physical layout have now become *multi-objective*. For example, in the case of circuit placement, in addition to the traditional objective of wireability optimization, it has become necessary to consider the performance of the circuit since register to register delays are so strongly affected by geometric distance – i.e., wiring delay. Accordingly, a number of researchers have attacked the *performance driven placement* problem (e.g., [24], [44], [11] and [43]). A similar situation holds in the case of routing. In the past, minimization of total routing wire length achieved multiple goals simultaneously; because delay was dominated by gate delay, minimizing the wiring load on a gate correlated very strongly with maximizing circuit performance. Thus, minimizing total wire length optimized power consumption, chip area and performance simultaneously. Unfortunately, we can no longer rely on this strong correlation between total wire length and performance and thus, various forms of the *performance driven routing* problem have recently received attention (e.g., [36], [3], [6] and [20]).

While the advent of deep submicron technology has changed the objectives of classical problems in physical layout, it has also introduced new physical design problems. For instance, the technique of buffer insertion is effective in optimizing both logic delay and wiring delay; its effectiveness in logic optimization is well studied and many logic synthesis tools perform automatic buffer insertion (typically under the name *fanout optimization* – e.g., [46], [42]). However, when interconnect delay is significant, it makes better engineering sense to approach buffer insertion as a physical layout problem; additionally, routability concerns favor a post-synthesis approach. By the same token, deep submicron technology has given rise to previously unstudied optimization techniques such as *wire sizing*; since wire resistance has become significant, it is now often the case that performance can be improved by selectively *widening* segments of a routing topology.

1.2 Contributions

In light of the clear need for timing optimization in physical design, this thesis focuses on several central problems in this area. Algorithmic techniques for simultaneous wire sizing and buffer insertion, performance driven routing and the largely unexplored optimization technique of logic replication are presented.

In all cases practical problem formulations and realistic delay models are adopted. Some of these decisions are highlighted below.

- Adoption of the intuitively appealing notion of *required arrival time*. When optimizing a net, the criticality of a particular sink is determined by its required arrival time (which in turn is determined by the critical path from that sink to a primary output).

- Careful attention to cost/performance tradeoffs. It is frequently the case that the maximum performance solution for a particular optimization technique is not a good engineering choice – e.g., such a solution may have intolerably overhead in terms of a cost metric such as power consumption or area. In the cases of wire sizing and buffer insertion, and performance driven routing, the proposed techniques also compute a *suite* of solutions which exhibit this cost/performance tradeoff.

- Adoption of reasonably accurate delay models. This includes incorporation of a generalized buffer delay model which takes into account signal transition time. The *Elmore* delay model is adopted in performance driven routing.

These decisions contrast with some previous approaches. For instance several pervious approaches have studied objective functions which are *weighted sums* of source to sink delays (where a sink weight is intended to capture the "criticality" of that sink). It is our contention that such formulations are heuristic and a direct approach based on required arrival times is preferred. Similarly, several previous works in performance driven routing have focused either on a *linear* delay model (where delay is assumed to be proportional to wire length and does not consider non-linear loading effects) or have targeted *maximum performance* solutions without regard to cost. This thesis proposes that these are serious practical deficiencies and have avoided such pitfalls in our work. Consider the delay/routing area tradeoff curve in Figure 1.1. The figure shows is the tradeoff curve for a 12 pin Multi Chip Module net computed by the techniques in Chapter 3 for performance driven routing with simultaneous wire sizing. For simplicity, the timing metric is maximum source to sink delay – i.e., all sinks have identical required times. As is typical of performance/cost tradeoffs, the curve exhibits a severe "flattening" as the maximum performance solution is approached; thus, since wiring area affects power consumption, routability and chip area, the logical engineering decision is to select a solution near the "elbow" of the curve where significant cost savings are achieved versus the minimum delay solution with a very small (perhaps negligible) reduction in performance – indeed, in this case a likely practical solution may reduce wiring area by almost half versus the maximum performance solution.

The remainder of this thesis is organized in four chapters. Chapter 2 presents algorithms for wire sizing and buffer insertion. These algorithms are devised for *static* (i.e., given) topologies. Attention is given to efficient implementation and a generalized buffer delay model is incorporated by the manipulation of piece-wise linear functions. The algorithms are able to efficiently identify *optimal* solutions. Chap-

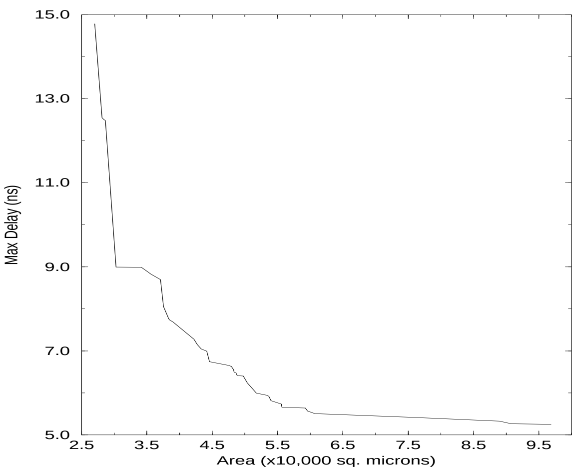


Figure 1.1: Area vs. Delay curve for a 12 pin MCM net

ter 3 studies a new performance driven routing technique – i.e., how to synthesize the routing topology itself. This is formulated as a generalization of the classical *Rectilinear Steiner Tree* problem. The proposed approach is able to incorporate simultaneous wire sizing and has produced state of the art routing solutions. Since the classical Rectilinear Steiner Tree problem is NP-hard, the approach does not assure optimality. However, rather than devising ad-hoc heuristics, the algorithm optimally solves a constrained variant of the problem in which the topology must be induced by a permutation on the sinks of the net; this allows efficient identification of optimal solutions in a rich and flexible solution space. The approach has been dubbed the *P-Tree* algorithm to reflect this permutation constraint. A novel technique for finding high quality sink permutations is also presented. Chapter 4 presents theoretical foundations for a complementary optimization technique: logic replication. The problem is characterized at both the logic level and the layout level and algorithms are proposed for each formulation. At the logic level an optimal solution to the two-way replication problem by applying a solution to the subset sum problem. At the layout level, the permutation constraint devised in P-Tree algorithm is adopted. Under this constraint, an algorithm for optimal multi-way replication is devised. Finally, we conclude in Chapter 5.

Chapter 2

Optimizing Static Topologies by Wire Sizing and Buffer Insertion

This chapter presents algorithms for optimal wire sizing and buffer insertion for a given routing topology. The *Elmore* delay model is adopted to approximate wiring delay. Buffer delay is first modeled by a standard RC model; the techniques are then generalized to accommodate a more accurate buffer delay model which considers the effect of signal transition time or *slew*.

The proposed algorithms have either polynomial or pseudo-polynomial complexity; their efficiency has been validated experimentally.

2.1 Background and Preliminaries

2.1.1 Notational Conventions

Throughout this chapter, the following notational conventions are adopted.

T_v	A routing tree rooted at node v
$l(v)$, $r(v)$	The left and right children of node v respectively
e_v	Tree edge (wire) from node v to its parent
l_e	Length of edge e
c_e	Capacitance of edge e
c_v	Input capacitance of sink v
r_e	Resistance of edge e
c_b	Input capacitance of buffer b
r_b, r_g	Output resistance of buffer b or gate g
d_b, d_g	Intrinsic delay of buffer b or gate g
p	Polarity; usually referring to a signal, $p = 1$ meaning inverted
p_b	Buffer type; $p_b = 1$ indicates b is an inverter; $p_b = 0$ o.w.
q_v	Required arrival time of sink node v
W	Largest possible wire width ($1..W$ are possible)
B	Buffer library
leaves(T)	Set of leaves of tree T

2.1.2 Delay Models

For a given routing tree possibly containing buffers, delay along a root-sink path is made of (1) delay along wires and (2) delay through buffers and the driving gate. This work adopts the Elmore [13] delay model for wires and considers two buffer delay models: a common RC model and an augmented model which considers signal slew.

The capacitance c_e and resistance r_e of wire segment e having width w_e are given as:

$$c_e = \alpha l_e \cdot w_e \qquad\qquad r_e = \beta l_e / w_e$$

where α and β are characteristic constants of the process technology.[1] From these relationships, the tradeoff associated with widening a wire are clear; wider wires can be faster because of reduced resistance, but present greater load upstream in the signal path.

To compute the Elmore delay of a wire e_v in tree T, we first recursively

[1]While our algorithms are presented with this model, we note that it is not essential – e.g., such phenomenon as fringe capacitance can be taken into account.

define $c(T_v)$, the total lumped capacitance of T_v, as follows.

$$c(T_v) = \begin{cases} c_v & \text{if } v \text{ is a sink node} \\ c_b & \text{else if buffer } b \text{ placed at } v \\ c(T_{l(v)}) + c(T_{r(v)}) + c_{e_{l(v)}} + c_{e_{r(v)}} & \text{otherwise} \end{cases} \tag{2.1}$$

Intuitively, $c(T_v)$ is simply the capacitive load seen at v – i.e., the sum of the loads of the left and right subtrees, $c(T_{l(v)})$, and $c(T_{r(v)})$, and the capacitance of the wires to those sub-trees, $c_{e_{l(v)}}$ and $c_{e_{r(v)}}$. Given this notation, the Elmore delay of wire e_v, is defined as

$$\text{elmore}(e_v) = r_{e_v}(\frac{c_{e_v}}{2} + c(T_v)).$$

The Elmore model is a very good choice for optimization procedures. While it certainly isn't as accurate as simulation based methods such as Spice, it provides an excellent balance between ease of computation and accuracy. In fact it has been said to be the most accurate model which is an algebraic combination of the R's and C's of a circuit model [35]. Further, the model has been shown to have very good *fidelity* [2]; while the Elmore delay exhibits significant error versus Spice computed delay, an optimal solution under the Elmore model is typically very close to an optimal solution under Spice computed delay. Two other features of the Elmore model make it an appealing optimization metric. First, it provide an absolute upper bound on the 50% delay for RC circuits [16] (i.e., although it exhibits error, it is one-sided error). Second, it has certain "separability" properties (i.e., the delay associated with a subtree can be computed independently of the rest of the routing tree) which are useful in the design of dynamic programming algorithms for timing optimization.

For completeness, it should be noted that Rubinstein, Penfield and Horowitz [37] devised a *sink independent* upper bound model (RPH) based on Elmore delay. Letting $r_{d,v}$ be the total path resistance from the driver (including the driver), the RPH bound is given as

$$\sum_{v \in T} r_{d,v}(c(T_v) + c_{e_v}).$$

The RPH formula clearly is an upper bound on the Elmore delay to any sink in the tree, however, it does not differentiate among those sinks.

The delay through a buffer b at node v in a basic model is determined by the load on the buffer's output $c(T_v)$, b's *intrinsic* (load independent) *delay* d_b and *output resistance* r_b. The delay through buffer b with load c_l on its output is

$$\text{buf\_delay}(b, c_l) = d_b + r_b c_l.$$

The key to buffer insertion in optimizing delay is the *isolation property* of buffers exhibited in Equation 2.1. Namely, the capacitance of a subtree rooted at a buffer, as seen by ancestors in the tree, is determined entirely by the input capacitance of that buffer. In other words, the buffer *decouples* the capacitance of its descendants from its ancestors.

A common generalization of this basic buffer delay model includes an additive term to account for the *slew* of the signal entering the buffer. One model for this delay is the product of a buffer dependent constant λ_b and the load delay of the previous stage, D_{Lprev} – i.e. the RC delay of the driving buffer. Thus we denote the augmented delay equation as

$$\text{buf\_delay}_{\text{slew}}(b, c_l) = \text{buf\_delay}(b, c_l) + \lambda_b D_{Lprev}. \tag{2.2}$$

This and similar models have been proposed in various contexts (e.g., [45],[27]). An extension of our algorithms to accommodate this delay model is discussed in Section 2.6.

2.1.3 Previous Work in Wire Sizing

Early work in wire sizing includes the work of Cong, Leung, Zhou and Koh who provided several studies of wire sizing in [6, 7, 8] and demonstrated its potential in improving delay. The first of these works [6] attacked the problem of minimizing delay under the Rubinstein, Penfield, Horowitz sink oblivious model discussed in Section 2.1.2 of this chapter. To address the shortcomings of the RPH model, [7] later addressed the problem of minimizing the *weighted sum* of the source-to-sink *Elmore* delays for a set of identified critical sinks in a given routing tree; however, the weighting coefficients are presumably provided by the user. Under this formulation they prove several properties which lead to an $O(n^r)$ algorithm for a net with n segments each having r possible widths. The authors also propose a greedy heuristic procedure with run time of $O(n^3 r)$. In [8], Cong et. al. also attacked the problem of incorporating a cost function such as area or power. Their formulation is, again, a weighted sum of their stated timing objective function and the cost function.

In [40], Sapatnekar studied the more common metric of *maximum source-to-sink delay* – or, more generally, the task of minimizing cost subject to given timing constraints (under the Elmore model). He noted that the property of *separability* used by Cong and Leung in designing their algorithm did not hold for this case. In addition, the property of *monotonicity* utilized in [7] does not apply when the length of all wire segments is not identical. In the same paper Sapatnekar proposed a convex programming formulation of the maximum delay, *continuous* wire-sizing problem followed by a sensitivity based mapping heuristic to discretize the solution.

Later, in [30], a dynamic programming algorithm which exploited the fact that the lengths of wire segments are *discrete* in nature (i.e., that they are integer multiples of a basic grid length) was given. This led to the observation that, over all possible width assignments to a subtree, the number of *distinct* capacitive values at the root is polynomially bounded. This yielded a polynomial time minimum delay wire sizing dynamic programming algorithm. However, power considerations were not explicitly taken into account.

2.1.4 Previous Work in Buffer Insertion

Research on buffer insertion includes the early works of Berman et. al., [1], Touati [46], van Ginneken [48] and Dhar [10]. Other contributions in this area include [32], [42], [30], [27], and [47]. Much previous work (e.g., [46], [42], [1]) approach buffer insertion as a logic synthesis problem; in other words, buffer trees are imposed on the network prior to physical layout. There are important engineering considerations associated with such an approach. Perhaps most important among these considerations are the difficulty of accurately taking into account the resistive and capacitive effects of interconnect, as observed in [27], and potential routability problems created by pre-layout buffer trees as mentioned in [1].

As a result of these practical considerations, the work presented here focuses on a post-layout methodology where topological information is available. Previous work on post-layout buffer insertion includes [48], in which van Ginneken gave an elegant polynomial time algorithm for delay-optimal buffer insertion into a given topology. He extended his algorithm to minimize the number of buffers subject to given timing constraints. He noted that this extension was not, in general, polynomial, but that efficient run-time was observed in practice. Implementation details of this extension were not given. His algorithm did not consider the effect of signal slew on buffer delay. It should be pointed out that earlier, in his Ph.D. thesis [10], Dhar presented a very similar approach.

Other post-layout techniques include [27] and [30]. The heuristic method of [27] is also notable in that it incorporated signal slew into the delay model. In [30], an algorithm minimizing maximum source to sink delay (or maximizing required arrival time) for a given routing topology by simultaneous buffer insertion and wire sizing was given. However, neither power (or area) considerations or signal slew were taken into account.

2.1.5 Contributions

We present efficient algorithms for wire sizing, buffer insertion and both techniques simultaneously. This work builds on the work of van Ginneken [48] and Dhar [10] as well as our preliminary work in this area [30]. Our main contributions are summarized as follows.

- Polynomial-time algorithms for the optimal "min power subject to timing constraints" wire sizing and buffer insertion problems (either separately or simultaneously). This includes computation of the entire power versus performance curve and an efficient data-structure for efficiently pruning sub-optimal solutions.

- Incorporation of signal slew into the buffer delay model by manipulation of piece-wise linear functions.

In this work, timing constraints are given explicitly as required arrival times at the sinks of the net rather than as coefficients of a weighted sum of the sink delays. We suggest that computation of the entire power-delay trade off curve is of practical significance as it provides added flexibility to the designer.

The incorporation of signal slew is also significant since its contribution to total delay can be over 50% (see e.g. [18]) and therefore cannot be neglected in practice.

The ability to use inverters as buffers rather than resorting to pairs of inverters to ensure proper signal polarity is also of practical utility.

The complexity of our algorithms without signal slew taken into account is summarized in the following table.

	Max Req-Time	Min Power
wire-sizing	$O(nmW^2)$	$O(nmW^2)$
buffer-insertion	$O(n^2\|B\|^2)$	$O(\|B\|n^3 c_{\max}^2 \log(nc_{\max}))$
both	$n^2 c_{\max} \max(\|B\|, W)$	$O((W + \|B\|)n^3 c_{\max}^2 \log(nc_{\max}))$

In the table, n is the number of sinks in the net, m is the number of basic (unit) grid segments in the routing tree (derived from the underlying routing grid), and $c_{\max}$ is the largest possible capacitive value of any component in the tree. Set B is the given buffer library and W is the largest multiple of the basic wire width allowed. Where $O(c_{\max}^2)$ is a component of the complexity, we assume the capacitive parameters of the problem are given as or translated into *polynomially-bounded integers*. As such, these algorithms have pseudo-polynomial complexity [14]. However, in these cases, the bounds are very pessimistic versus observed behavior.

When signal slew is incorporated into the delay model, we are not able to give polynomial bounds due to degenerate situations. However, in practice we observe these algorithms to perform similarly to their simpler counterparts – typically a constant factor slower.

To the best of our knowledge, this work represents the most efficient algorithms to date for these problems and the first work on buffer insertion to incorporate signal slew into the delay model while guaranteeing optimality. We also improve on the results of [30] in terms of run-time when minimizing maximum source to sink delay (or maximizing required time) independent of cost is the goal.

The proposed algorithms adopt a bottom-up dynamic programming approach. Rather than computing a single solution for each subtree, we compute a set of solutions where each member of the set is characterized by both the timing properties and capacitance of the associated solution. Solution sets are kept small by employing an observation made by van Ginneken [48] and Dhar [10] which essentially says that when combining the solution sets of a node's left and right children to create a new solution set, the new set need not consider all pairs of left and right solutions; rather only a *linear* number of pairs need be considered since one branch will always dominate. In addition, when minimizing power, we employ a similar observation to identify inherently sub-optimal solutions and thereby drastically reduce the size of solution sets. This property is applied efficiently by use of an augmented tree data-structure. Generalizations of these techniques are developed to handle the case where slew is taken into account.

2.2 Delay Models and Problem Formulations

2.2.1 Maximum Required Time Formulation

We adopt maximization of required arrival time at the root of the net as our timing metric. The required arrival time at node v, $q(T_v)$, is the latest time at which the input(s) of v must be available for the required arrival times of all sinks in T_v to be met. This measure naturally captures the typical situation of optimizing a net in a combinational circuit – i.e., the required-times are determined by the rest of the circuit. Formally, $q(T_v)$ is defined as:

$$q(T_v) = \min_{u \in \text{leaves}(T_v)} (q_u - \text{delay}(v, u)).$$

If the required arrival time, $q(T)$ at the root is non-negative, the tree T is said to meet its timing requirements. A solution which maximizes $q(T)$ is a maximum performance solution; finding such a solution (irrespective of cost) is the topic of Section 2.4. Note that the required time formulation is a generalization of the maximum delay formulation – i.e., if $q_v = 0$ for each sink v then *max delay* $= -q(T)$.

2.2.2 Minimizing Power Subject to Timing Requirements

A more practical formulation considers the *cost* of the solution – i.e., we may want to minimize cost subject to the given timing constraints. We also study this formulation when our cost objective is dynamic power dissipation.

The dynamic power dissipation for CMOS technology P_d is given in [50] as

$$P_d = C_L V_{DD}^2 f_p$$

where C_L is load capacitance and f_p is the switching frequency. Thus, with respect to buffer insertion and wire sizing, total capacitance is the correct measure of dynamic power dissipation since f_p and V_{DD} are unaffected by these methods.

Letting C_{total} be the total capacitance associated with a buffered and sized routing tree we have the following problem.

$$\begin{array}{ll} \text{minimize} & C_{\text{total}} \\ \text{subject to} & q(T) \geq 0 \end{array}$$

Alternatively, an attractive approach to the problem is to provide the designer with a power-delay tradeoff curve from which the desired solution may be chosen.

Implicitly we have assumed that dynamic power dissipation dominates short-circuit power dissipation. We justify this by the presumption that design techniques have been employed to eliminate or drastically reduce short-circuit power dissipation.

2.3 Algorithmic Framework

We first give the framework for a high level dynamic programming algorithm into which all subsequent algorithms will fit. The framework covers all variants of interest. For instance, if we are only interested in wire sizing, we simply run the algorithm with an empty buffer library B. The specific algorithms will differ in their implementation of the basic routines called by the general algorithm and the characteristics of the solution sets they compute.

This General Dynamic Programming algorithm *GDP()* is given as pseudo-code in Figure 2.1. The algorithm computes the solution sets $S_b(v)$ and $S_t(v)$. The set

```
Algorithm GDP(T,B,W)
  Foreach node v ∈ T in topological order from leaves to root
    If v is a leaf
      Compute  S_b(v) ←Base_Case(v)
    Else
      Compute  S_b(v) ← Bottom_Solutions(v, S_t(l(v)), S_t(r(v)))
    If v is not the root
      Compute  S_t(v) ← Top_Solutions(v, S_b(v))
    Else /* v is the root */
      Compute  Optimal_Soln(v, S_b(v))
```

Figure 2.1: General algorithm structure

$S_b(v)$ can be thought of as the set of solutions for subtree T_v, including the possibility of inserting a buffer at v. Similarly, the set $S_t(v)$ can be thought of as the set of solutions for T_v *augmented* by the wire from its parent e_v and including possible sizing of e_v.

The four procedures *Base_Case()*, *Bottom_Solutions()*, *Top_Solutions()*, *Optimal_Soln()* are routines which inductively compute solution sets from the solution sets of descendants. For each particular algorithm, these sets are parameterized differently. Intuitively, these routines can be thought of as follows.

Base_Case()	Compute the singleton set giving relevant parameters at sink v
Bottom_Solutions()	Given solution sets for left and right children, compute the solution set *at* v including the possibility of inserting a buffer at v
Top_Solutions()	Given the solution set *at* v, construct the solution set for T_v augmented by wire e_v
Optimal_Soln()	Given the solution set at the root, select the best solution when combined with the driver

The key to solving a particular problem formulation (e.g. minimizing power subject to timing constraints), is in the content of the solution sets, their efficient computation by the above routines and limiting their size as much as possible to ensure computational efficiency.

In the next section we study the simplest scenario: maximizing the required arrival time $q(T)$ at the root of the tree under the basic RC delay model. Subsequently we generalize this to the problem of minimizing power dissipation subject to given timing constraints (again under the basic RC model). Finally, we further generalize the algorithm to solve either of the preceding problems with the generalized buffer delay model taking into account the effect of signal slew. In each of these cases, the algorithms are sketched based on the framework of this section.

2.4 Maximizing Required Arrival Time

In this case we maximize $q(T)$, the required arrival time at the root of T, under the basic RC delay model. The solution sets $S_b(v)$ and $S_t(v)$ are each partitioned into two disjoint subsets as

$$S_b(v) = S_b^+(v) \cup S_b^-(v)$$
$$S_t(v) = S_t^+(v) \cup S_t^-(v).$$

The reason for this partitioning is to deal appropriately with the fact that, since we may be using inverters as buffers, the signal may be inverted for some portions of the net. The sets with superscript "+" contain solutions where we assume the incoming signal is *non-inverted* and the sets with superscript "-" contain solutions where the incoming signal *is* assumed to be inverted. In other words, the superscript "+" ("-") ensure that each path to a sink passes an *even* (*odd*) number of inverters.

The solutions themselves are *load, required-time*, or (c, q) pairs. Intuitively, the English meaning of these sets is, for example,

$(c, q) \in S_b^+(v) \Leftrightarrow$ "There exists an assignment to T_v with upward load c and required time q at v when the incoming signal is not inverted."

An important initial observation made by van Ginneken [48] and Dhar [10] is the following.

Property 2.4.1 *For* $(c, q), (c', q') \in S$, *if* $c' \geq c$ *and* $q' < q$ *then* (c', q) *is sub-optimal.*

This is clear since a larger load can only worsen delay of ancestor components. In other words, we always prefer smaller load and larger required time. Supposing that these sets are arranged in increasing order of load leads to the following property.

Property 2.4.2 *Any load-required_time set* S *in increasing order of load, may be replaced by* $S' \subseteq S$ *where* S' *is strictly increasing in required time.*

We maintain this sorted order as an invariant so that we may easily exploit this property.

In the context of our algorithmic framework, we fully specify the algorithm as follows. Recall that c_v and q_v are the input capacitance and required arrival time of sink v respectively. The routine *Base_Case*(v) simply sets $S_b^+(v) \leftarrow (c_v, q_v)$ and $S_b^-(v) \leftarrow \emptyset$ since no valid solution will have an inverted signal at a sink.

Computation of *Bottom_Solutions*() is described in pseudo-code in Figure 2.2. The algorithm first computes the optimal (c, q) pairs for unbuffered solutions in lines 2-13. For each achievable arrival time q, we find the smallest load achieving q. This is done in a manner similar to the merging of two sorted lists and ensures that Property 2.4.2 holds. This is a linear time operation and the size of the resulting set is linear as observed by van Ginneken [48]. Next we find the optimal buffer

Algorithm: *Bottom_Solutions*$(v, S_t(l(v)), S_t(r(v)))$

1. /* First compute unbuffered solutions */
2. $S_b^+(v) \leftarrow \emptyset$
3. **Let** $S_l = S_t^+(l(v))$
4. **Let** $S_r = S_t^+(r(v))$
5. /* S_l, S_r are indexed and ordered by c */
6. $i \leftarrow 1$; $j \leftarrow 1$
7. **While** $(i \leq |S_l|$ and $j \leq |S_r|)$
8.   **Let** $(c_l, q_l) = S_l[i]$
9.   **Let** $(c_r, q_r) = S_r[j]$
10. $S_b^+(v) \leftarrow S_b^+(v) \cup \{(c_l + c_r, \min(q_l, q_r))\}$
11.   **If** $(q_l \leq q_r)$ /* Left Critical */
        $i \leftarrow i + 1$
12.   **If** $(q_r \leq q_l)$ /* Right Critical */
        $j \leftarrow j + 1$
13. **Compute** $S_b^-(v)$ analogously
14. /* Now compute buffered solutions */
15. **Foreach** buffer $b \in B$
16.   **If** b is an inverter
17.   **Find** $(c, q) \in S_b^-(v)$ s.t.
          $q_b^+ = q - d_b - r_b c$ is maximized
18.   **Else**
19.   **Find** $(c, q) \in S_b^+(v)$ s.t.
          $q_b^+ = (q - d_b - r_b c)$ is maximized
20.   Analogously compute q_b^-
21. $S_b^+(v) \leftarrow S_b^+(v) \cup \{(c_b, q_b^+) | \forall b \in B\}$
22. $S_b^-(v) \leftarrow S_b^-(v) \cup \{(c_b, q_b^-) | \forall b \in B\}$
23. Prune $S_b^+(v)$ and $S_b^-(v)$ by Property 2.4.2

Figure 2.2: *Bottom_Solutions()* routine for max required time

configurations in lines 15-20 by pairing buffers $b \in B$ with unbuffered solutions at v. We then perform merging and additional pruning in lines 21-23. Also note that the final pruning step is also linear since the sets are in sorted order.

We give the implementation of *Top_Solutions()* in Figure 2.3. We examine

Algorithm: *Top_Solutions*$(v, S_b(v))$
1. $S_t^+(v) \leftarrow \emptyset$ 2. **Foreach** $(c,q) \in S_b^+(v), w \in \{1..W\}$ in increasing order of $c' = c + \alpha w l_{e_v}$ 3. $S_t^+(v) \leftarrow S_t^+(v) \cup \{(c', q - \text{elmore}(e_v))\}$ /* elmore delay evaluated at width w */ 4. **Compute** $S_t^-(v)$ analogously 5. Prune $S_t^+(v)$ and $S_t^-(v)$ by Property 2.4.2

Figure 2.3: *Top_Solutions()* routine for max required time

all pairings of widths w for wire e_v (having length l_{e_v}) with solutions $(c,q) \in S_b(v)$ at v. Since the loads c' are of the form $c + wZ$ where Z is fixed and $w \in \{1..W\}$, we can visit all c''s in order without explicitly sorting them. For each such pairing, we obtain a new required time. In a final sweep we again apply Property 2.4.2 to ensure that the set is in strictly increasing order of both c and q.

Finally, we must implement *Optimal_Soln*$(v, S_b(v))$. This is done simply by pairing all of the previously computed solutions $(c,q) \in S_b^+(v))$ (since the signal leaving the driver is non-inverted) with the properties (i.e., output resistance) of the driver and selecting the solution with the largest resulting required arrival time. Note that if we are not allowed to buffer the driver output, we can simply include only un-buffered solutions in $S_b^+(v)$.

To construct the solution achieving this timing we recursively re-visit the tree to determine which choices of buffering and wire sizing led to the optimal solution. This is accomplished by storing with each (c,q) pair, local information indicating the choices which led to that solution.

2.4.1 Comments

For simplicity, we have presented the algorithm in terms of a binary tree, but note that the algorithm is easily applied to general trees. One straight-forward method to achieve this is to convert a non-binary tree to an equivalent binary tree simply by adding zero-length wires. For instance, suppose we have a node v with fanout 3 to nodes A, B and C. We replace v with two nodes v' and v'' where node v' will have children A and B, and v'' will have children C and v' with the wire from v'' to v' having length 0. The algorithm can be modified to prohibit the placement of buffers at particular nodes – v' in this case.

Another issue is that, as described, the algorithm assumes exactly one sizeable wire segment between nodes and buffer insertion only at nodes in the tree. However, the algorithm is easily implemented to accommodate multiple sizeable segments in a single wire and and buffer insertion within a wire by introduction of intermediate nodes.

We further note that the algorithm can easily be extended to allow for optimal sizing of the driving gate. However, it should be realized that such sizing may have global effects by altering the input capacitance of the driver thereby affecting the timing requirements and the system as a whole.

These comments apply to all subsequent algorithms in this chapter.

2.4.2 Run Time

We analyze the running time of the basic algorithm in three scenarios:

(1) $|B| = 0$, $W > 1$ (Wire sizing alone).

(2) $|B| \geq 1$, $W = 1$ (Buffer insertion alone).

(3) $|B| \geq 1$, $W > 1$ (Both methods).

In scenario 1 (wire sizing alone), we introduce the notion of "basic segment-length" to analyze the complexity. This parameter is derived from an underlying routing grid.

Property 2.4.3 *In scenario* 1 *the size of each load-required_time set* S *is bounded by* mW *where* m *is the total number of basic grid lengths in the tree.*

This can be seen by considering that the load at node v can be expressed as

$$c(T_v) = \text{Sink\_Load} + \text{Interconnect\_Load}$$

where *Sink_Load* is fixed and *Interconnect_Load* $= \gamma \sum w_i$ where $w_i \in \{1..W\}$ is the width of the ith wire and γ is a constant derived from α, the basic grid length and the minimum width. Thus, the load is entirely determined by $\sum w_i$ which can only take on integer values in range $\{m..mW\}$. This gives an upper-bound of mW on the size of individual solution sets the algorithm computes since it bounds the number of *distinct* load values. Thus, even though there are an exponential number of width assignments, there are only a polynomial number of distinct resulting loads. The resulting run-time is $O(nW(mW)) = O(nmW^2)$. In the case where every sizeable segment is of identical length, we have $O(n^2W^2)$ since, effectively, $n = m$ in this case.

Scenario 2 is a generalization of the situation for the algorithm of van Ginneken [48]. (It is a generalization only in the sense that it allows multiple buffers in its library). Since $W = 1$ in this case, computation of S_t sets is trivial. Thus the size of the S_b sets is the key factor in the run-time. We first state the following properties alluded to earlier.

Property 2.4.4 *For* $S_b(v)$*, let* S_l *and* S_r *be the* S_t *solution sets of* v*'s left and right children respectively of the same polarity. The following inequality holds:* $|S_b(v)| \leq |S_l| + |S_r| + |B|$.

Property 2.4.5 *In scenario* 2*, for all load-required_time sets* S*,* $|S| \leq n + n|B|$.

These properties coupled with the fact that the merging operation is linear in $|S_l|+|S_r|$ gives an overall worst-case complexity of $O(n|B|(n + n|B|)) = O(n^2|B|^2)$.

Scenario 3 is complicated by the fact that the input capacitance of the buffers may not be simple multiples of the capacitance of a unit-length wire. However, in practice it is reasonable to assume that capacitive values can be linearly mapped onto a polynomially-bounded integer domain with sufficient precision or are given as such. In such a situation we introduce another value $c_{\max}$, which is the largest capacitance possible among the individual components of the tree (e.g., it may be the capacitance of the longest wire at the maximum width). Under this formulation we upper bound the size of the solution sets by $nc_{\max}$ and the overall run-time by $O(n^2 c_{\max}(\max(W, |B|)))$. In practice, observed run-times are typically much less than this bound.

2.5 Minimizing Power Subject to Given Timing Constraints

We now extend the algorithm to minimize dynamic power dissipation subject to given timing constraints. For clarity, we present this and subsequent algorithms without regard to signal polarity; application of the ideas of the previous section to deal with the presence of inverters is straight-forward. We note also that the extension presented in this section can easily be modified by minimize area subject to timing constraints.

The first issue is how to parameterize the solution sets. We are not only concerned with the load and required-time of a sub-solution, but also the power it consumes. Thus, a sub-solution is now characterized by the parameters p (power), c (capacitance) and q (required-time).

We organize solution sets by first partitioning them into subsets with identical power values p. In other words, solution sets $S_b(v)$ and $S_t(v)$ now contain pairs (p, S_p) where p is power consumption (as a capacitive value), and S_p is an ordered set of load, required time pairs (c, q) as in the previous algorithm. For example, $(p, S) \in P_b(v)$ indicates that for power p and every $(c, q) \in S$, there exists an assignment for T_v consuming power p, presenting load c upward and yielding required time q at v.

We organize these sets first by sorting them in increasing order of power. Each set S_p is ordered by load c as in the basic algorithm.

One might think that the sets S_p are typically singleton sets, however, this is not the case. Because many different configurations may consume precisely the same power (by, for example, assigning an identical set of buffers to different locations), these sets can be quite dense.

Recalling that total capacitance is the correct measure for dynamic power dissipation, and thus, c_v is the "power" associated with sink v, *Base_Case*(v) simply sets $S_b(v) = \{(c_v, \{(c_v, q_v)\})\}$.

Pseudo-code for *Bottom_Solutions*() is given in Figure 2.4. We visit all possi-

Algorithm: *Bottom_Solutions*$(v, S_t(l(v)), S_t(r(v)))$

1. **Let** $B' = B \cup \{\phi\}$
   /* ϕ indicates "no buffer", $c_\phi = 0$ */
2. $S_b(v) \leftarrow \emptyset$
3. **Foreach** triple $(p_l, S_{p_l}) \in S_t(l(v))$, $(p_r, S_{p_r}) \in S_t(r(v))$, $b \in B'$
   in increasing order of $p = p_l + p_r + c_b$
4.   Combine S_{p_l}, S_{p_r} as in lines 7-12 of Figure 2.2 to give S'
5.   **If** $(b \neq \phi)$
6.   Find $(c, q) \in S'$ s.t. $q' = q - \text{buf\_delay}(b, c)$ is maximized
7. $S' \leftarrow \{(c_b, q')\}$
8.   **Else**
9. $S' \leftarrow S' \setminus \{(c, q) \in S' | \exists (c', q') \in S_{p'}, p' < p, c' \leq c, q' \geq q\}$
         /* where $(p', S_{p'}) \in S_b(v)$ */
10.   **If** $(p, S_p) \in S_b(v)$ (previous triple gave same p)
11. $S_p \leftarrow S_p \cup S'$
12.   Prune S_p by property 2.4.2
13.   **Else**
14. $S_b(v) \leftarrow S_b(v) \cup \{(p, S')\}$

Figure 2.4: *Bottom_Solutions()* routine for low power

ble values of total power consumption at v. These values are from among the buffered and unbuffered configurations. The notion of a "non-buffer" ϕ is introduced to unify the notation. In this case, we explicitly sort the values $p = p_l + p_r + p_b$. However, we observe that the number of such *distinct* values p is often orders of magnitude less than the worst case (quadratic). Because of this observation we utilize a hash table to make an initial pass over all pairs to extract the *distinct* values which we then sort. This avoids an overly expensive sorting operation.

Top_Solutions() is implemented in a similar manner; its pseudo-code appears in Figure 2.5.

Algorithm: $\mathit{Top\_Solutions}(v, S_b(v))$	
1.	$S_t(v) = \{(p, \emptyset) \mid p$ is a possible power$\}$
2.	**Foreach** pair $w \in 1..W$, $(p_b, S_{p_b}) \in S_b$ in increasing order of $p = p_b + \alpha w l_{e_v}$
3.	$S_p \leftarrow \emptyset$
4.	**Foreach** $(c, q) \in S_{p_b}$
5.	$S_p \leftarrow S_p \cup \{(c + \alpha w l_{e_v}, q - \text{elmore}(e_v))\}$ /* elmore delay evaluated at width w */
6.	Prune S_p by Property 2.4.2
7.	$S_p \leftarrow S_p \setminus \{(c, q) \in S_p \mid \exists (c', q') \in S_{p'}, p' < p, c' \leq c, q' \geq q\}$ /* where $(p', S_{p'}) \in S_t(v)$ */
8.	**If** $(S_p \neq \emptyset)$
9.	$S_t(v) \leftarrow S_t(v) \cup \{(p, S_p)\}$

Figure 2.5: *Top_Solutions()* routine for low power

As described, these algorithms implement two types of pruning. First we prune solutions $(c, q) \in S_p$ for a power p in the same way as before by Property 2.4.2. However, an additional pruning condition is utilized in Figures 2.4 and 2.5 on lines 9 and 7 respectively. This pruning condition is given by the following property.

Property 2.5.1 *Consider solution (c, q) consuming power p. If $\exists$ solution (c', q') consuming power p', where $p' < p$, $c' \leq c$ and $q' \geq q$ then the solution (c, q) is sub-optimal.*

The efficient application of this property has proven essential in giving reasonable running times in practice. Efficient detection of Property 2.5.1 is addressed in the next subsection.

We implement *Optimal_Soln*$(v, S_b(v))$ simply by selecting the lowest power un-buffered solution at the root giving required-time $q(T) \geq 0$ when paired with the driver. Alternatively, this set of un-buffered solutions gives the full tradeoff curve from which a solution may be selected.

2.5.1 Efficient Detection of Property 2.5.1

When constructing a solution set S_p consuming power p, we would like to efficiently determine if Property 2.5.1 holds for each $(c, q) \in S_p$. Since the solution sets can grow to be of substantial size, a linear scan of all other solutions to detect this property would likely be a disaster. We now outline a data-structure to efficiently support this operation.[2]

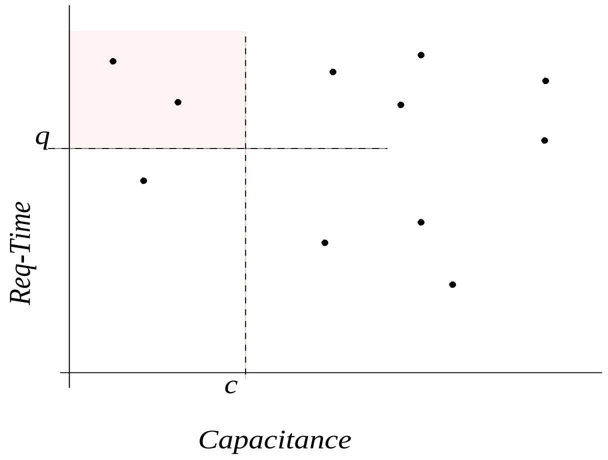


Figure 2.6: Geometric illustration of detection of Property 2.5.1

Recall that we visit all power values in order and therefore, we know that the entries in the data-structure are for all power values $p' < p$. Thus the data structure need only concern itself with c and q values – i.e., any solution which can eliminate a solution consuming power p has previously been constructed and stored in the data-structure; furthermore, all such previously visited solutions satisfy the $p' < p$ requirement by default.

We need a data structure which efficiently supports the following operations.

[2]The help provided by Matthew Clegg in devising this data-structure must be acknowledged.

- *insert*(c, q): update the data structure to reflect solution (c, q)
- *sub_opt*(c, q): returns TRUE if $\exists(c', q')$ previously inserted s.t. $c' \leq c$ and $q' \geq q$, FALSE otherwise.

The *sub_opt()* operation is illustrated in Figure 2.6. The data-structure must store points in two dimensional space and determine if a candidate point (c, q) is sub-optimal – i.e., if the grey region in the figure contains any points. Such a data-structure solves a special case of the *Orthogonal Range Query* problem from Computational Geometry (see e.g., [51]). Our problem is a special case for two reasons: first we need not retrieve or count all (c', q')'s satisfying the property and second, the subspace we are interested in is defined by two inequalities, $c' \leq c$ and $q' \geq q$, rather than four; in other words, our subspace is the intersection of the half-planes formed by two inequalities rather than a rectangle formed by four such inequalities. These special properties of our problem allow us to support the operations above in $O(\log m)$ time and $O(m)$ space for m entries by use of an augmented binary search tree. In contrast, the fastest known approaches to the general 2-dimensional orthogonal range query problem also run in $O(\log m)$ time but use $O(m \log m)$ space.

To support the operations we order a binary search tree tree by load values c. At each node t of the search tree we store the load value c *and* the *largest* q value among the entries in t's left sub-tree or *at* t itself. We refer to this value as $t.q_{l\_max}$. Given this data-structure, *insert()* is easily implemented recursively and *sub_opt()* can be implemented by examining the four following cases with respect to the given c, q and $t.c$ and $t.q_{l\_max}$ stored at the current node in the tree (boundary conditions are not given for simplicity):

$c < t.c, q < t.q_{l\_max}$	Explore left subtree
$c < t.c, q > t.q_{l\_max}$	Return FALSE
$c > t.c, q < t.q_{l\_max}$	Return TRUE
$c > t.c, q > t.q_{l\_max}$	Explore right subtree

By following these guidelines recursively down the search tree, we can detect the property in time proportional to the depth of the tree.

2.5.2 Run Time

With respect to wire-sizing alone, i.e. $|B| = 0$, we notice that $p = c$ for every candidate sub-solution since there is no decoupling by buffers. Thus the basic algorithm is sufficient to solve the low-power problem: we get power minimization "for free".

In the general case of simultaneous wire sizing and buffer insertion (or buffer insertion alone if $W = 1$), we have to take into account the quadratic nature of the merging steps of the algorithm. Since we examine all *pairs* of power values from the left and right children the solution sets are no longer assured to be linear in size. However, when the capacitive values are given as *polynomially-bounded integers* or can be mapped to such, once again we can show the run-time of the algorithm to be polynomial.

As in Section 2.4.2 let $c_{\max}$ be the largest possible capacitive value among the components. Under this scenario, we bound the number of load-required_time pairs at a node by $(nc_{\max})^2$. This gives an overall run time bound of $O(n(|B| + W)(nc_{\max})^2 \log(nc_{\max})) = O((|B| + W)(n^3 c_{\max}^2 \log(nc_{\max})))$. The log factor is an artifact of the sorting performed on the power values.

In practice, we observe much better run times as a result of the additional pruning described in the previous section (and not included in this analysis since we cannot prove it improves the worst case performance).

2.6 Accounting for Signal Slew

We now give a further generalization of the algorithm to account for the effect of signal slew on buffer delay. The key to our approach is the manipulation of piece-wise linear functions to model the effect of signal slew.

2.6.1 Overview

By Equation 2.2, buffer delay is augmented by the term $\lambda_b D_{Lprev}$ (recall that λ_b is a characteristic constant of buffer b and that D_{Lprev} is the RC delay of the previous stage). Since our algorithm proceeds in bottom-up order, this is an unknown value when computing the delay associated with a buffer. Conceptually, we would like to support queries of the form "What is the optimal solution at v with capacitance c *and* $D_{Lprev} = x$?"

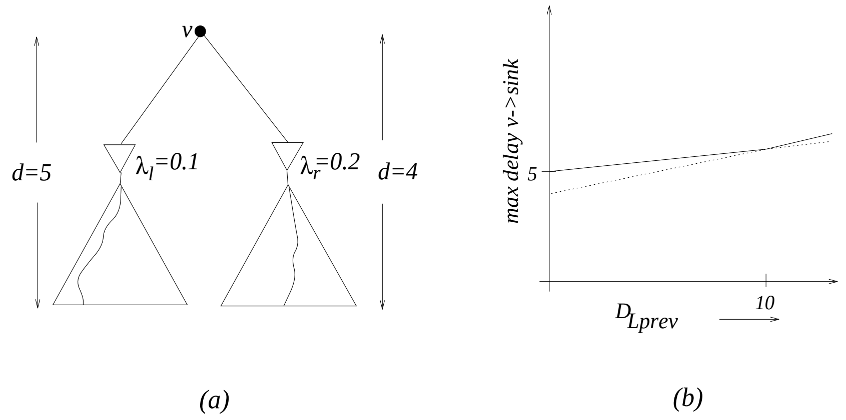


Figure 2.7: Piece-wise linear function modeling effect of signal slew

Since $\lambda_b D_{Lprev}$ is linear in D_{Lprev}, we utilize *piece-wise linear* functions to model this effect. Where we previously had *load-required_time* pairs (c, q), we now have *load-required_time_func* pairs (c, f) where f is a piece-wise linear function; $f(x) = q$ is the optimal required time q at v for load c and $D_{Lprev} = x$.

We illustrate the modeling of delay by a piece-wise linear function in Figure 2.7. Figure 2.7(b) shows the piece-wise linear delay function f at node v in Figure 2.7(a). The left and right subtrees have maximum delays of 5 and 4 units respectively when $D_{Lprev} = 0$. However, since the left and right subtrees are driven by different buffer types, they have different sensitivities λ_l and λ_r. The straight lines in Figure 2.7(b) equal to the two delay functions contributed by the two subtrees with slopes corresponding to the sensitivities λ_l and λ_r. The resulting delay function f at node v is shown as a solid line, which is the max of the two. Thus, different values of

D_{Lprev} can result in different critical paths.

We represent a piece-wise linear function f by a linked list of quadruples $(x_0, y_0, s, x_{\text{end}})$; each quadruple is a segment starting at point (x_0, y_0) ending at x_{end} and having slope s.

Our manipulation of piece-wise linear functions is based on three basic operations:

$$f = \text{pwl\_max}(f_1, f_2) \iff f(x) = \max(f_1(x), f_2(x)) \quad \forall x$$
$$f = \text{pwl\_min}(f_1, f_2) \iff f(x) = \min(f_1(x), f_2(x)) \quad \forall x$$
$$f' = \text{pwl\_add\_scalar}(f, d) \iff f'(x) = f(x) + d \quad \forall x$$

The first two of these operations can be performed in a manner similar to the merging of two sorted lists in linear time by stepping through the lists and examining points of intersection. Further, they can be generalized to operate on *sets* of functions rather than pairs by repeated application, giving for example, $\text{pwl\_max}(f_1, .., f_k)$. The third operation is achieved simply by adding d to the starting y-coordinate (y_0) of each segment in the function.

With respect to the dynamic programming algorithm, we must also associate with each segment in the piece-wise linear function f the relevant configuration information which yields the solution (e.g. wire-width, buffer type).

The algorithm modifications are summarized as follows:

(1) Where we had *load-required_time* pairs (c, q), we now have *load-required_time_func* pairs (c, f)

(2) Where we computed scalar *max* and *min* operations on arrival times q, we now compute *pwl_max* and *pwl_min* operations on piece-wise linear functions.

(3) Where we eliminated sub-optimal solutions (c, q) by Properties 2.4.2 and 2.5.1, we now eliminate sub-optimal *portions* of solutions (c, f) by analogous properties. These sub-optimal solutions now give required arrival time of $-\infty$ for certain values of D_{Lprev}.

A detailed description of the generalized algorithm follows. For generality, we focus on the low power formulation of the problem.

2.6.2 Algorithmic Details

Before presenting *Bottom_Solutions()* and *Top_Solutions()*, we first discuss the basic pruning operations used in the algorithm.

Previously, a basic operation in our algorithms was the merging of two *load-required_time* sets. If we had two sets S_1 and S_2 where the solutions in each set consumed identical power p (or we were not concerned with power), we computed a new set $S_{1,2} = (S_1 \cup S_2) \setminus S^*$ where S^* was a set of provably sub-optimal solutions by Property 2.4.2; i.e., $(c, q) \in S^*$ implied that $\exists (c', q') \in S_{1,2}$ s.t. $c' \leq c$ and $q' \geq q$.

Algorithm: *Merge_Load-Func_Sets*(S_1, S_2)
Let $S' \leftarrow S_1 \cup S_2$
$f_{\max}(x) = -\infty \quad \forall x$
Foreach $(c, f) \in S'$ in increasing order of c
Let $f'(x) = f(x)$ if $f(x) > f_{\max}(x)$
$f'(x) = -\infty$ otherwise
$S' \leftarrow S' \setminus \{(c, f)\}$
If $\exists x$ s.t. $f'(x) \neq -\infty$
$S' \leftarrow S' \cup \{(c, f')\}$
/* *Otherwise,* (c, f') *is useless* */
$f_{\max} \leftarrow \mathrm{pwl\_max}(f_{\max}, f)$
Return S'

Figure 2.8: Algorithm for merging load-func sets (presumably consuming the same power)

We generalize this concept for *load-required_time_func* sets. The property analogous to Property 2.4.2 is the following:

Property 2.6.1 *Let load-required_time set* $S = \{(c_1, f_1), .., (c_m, f_m)\}$ *be ordered by* *load; i.e.* $c_i < c_{i+1}$*. For* $(c_i, f_i) \in S$ *let* f_i' *be defined for all* x *as:*

$$f_i'(x) = \begin{cases} f_i(x) & \textit{if } pwl\_max(f_1(x), .., f_{i-1}(x)) > f_i(x) \\ -\infty & \textit{otherwise.} \end{cases}$$

We may replace $(c_i, f_i) \in S$ *with* (c_i, f_i') *while maintaining optimality. Further, if* $f_i'(x) = -\infty \; \forall x$*, then* (c_i, f_i') *may be eliminated altogether.*

This property is implemented in Figure 2.8 as the routine *Merge_Load_Func_Sets()* and is used by both *Bottom_Solutions()* and *Top_Solutions()*.

In addition, the previous low power algorithm utilized a more general pruning method given by Property 2.5.1 which said that, for a pair of solutions (p, c, q) and (p', c', q'), if $p < p'$, $c \leq c'$ and $q \geq q'$ then (p', c', q') is sub-optimal and may be eliminated. The analogous property for the slew sensitive generalization is as follows.

Property 2.6.2 *For a solution* (p, c, f)*, let* S' *be the set of all other solutions s.t.* $(p_i, c_i, f_i) \in S'$ *iff* $p_i < p$ *and* $c_i \leq c$*. Further, let* $f_{\max} = pwl\_max(f_i)$ *over all* $(p_i, c_i, f_i) \in S'$*. In a manner similar to Property 2.6.1, we may replace* f *with*

$$f'(x) = \begin{cases} f(x) & \textit{if } f(x) > f_{\max}(x) \\ -\infty & \textit{otherwise.} \end{cases}$$

Again, if $f'(x) = -\infty \;\; \forall x$*, then* (p, c, f) *may be eliminated entirely.*

A generalization of the data-structure in Section 2.5 is be presented later in this section to efficiently implement the pruning of Property 2.6.2.

Finally, we give pseudo-code for *Bottom_Solutions()* and *Top_Solutions()* in Figures 2.9 and 2.10 respectively. The routines follow the previous versions quite closely. However, because Properties 2.6.1 and 2.6.2 introduce partially defined functions there is no total order on the (c, f) sets. Therefore, when combining solutions of children, we may look at all such pairs in the worst case rather than performing a simple "merge" as previously (Figure 2.2, lines 7-12). This can be seen in the *for loop* of *Bottom_Solutions()* in Figure 2.9 – for a particular $f_l[i]$, we must find the appropriate $f_r[j]$'s to pair with $f_l[i]$. In the worst case, all j's may be candidates –

Algorithm: *Bottom_Solutions*$(v, S_t(l(v)), S_t(r(v)))$

```
B′ ← B ∪ {φ} where φ indicates "no buffer", c_φ = 0
S_b(v) ← ∅
Foreach triple (p_l, S_{p_l}) ∈ S_t(l(v)), (p_r, S_{p_r}) ∈ S_t(r(v)), b ∈ B′
        in increasing order of p = p_l + p_r + c_b
  /* Let (c_l, f_l) ∈ S_{p_l} and (c_r, f_r) ∈ S_{p_r} be indexed */
  /* E.g. (c_l[i], f_l[i]) is the ith smallest load in */
  /* and corresponding piece-wise linear function in S_{p_l} */
  For i = 1..|S_{p_l}|
    If (i = 1)
      j ← 1
    Else
      Let j be the smallest index s.t. f_l[i − 1](x) < f_r[j](x) for some x
    While ∃x s.t. f_r[j](x) < f_l[i](x)
      f_min ←pwl_min(f_l[i], f_r[j])
      If (b = φ)
        f′ ← f_min
        c′ ← c_l[i] + c_r[j]
      Else
        x ← (c_l[i] + c_r[j]) · r_b
        q ← f_min(x)        /* Req. arrival time at output of buffer */
        q ← q − (c_l[i] + c_r[j])r_b − d_b
        c′ ← c_l[i] + c_r[j]
        Let f′(x) = q − λ_b x  ∀x        /* One segment pwl func. */
      S′ ← S′ ∪ {(c′, f′)}
      j ← j + 1
    Prune solutions in S′ per Property 2.6.2
    If (∃(p, S_p) ∈ S_b)
      S_b ← S_b − {(p, S_p)}
      S′_p ← Merge_Load-Func_Sets(S′, S_p)
      S_b ← S_b ∪ {(p, S′_p)}
    Else      /* New power value */
      S_b ← S_b ∪ {(p, S′)}
```

Figure 2.9: *Bottom_Solutions()* routine for low power and signal slew

and this may hold for all i's. However, in practice these sets tend to remain linear in size. A related issue is the complexity of the functions themselves. In principle, the size of the functions can grow exponentially. However, again we do not observe this phenomenon in practice.

Algorithm: $Top\_Solutions(v, S_b(v))$

$S_t(v) \leftarrow \emptyset$
Foreach pair $w \in 1..W$, $(p_b, S_{p_b}) \in S_b$ in increasing order of
  $p = p_b + \alpha w l_{e_v}$
  Foreach $(c, f) \in S_{p_b}$
    $d \leftarrow \text{elmore}(e_v)$ (at width w)
    $S_p \leftarrow S_p \cup \{(c + \alpha w l_{e_v}, \text{pwl\_add\_scalar}(f, -d)\}$ ($S_p = \emptyset$ initially)
  Prune sub-optimal solutions from S_p by Property 2.6.1
  Prune sub-optimal solutions from S_p by Property 2.6.2
  If $(S_p \neq \emptyset)$
    $S_t(v) \leftarrow S_t(v) \cup \{(p, S_p)\}$

Figure 2.10: *Top_Solutions()* routine for low power and signal slew

2.6.3 Examples

To give some intuition regarding the operation of the slew version of the algorithm, we now give some illustrations. Figure 2.11 shows function f having load

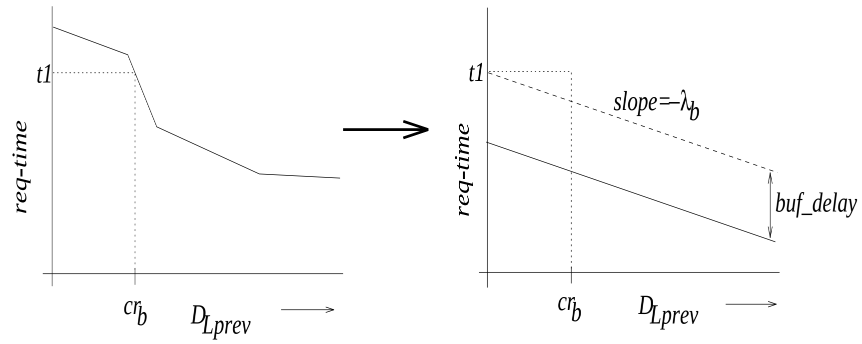


Figure 2.11: Effect of inserting a buffer

c before and after buffer b is considered for insertion at v as is performed in *Bottom_Solutions()*. Since b will drive load c, D_{Lprev} for its descendants is cr_b. Therefore, the required arrival time *at the output* of the buffer is $t1$. Subtracting the buffer delay *buf_delay*, we have the required arrival time function *at the input* of the buffer shown as a solid line in the right figure with slope $-\lambda_b$.

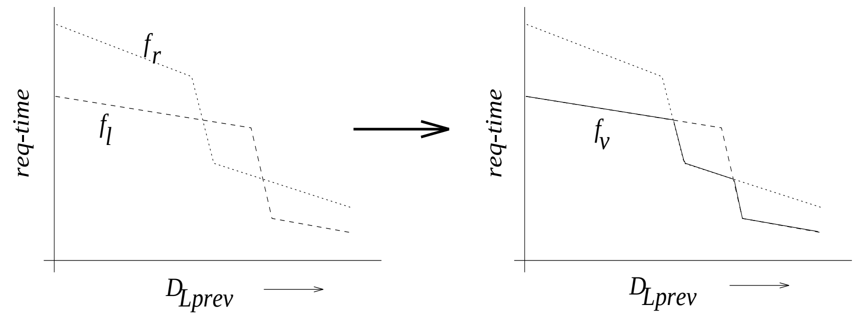


Figure 2.12: Combination of left and right solutions

Figure 2.12 illustrates the operation of combining left and right ("top") solutions. For some values of D_{Lprev}, the left subtree is critical and for others, the right is. The solid line in the graph on the right shows the function resulting from the *pwl_min* operation to capture the combination of these two solutions.

2.6.4 Implementation of Property 2.6.2

We now give a generalization of the data-structure of Section 2.5.1 to implement Property 2.6.2. Given a set of pairs (c_i, f_i) and a candidate pair (c, f), we want to efficiently compute f' as described in Property 2.6.2. (Recall that as the algorithm is organized, the power p associated with (c, f) is strictly greater than each p_i associated with each (c_i, f_i)).

To accomplish this we alter the previous augmented tree data structure to store the pair $(t.c, t.f_{l\_max})$ at each tree node t rather than $(t.c, t.q_{l\_max})$. As before, the search tree is ordered by c. The piece-wise linear function $t.f_{l\_max}$ is $pwl\_max(f_i)$ over all f_i where (c_i, f_i) resides either at t $(c_i = t.c)$ or in its left subtree.

Pseudo code implementing Property 2.6.2 is given in Figure 2.13 – i.e., given (c, f) we return the portions of f which are *not* sub-optimal in the form of f'. Updating the data-structure can be done by recursively traversing the search tree and

Algorithm: $PWLPrune(t, c, f)$

if t=NULL return f
else if $c < t.c$
    return $PWLPrune(t.left, c, f)$
else if $c \geq t.c$
    Define f' as:
        $f'(x) = f(x)$ if $f(x) > t.f_{l\_max}(x)$
        $f'(x) = -\infty$ otherwise
    if ($f'(x) = -\infty \;\; \forall x$ or $t.c = c$)
        return f'
    else
        return $PWLPrune(t.right, c, f')$

Figure 2.13: Implementation of Property 2.6.2

updating the f_{l_max} function at each node as we go.

The complexity of these operations is logarithmic in the number of entries in the data structure multiplied by the average complexity of the piece-wise linear operations (which has been very small in our experiments).

2.7 Experimental Results

We have implemented our algorithms under the C/UNIX environment on a Sun SPARC 20 workstation. We ran our algorithms on randomly generated routing topologies of various sizes with non-uniform segment lengths. In these experiments, discretization was done on an arbitrarily large integer domain (e.g. $1,000,000$), and yet impressive run-times were obtained.

Since our algorithms derive optimal solutions, the main focus of our experiments were run-time, the nature of the tradeoff curves and the effect of signal slew.

We used 5 different buffer types; the smallest (1X) buffer having $r_b = 3170\Omega$, c_b = 10fF, d_b = 300ps and $\lambda_b = .08$. The largest buffer was 8X. Intrinsic delay d_b was identical for all buffers and λ_b was assumed to be inversely proportional to width (largest λ_b being 0.08, smallest 0.01). Our experiments used a variety of wire widths from $0.5\mu m$ to $5\mu m$ (additional benefit typically wasn't observed for our test cases beyond this width). In these experiments, we used maximum delay as the metric (i.e., all required times of sinks were 0).

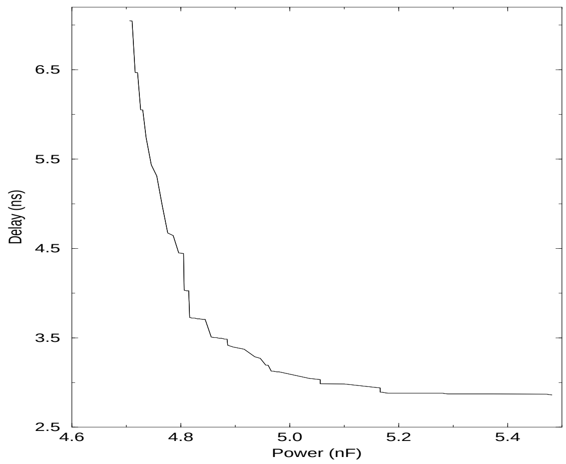


Figure 2.14: Power vs. Delay curve for a 20 sink net

Figure 2.14 shows the optimal power vs. delay curve for a 20 sink net. We utilized both wire sizing and buffer insertion on this example. Observed run times for nets of this size are typically in the 20-30 second range. The unsized/unbuffered delay is at the left-most point and the minimum delay solution is at the right-most point of the curve. Clearly much better engineering choices appear at the "elbow" of the curve.

Our second set of experiments appear in Table 2.1. Here we show the importance of taking slew into account for buffer insertion. We performed experiments on nets ranging from 10 to 30 sinks. We introduced a secondary variable K which was a scaling coefficient for λ_b; in other words, for a particular value of K, we replaced each λ_b with $K\lambda_b$. For each net size, we performed the following experiment: first we ran the basic algorithm (i.e., non-slew) on the topology to give the minimal delay solution. We then evaluated that topology taking into account the signal slew model, giving a new delay value. We did the same using the extended algorithm of Section 2.6. The figures in the table indicate the ratio of the resulting delay values. For instance, a value of 1.5 would indicate that by taking slew into account, we can reduce the resulting delay by one third for the minimal delay solution (up to the accuracy of the delay model). The right-most column is the worst run-time among all experiments in that row.[3] As K and n grow we see large variation between observed delays. The purpose of this table is to give an idea of the trend as λ_b increases. Values for λ_b over 0.5 appear to be possible[18]. Thus, values of K in the $8-16$ range give perhaps the most interesting information. Nevertheless, we must point out that these results should only be interpreted as a trend since the sensitivity parameters are not derived from a true technology library (we have not yet been able to obtain such data from industry sources). Issues such as the relationship between component size and sensitivity clearly play an important role here.

n	$K=1$	$K=2$	$K=4$	$K=8$	$K=16$	$K=32$	max cpu
10	1.0	1.0	1.0	1.0	1.0	1.0	0.2
15	1.0	1.0	1.0	1.06	1.15	1.35	0.8
20	1.02	1.02	1.04	1.08	1.16	1.26	1.3
25	1.0	1.0	1.0	1.0	1.04	1.19	3.8
30	1.01	1.03	1.06	1.13	1.26	1.49	6.8

Table 2.1: Basic Algorithm vs. Slew Algorithm for various sensitivities. Values are delay ratios.

2.8 Conclusions

We have presented efficient algorithms for optimal wire sizing and buffer insertion. We adopted the flexible problem formulation of minimizing power subject to timing constraints and alternatively can compute the entire power-delay tradeoff curve with no additional complexity. The algorithm can easily be adapted to perform area minimization. In addition we incorporateed the contribution of signal slew into the delay model which has been shown to be a significant contributor to overall delay.

[3] Run-times are of the low-power variant.

Chapter 3

Topology Synthesis

In the previous chapter, we presented techniques for optimizing a static routing topology by wire sizing and buffer insertion. In this chapter, we turn our attention to the problem of finding the routing topology itself. The technique presented is also able to perform simultaneous wire sizing.

The proposed approach is a departure from previous methods in that, as in the previous chapter, we are able to explicitly capture the notion of an area/delay tradeoff. As a result, we not only improve on the delay achieved by previous approaches but, perhaps more importantly, we are able to produce high-performance routing trees with substantially less overhead in routing area versus previous approaches.

3.1 Background and Overview

3.1.1 Previous Work

Among performance driven routing techniques, the SERT algorithm of Boese et. al. [3] has produced some of the best delay figures reported in the literature. While the performance of the routing trees produced by SERT is promising, the area overhead is discouragingly large. Because of the greedy nature of the algorithm, it often results in star-like topologies for some technologies - i.e., each sink connects to a point very near the driver or to the driver itself. The practical utility of such topologies is unclear since the routing area they consume is so high and the congestion near the driver may be intolerable. The later work of Hodes et. al. [19] had some success in reducing this tendency of SERT by incorporating the notion of wire sizing to drive the construction.

Other performance driven routing work includes the A-Tree algorithm of Cong et. al. [6] which attempts to find a min-area routing tree which is also a shortest paths tree. Thus, the A-Tree algorithm uses a geometric abstraction and a linear delay model and suffers from the inherent inaccuracies of not using an RC model and not handling such common phenomena as asymmetric loads (and in fact does not take technological parameters into account at all). Nevertheless, the routing trees it produces are often of high quality. The work of Vittal and Marek-Sadowska [49] used the abstraction of Alphabetic Trees to heuristically construct routing trees. Their reported results are comparable to those of SERT in both delay and area.

Previous work in static topology wire sizing appears in Chapter 2.

3.1.2 Contributions

We present two algorithms: the P-Tree$_A$ algorithm whose goal is area minimization and the P-Tree$_{AT}$ algorithm which derives an area/delay tradeoff curve under the Elmore delay model and also incorporates simultaneous wire-sizing. Central to this work is the notion of routing topologies being induced by a permutation on the sinks of a net and the mapping of such topologies to a routing domain Thus, the routing topologies we produce are *Permutation-constrained routing Trees*, giving rise to the name *P-Tree*.

Experimentally the approach has proved effective. When delay is set to match previous approaches we observe substantial reductions in area overhead – for instance, previous approaches can consume over 70% more area on average than the P-Tree$_{AT}$ solution while achieving the same or worse delay. We also observe substantial improvements under the AT metric – i.e., the product of delay and area. When minimum delay is desired we also improve significantly over previous approaches. These trends hold whether wire sizing is incorporated or not.

While the preceding represent the fundamental contributions of this work, there are two interesting side-effects as well.

First, as an interesting special case, the P-Tree$_A$ algorithm has achieved high quality results for area minimization. We have devised a scheme for systematically improving a given topology. This is done by deriving a sink permutation from the given topology. By utilizing this technique with the Minimum Spanning Tree as the given topology we approach the solution quality of the Batched 1-Steiner (B1S) algorithm of Kahng and Robins [26] which has reported among the best results in the literature. When using the result of Batched 1-Steiner as the given topology, we are able to improve routing area by 0.38% on average for 20-pin nets, further closing the gap on the Minimum Rectilinear Steiner Tree.

Second, we provide empirical data giving insight into the nature of performance driven routing. The classical result of Hanan tells us that only points on the "Hanan Grid" need be considered as candidate Steiner points to achieve minimum area. In [5] Boese et. al. showned that Hanan's theorem does not hold when minimizing maximum source-to-sink Elmore delays – sometimes the optimal placement of Steiner nodes is not on the Hanan Grid. However, the authors suggest that they believe the error caused by considering only Hanan Grid points is vanishingly small. Our formulation permits introduction of points off of the Hanan Grid and thereby allows empirical evaluation of their impact on solution quality. We show that extending the Hanan Grid from nxn to $(2n-1)$x$(2n-1)$ yields an extremely small improvement in delay - yielding only a 0.05% improvement on average, lending support to the contention of [5].

3.1.3 Algorithm Overview

As mentioned previously, a key concept in our approach is the idea of routing topologies being induced by a sink permutation. The idea is as follows. Given a permutation on the sinks of a net, consider the (exponentially large) set of binary search trees with sinks identified with leaves that obey the order of the permutation – i.e., an in-order traversal of the tree visits the terminals in the order of the permutation. Figure 3.1 illustrates this concept by showing four trees which obey the permutation "a b c d e" (with the driver pin of the net being labeled *f*). Now consider the possible mapping of the internal nodes of such a tree to a set of points in the plane or the vertices of a routing graph. It is this set of routing solutions to which we constrain our search.

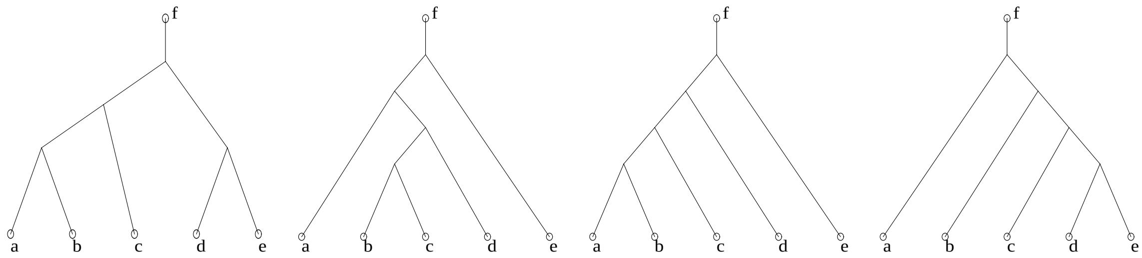


Figure 3.1: Example trees induced by sink ordering "a b c d e" (driver is f)

Intuitively, if we take care in finding a high quality permutation, the solution space will be sufficiently rich, containing many high quality solutions. On the other hand, the solution space is sufficiently constrainted to allow efficient identification of such solutions. Furthermore, with respect to minimizing area, our techniques for finding such high quality permutations can guarantee a solution at least as good as that of a given structure (e.g., a Minimum Spanning Tree or the result of another Steiner Tree heuristic).

Thus, this work is a departure from the constructive greedy heuristics proposed in the past (e.g., [3], [19]). Our approach bears some similarity to the Alphabetic Tree approach of [49] in that both approaches draw their solutions from the set of binary trees induced by a sink permutation. However, our approach differs in a number of fundamental ways. First, placement of Steiner nodes in [49] is done heuristically while we go to great effort to optimally select their placement. Second, the objective function used in [49] is a weighted sum of source-to-sink delays while we minimize maximum delay or, more generally, maximize required arrival time; we propose the required-time formulation is preferred because it deals directly with the timing requirements seen in practice. Third, area overhead was not taken into account in [49] while our approach derives an explicit area/delay tradeoff. Fourth, we use more sophisticated techniques for finding high-quality sink permutations. Finally, we incorporate simultaneous wire sizing.

The remainder of the paper is organized as follows. Section 3.2 gives preliminary information on delay models and graph theoretic background, Section 3.3 presents an algorithm for constructing sink permutations, Section 3.4 presents the P-Tree$_A$ and P-Tree$_{AT}$ routing algorithms for a given sink permutation, Section 3.5 gives experimental results and we conclude in Section 3.6.

3.2 Preliminaries

3.2.1 Delay Models

As in Chapter 2, we adopt the Elmore delay model [13]. To review the Elmore model, the delay of a wire segment $e = (u, v)$ transmitting a signal from node u to node v is defined as follows. Let r_e and c_e be the resistance and capacitance of e respectively (both of which are proportional to wire length l_e). Further, let $c(T_v)$ be the capacitive load at node v. The delay of the segment is expressed as

$$r_e(\frac{c_e}{2} + c(T_v)).$$

When wire sizing is taken into account, we need a model for resistance r_e, and capacitance c_e as a function of width w_e. A typical model is the following:

$$c_e = \alpha l_e \cdot w_e \qquad\qquad r_e = \beta l_e / w_e$$

where α and β are characteristic constants. For consistency with previous work, we adopt this model.

Delay of a driver g is defined similarly as

$$d_g + r_g \cdot c_{load}$$

where d_g is a constant known as intrinsic delay, r_g is the output resistance of the driver and c_{load} is the capacitive load on g's output. Since d_g is constant, it may be neglected in relative comparisons of routing algorithms. We note also that since we can consider the slew based delay model presented in Chapter 2 to model driver delay. However, since it depends on the delay of the *previous* stage, it can be considered a constant and incorporated into the intrinsic delay.

Again, as in Chapter 2, we adopt a *required arrival time* formulation. Given a timing constraint on each sink u in the form of required arrival time q_u, the required arrival time for a routing tree T driven by gate g is

$$q(T) = \min_{sinks\ u} (q_u - delay(g \rightarrow u))$$

where we include driver delay in the equation. If $q(T) \geq 0$, T is said to meet its timing requirements. A special case of required arrival time is maximum delay: letting $q_u = 0\,\forall u$, the max source to sink delay is $-q(T)$.

3.2.2 Graphs, Trees and Permutations

One of the classical results in the theory of Rectilinear Steiner Minimum Trees (RSMTs) is Hanan's theorem [17] which states that there always exists a RSMT for a terminal set N where all Steiner points are drawn from the n^2 points formed by the intersection of horizontal and vertical lines drawn through the terminals ($n = |N|$).

The grid formed by this set of points leads to a useful graph-theoretic formulation of the problem based on the the *Grid Graph* of N.

Definition 3.2.1 Grid Graph [21]: *Given a set of terminals N, N's* Grid Graph *$GG(N) = (V, E)$ is defined by the following process:*

Construct vertical and horizontal lines through each terminal.

Let V be identified with the set of intersection points of these lines.

There is an edge in E between two vertices iff the corresponding intersection points are connected by a single horizontal or vertical segment.

Our algorithms produce routing topologies embedded in a graph structure such as $GG(N)$. We will use $v_{\{r,c\}}$ to denote the vertex in $GG(N)$ located at the r'th row and c'th column (position $(1,1)$ being upper left). For convenience, we also use v_i and v_d to denote the vertex in $GG(N)$ coincident with the i'th pin of the permutation and the vertex coincident with the driving pin of the net respectively.

For completeness we provide the following definitions relating pin permutations and routing topologies.

Definition 3.2.2 Permutation Induced Abstract Topology: *Consider a terminal set N and a permutation π on the sinks of N. A binary tree T is an abstract topology induced by π if its leaves are identified with the terminals in N and it obeys the ordering imposed by π when T is interpreted as a binary search tree (the driver being implicitly attached to the root).*

Definition 3.2.3 Abstract Topology Embedding: *Given a abstract topology T (as a binary tree), leaves identified with the terminal set N and a target graph $G = (V, E)$ (e.g., $G = GG(N)$), an embedding of T into G is a mapping of the internal nodes of T to V.*

To avoid confusion, we use the term "abstract topology" to emphasize that such a topology is not a true routing topology in that its internal nodes are not mapped. Only when such a topology is embedded in the plane or a routing graph such as $GG(N)$, does a physical routing topology result.

We propose the notion of "consistency" as a convenient way of assessing the quality of a pin permutation. If we can show that a permutation π is consistent with a given routing topology T such as the Minimum Spanning Tree (MST), we ensure that the min-area solution induced by π is no worse than T. A similar notion is applied to a hierarchical decomposition of a terminal set, i.e., a binary tree with leaves identified with pins. Formally we have the following.

Definition 3.2.4 Consistency: *Given a set of terminals N and a tree structure T embedded in a routing graph and connecting N, a permutation π on N is* consistent *with T if π can induce a structure isomorphic to T (up to 0-length edges). Further, let H be a binary tree with all points $p \in N$ represented as leaves. We call H a* hierarchical decomposition *of N. H is* consistent *with T if* all possible depth-first traversals of *H visit the leaves in an order consistent with T.*

Thus, if we can prove that an algorithm for constructing a permutation ensures consistency with the MST, we guarantee that the min area solution induced by π is no worse than $\frac{3}{2}$ times the optimal Steiner tree [22] while only relying on the MST indirectly in constructing the final topology. We present an algorithm for constructing permutations consistent with a given topology T such as the MST in the next section. Previous permutation based approaches such as [49] provide no such assurance. Rather, they derive a sink permutation from a circular ordering of the sinks with the driver as the center of the circle which can result in unbounded overhead vs. the MST.

3.3 Finding High Quality Sink Permutations

The method we propose for constructing high quality permutations is broken into three phases: (1) construction of a hierarchical decomposition of the terminals consistent with a given structure such as the MST, (2) reorienting the hierarchical structure such that the driver is attached to the root while maintaining consistency and (3) application of a novel dynamic programming algorithm to further optimize the induced permutation.

Phase 3 is based on the following observation. Since the hierarchy from step (1) is consistent with the given topology, any depth-first traversal of the hierarchy

Algorithm: MST_to_hierarchy
Let T_m be a rectilinear Minimum Spanning Tree on terminal set N For each $v \in T_m$, let $h(v)$ be a single node tree labeled v Repeat $n-1$ times 1. select a leaf node v and its parent u in T_m 2. delete edge (u, v) from T_m 3. replace $h(u)$ with a new tree t where $t.\mathit{left} = h(u)$ and $t.\mathit{right} = h(v)$ return last tree formed

Figure 3.2: Pseudo-code to Construct Hierarchy Consistent with MST

visits the terminals in an order consistent with the given topology (e.g., MST). Can we perform such a traversal in a way to optimize the resulting permutation? In other words, we have a binary tree and we wish to assign a 0 or 1 to each internal node of the tree where 0 indicates that left subtree is explored first and 1 indicates that the right subtree is explored first. We propose that an appropriate metric for the quality of such an assignment is the tour length of the permutation induced by the resulting traversal in the traditional sense of the Travelling Salesman Problem. The intuition is that good TSP tours provide good clustering information which can identify good candidate sets for sub-trees of a good Steiner Tree. We present a dynamic programming algorithm for solving this sub-problem optimally.

3.3.1 Phases 1 and 2: Hierarchy Construction and Re-Orientation

Consider the algorithm *MST_to_hierarchy* given in Figure 3.2 which takes a Minimum Spanning Tree T_m and produces a hierarchy. The following consistency lemma holds by induction on the sub-trees the algorithm computes.

Lemma 3.3.1 *The hierarchy formed by Algorithm* MST_to_hierarchy *in Figure 3.2 is consistent with the minimum spanning tree of the given point set.*

Figure 3.3 illustrates the operation of *MST_to_hierarchy*, part (a) giving an example of a point set and its MST, and part (b) showing a hierarchical decomposition

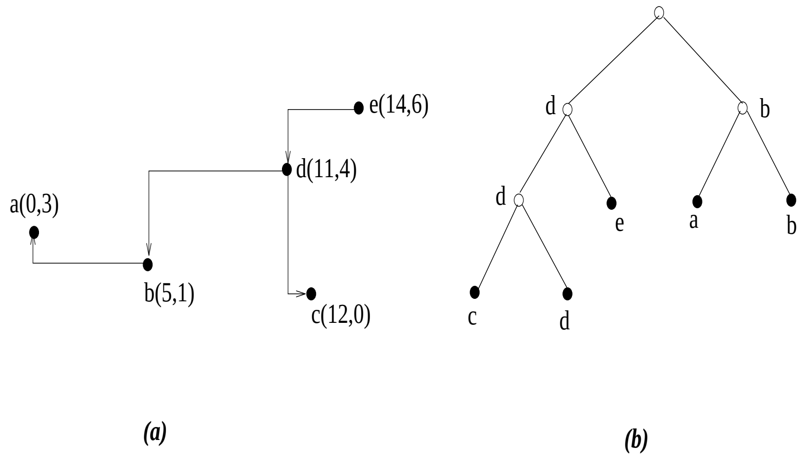


Figure 3.3: Constructing a Hierarchy Consistent with MST

consistent with the MST produced by the algorithm. Labels on internal nodes of the hierarchy indicate the parent node from the merge step in the algorithm.

In the case where the given structure is a Steiner Tree from another algorithm rather than a Spanning Tree, the algorithm may be modified as follows to deal with Steiner nodes. First, for each Steiner node u, $h(u)$ is initially NULL (for terminal nodes v, $h(v)$ is initialized as before). Second, when constructing a new tree in step 3 of the loop, if one of the subtrees is NULL, we simply copy the other tree. Finally, the number of times we execute the loop now equals the number of edges in the given Steiner tree.

The reader may have observed that in step 1 of the loop, there may be multiple candidate leaves v from which to choose and that the order in which they are chosen will impact the final hierarchy. We have examined several heuristics for selecting the next leaf. We have had success with the following: select v such that the depth of the tree resulting from step 3 is minimized; if there is a tie, select from among the candidates the node with minimum distance to it's parent.

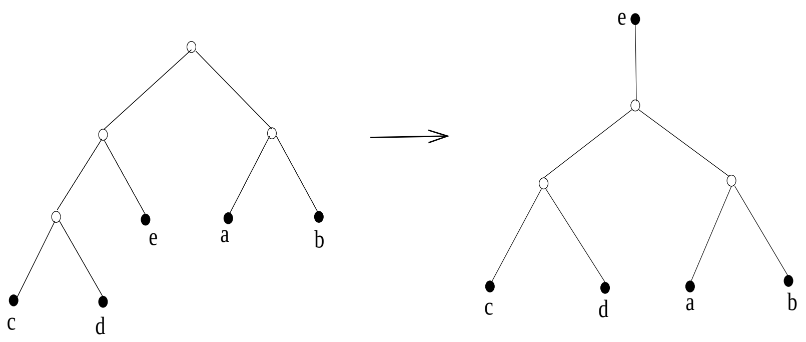
The intuition behind the heuristic is that a more balanced hierarchy is more likely to be better optimized by the subsequent tour length minimization phase and that nodes near each other should be clustered at the lower levels of the hierarchy.

After the hierarchy has been established, we must reorient the tree such that the driver is associated with the root. This is a necessary step since the signal direction is critical in performance driven routing: we want the signal to flow from root to leaves. Such reorientation does not disturb the consistency of the hierarchy and is illustrated in Figure 3.4 (note that the original root becomes a degree 2 internal node and therefore is redundant and eliminated).

3.3.2 Phase 3: Tour Length Minimization

In this phase we solve the following problem.

Problem 3.3.1 *Given a binary tree where leaves are identified with points in the plane, consider a labeling $s(v) \in \{0,1\}$ of each internal node v with the following interpretation: if $s(v) = 0$, visit v's left child first, if $s(v) = 1$, visit v's right child first. Over all such labelings, find the one such that the tour length of the order in which the leaves are visited is minimized.*



PERM	TOUR LEN
c d a b	24
d c a b	27
c d b a	21
d c b a	20

Figure 3.4: Re-orientation to reflect driver and possible permutations induced by reoriented tree

Referring again to Figure 3.4, we see the sink permutations which can be derived from the reoriented hierarchy and their (non-circular) tour length, “d c b a” being optimal. (There are actually 8 possible permutations corresponding to the 2^3 possible labelings of the internal nodes – the four shown and their reverse).

We solve Problem 3.3.2 by dynamic programming. For each node v in the tree let $P_l(v)$ and $P_r(v)$ be the sets of sink pins in v's left and right sub-trees respectively. For each pair $i \in P_l(v), j \in P_r(v)$, we compute $D_v(i,j)$, the minimum length tour of the terminal set $P_l(v) \cup P_r(v)$ beginning at i and ending at j over all possible labelings of the internal nodes of the subtree rooted at v. Once we have computed $D_v(i,j)$ at the root for all possible i,j, we simply select the optimal solution. (This may be a circular tour or one which includes the driver itself depending on implementation).

For a given v, i and j, let $l(v)$ and $r(v)$ be v's left and right children respectively. We describe $D_v(i,j)$ by the following recurrence

$$D_v(i,j) = \min_{k_1 \in P_l(v) \setminus \{i\}, k_2 \in P_r(v) \setminus \{j\}} \{D_{l(v)}(i,k_1) + d(k_1,k_2) + D_{r(v)}(k_2,j)\} \tag{3.1}$$

where $d(k_1,k_2)$ is the rectilinear distance between terminals k_1 and k_2. By symmetry we also know that $D_v(j,i) = D_v(i,j)$.

A naive algorithm for computing D_v has $O(n^4)$ complexity; i.e. we examine each i, j, k_1, k_2 quadruple. This gives an overall run-time of $O(n^5)$ since this must be performed for each node in the tree.

However, a run time of $O(n^4)$ can be achieved by reducing the computation of D_v to $O(n^3)$. For a particular node v, consider the directed graph $G_{tsp} = (V,E)$ defined subsequently (all edges not explicitly defined have weight ∞) and Lemma 3.3.2.

For each terminal $i \in P_l(v) \cup P_r(v)$, we have nodes i and i' in V

For each ordered pair $i, k \in P_l(v)$, introduce edge (i, k') with weight $w(i,k') = D_{l(v)}(i,k)$

For each pair $k_1 \in P_l(v), k_2 \in P_r(v)$, introduce edge (k_1', k_2') with weight $w(k_1', k_2') = d(k_1, k_2)$

For each ordered pair $k_2, j \in P_r(v)$, introduce edge (k_2', j) with weight $w(k_2', j) = D_{r(v)}(k_2, j)$.

Lemma 3.3.2 *The shortest path from node $i \in P_l(v)$ to $j \in P_r(v)$ in $G_{\ tsp}$ visits nodes k_1' and k_2' where k_1 and k_2 minimize expression 3.1 and the path length equals the tour length.*

The idea is that a path from a vertex $i \in P_l(v)$ to a vertex $j \in P_r(v)$ must be of the form i, k_1', k_2', j where k_1 and k_2 are possible intermediate nodes in the recurrence (Equation 3.1) and the sum of the edge weights matches the sum in the recurrence. By Lemma 3.3.2 we can compute $D_v(i, j)$ for all i, j by the Floyd-Warshall all-pairs shortest paths algorithm[9].

Since the Floyd-Warshall algorithm runs in $O(n^3)$ time and is applied at each of the internal nodes of the tree, we have the following.

Lemma 3.3.3 *Problem 3.3.2 is solved in $O(n^4)$ time by the above scheme.*

3.4 Tree Construction For a Given Permutation

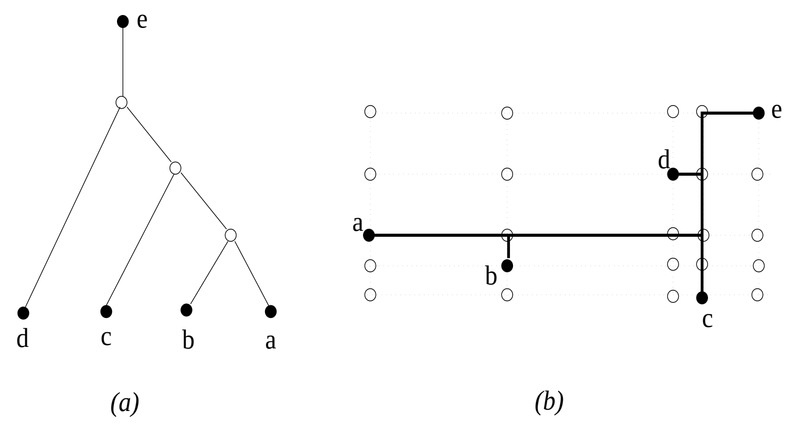


Figure 3.5: A permutation-induced abstract topology and an embedding in $GG(N)$.

This section describes the P-Tree algorithms which take as input a sink permutation established by the techniques of the previous section. Again using the

the example point set begun in Figure 3.3, we give Figure 3.5 where an abstract topology is induced in 3.5(a) and that abstract topology is embedded in $GG(N)$ to form the routing topology in 3.5(b) (yielding, in this case, a min-area Steiner Tree). It is this process which the P-Tree algorithms perform optimally.

We first give an overview and subsequently describe the P-Tree$_A$ algorithm for area minimization and then the P-Tree$_{AT}$ algorithm for deriving area/delay tradeoff in detail.

3.4.1 Overview

Recall the definition of the Grid Graph $GG(N) = (V, E)$ of a terminal set N ($n = |N|$); V contains n^2 vertices, $|E| = O(n^2)$ and all vertices have degree no greater than 4. Recall that $v_d \in V$ denotes the vertex associated with the driver of the net. We also refer to the i'th sink in the given permutation as sink i (i.e., without loss of generality we relabel the sinks based on the permutation). Also recall that $v_i \in V$ refers to the graph vertex coincident with sink i.

Our algorithms construct trees rooted at vertices $v \in V$. These trees have the property that they drive a consecutive subsequence of the sink permutation; the subsequence is defined by its starting and ending indices i and j. The trees provide the sub-solutions for a dynamic programming algorithm. More specifically, we inductively compute the following solutions or solution sets for a given sink permutation.

$S(v, i, j)$: the cost of the optimal solution(s) over all permutation induced (and embedded) routing topologies driving sinks $i..j$ and rooted at v.

$S_b(v, i, j)$: the cost of the optimal solution(s) over all permutation induced (and embedded) routing topologies driving sinks $i..j$ and rooted at v where v is additionally constrained to have degree 2 – i.e., v is a *branching point* ($i < j$).

Notice that solutions represented in S may either have v as a branching point or be single-stem trees rooted at v. The idea is to use $S_b(v', i, j)$ to compute $S(v, i, j)$.

1. Compute $S(v,i,i) \quad \forall v \in V, 1 \leq i < n$
2. For $I = 1..n-2$
3.   For $i = 1..n-1-I$
4. $j = i + I$
5.   Compute $S_b(v,i,j) \quad \forall v \in V$
6.   Compute $S(v,i,j) \quad \forall v \in V$ using results of step 5
7. $S(v_d, 1, n-1)$ now gives the sub-solution(s) to be paired with the driver.

Figure 3.6: Framework for Permutation Constrained Routing

In the case of minimizing area alone, the "costs" $S(v,i,j)$ and $S_b(v,i,j)$ are scalars giving the area of the optimal solution. The min area version of the algorithm can be thought of as a special case of the *Dreyfus-Wagner* (DW) algorithm [12] for the Graph Steiner Problem. The DW algorithm uses a similar decomposition into sub-solutions; the difference is that instead of terminal subsets being determined by consecutive subsequences of a permutation, the DW algorithm considers *all* 2^n subsets of terminals.

For performance driven routing, sub-solutions are more complex since not only are we interested in the area of the sub-solutions, but also their timing properties and their capacitance since this will affect the delay of ancestor wires and the driver. Fortunately, area is proportional to capacitance. Thus, the capacitance c and required arrival time q are the key properties of a routing tree. In much the same manner as in Chapter 2, we maintain sets of (c,q) pairs ordered by c – this gives the notion of cost as a tradeoff curve in the two dimensional space of c and q. Again, as in Chapter 2, these sets have the property that they are strictly increasing in both c and q. Properties of these sets and primitives manipulating them are given later in this Chapter.

While there are fundamental differences in the sub-solutions computed by P-Tree$_A$ and P-Tree$_{AT}$, both algorithms follow the same basic structure given in Figure 3.6. The variable I indicates the length of the sub-sequences we are currently considering. This ensures that $S(v,i,k)$ and $S(v,k+1,j)$ have previously been computed for all $k \in \{i..j-1\}$ when we reach step 5 in the loop. Similarly, $S(v,i,j)$ is derived in step 6 from the the result of step 5.

The details of the individual steps in Figure 3.6 for P-Tree$_A$ and P-Tree$_{AT}$ are given in the next two sections respectively.

3.4.2 Routing for Min-Area: The P-Tree$_A$ Algorithm

The first and simplest step in P-Tree$_A$ is computation of base cases $S(v,i,i)$. We are just interested in the length of a wire from v to pin i as follows.

$$S(v,i,i) = d(v,v_i) \quad \forall i \in \{1..n-1\}$$

where $d(v,v_i)$ is the rectilinear distance between vertices v and v_i in $GG(N)$.

Next we need to show how to compute $S_b(v,i,j)$. Since by definition of S_b, v must be a branching point we have the following.

$$S_b(v,i,j) = \min_{k \in \{i..j-1\}} \{S(v,i,k) + S(v,k+1,j)\}$$

Note that since $k-i < j-i$ and $j-k-1 < j-i$, $S(v,i,k)$ and $S(v,k+1,j)$ have been previously computed.

Computation of $S(v,i,j)$ is more complex. A naive approach is given by implementing the following.

$$S(v,i,j) = \min_{v' \in V} \{d(v,v') + S_b(v',i,j)\}$$

In other words, any single-stem tree rooted at vertex v connects to some branching point v'; we find the optimal such branching point. Since there are n^2 candidates for v', sequentially computing $S(v,i,j)$ for all v by this procedure takes $O(n^4)$ time for given i,j. On the other hand, this procedure only computes $O(n^2)$ values. The $O(n^4)$ complexity for this step is unnecessary. In fact, we can compute $S(v,i,j)$ for each v in $O(n^2)$ time.

The key in achieving the $O(n^2)$ run time is the underlying graph $GG(N)$. We can show by contradiction the following simple lemma relating the optimal solution

```
/* Phase 1 */
```
$LS(v_{\{r,1\}}) \leftarrow S_b(v_{\{r,1\}}) \quad \forall r \in \{1..n\}$
for $r = 1$ to n
    for $c = 2$ to n
        $LS(v_{\{r,c\}}) \leftarrow \min\{S_b(v_{\{r,c\}}), LS(v_{\{r,c-1\}}) + d(v_{\{r,c\}}, v_{\{r,c-1\}})\}$
/* Phase 2 */
$RS(v_{\{r,n\}}) \leftarrow LS(v_{\{r,n\}}) \quad \forall r \in \{1..n\}$
for $r = 1$ to n
    for $c = n-1$ to 1
        $RS(v_{\{r,c\}}) \leftarrow \min\{LS(v_{\{r,c\}}), RS(v_{\{r,c+1\}}) + d(v_{\{r,c\}}, v_{\{r,c+1\}})\}$
/* Phase 3 */
$US(v_{\{1,c\}}) \leftarrow RS(v_{\{1,c\}}) \quad \forall c \in \{1..n\}$
for $c = 1$ to n
    for $r = 2$ to n
        $US(v_{\{r,c\}}) \leftarrow \min\{RS(v_{\{r,c\}}), US(v_{\{r-1,c\}}) + d(v_{\{r,c\}}, v_{\{r-1,c\}})\}$
/* Phase 4 */
$S(v_{\{n,c\}}, i, j) \leftarrow US(v_{\{n,c\}}) \quad \forall c \in \{1..n\}$
for $c = 1$ to n
    for $r = n-1$ to 1
        $S(v_{\{r,c\}}, i, j) \leftarrow \min\{US(v_{\{r,c\}}), S(v_{\{r+1,c\}}, i, j) + d(v_{\{r,c\}}, v_{\{r+1,c\}})\}$

Figure 3.7: Computation of $S(v, i, j)\ \forall v \in V$ in P-Tree$_A$

at a vertex v to the optimal solutions at v's neighbors in $GG(N)$. This gives the optimal sub-structure property required for dynamic programming.

Lemma 3.4.1 *For fixed i, j, the routing tree giving $S(v, i, j)$ either uses v as a branching point (i.e., $S(v, i, j) = S_b(v, i, j)$)* or *$S(v, i, j) = S(v'', i, j) + d(v, v'')$ where v'' is one of v's immediate neighbors in $GG(N)$.*

Referring back to Figure 3.5, the idea of the lemma is illustrated by the vertices labeled v and v'' where $S(v, b, a)$ is derived from $S(v'', b, a)$.

Without loss of generality, assume that the wire connecting v to its optimal branching point v' first travels along v's column to v''s row and then finishes the trip horizontally to v'. This leads to the four phase strategy given in Figure 3.7 (for fixed i, j).

The algorithm computes intermediate solutions $LS(v)$, $RS(v)$ and $US(v)$, the meanings of which are summarized as follows.

$LS(v)$:	the area of the optimal single-stem tree rooted at v with branching point v' constrained to be at v or to v's *left* in the same row as v.
$RS(v)$:	the area of the optimal single-stem tree rooted at v with branching point v' constrainted to be in the same *row* as v.
$US(v)$:	the area of the optimal single-stem tree rooted at v with branching point v' constrained to be in v's row or a row *above* v ("up" from v).

At the beginning of each phase, we have a boot-strap step. For instance, if v is in the bottom row then $US(v)$ effectively has no constraint on its branching position v' and therefore equals $S(v, i, j)$. With this a a base case we inductively compute $S(v, i, j)$ from bottom to top. Once $US(v)$ has been computed, we are then able to compute $S(v, i, j)$ $\forall v$ in a single additional pass. Each phase is implemented in a manner similar to the computation of the prefix sum or prefix max of a list of numbers only examining an immediate neighbor to perform the computation (as provided by Lemma 3.4.1).

This process yields an $O(|V|) = O(n^2)$ complexity for computing, for all $v \in V$, $S(v, i, j)$ for a fixed i, j. Thus the overall complexity of *P-Tree*$_A$ is dominated by the computation of $S_b(v, i, j)$ which is $O(n)$ for each v, i, j triple. Since there are $O(n^4)$ such triples, this gives an overall complexity of $O(n^5)$.

Finally, we can show the optimality of the solution computed by P-Tree$_A$ inductively.

Lemma 3.4.2 $S(v_d, 1, n-1)$ *computed by the P-Tree*$_A$ *algorithm is the minimum area over all possible Rectilinear Steiner trees induced by given sink ordering* π*. This solution is computed in* $O(n^5)$ *time.*

3.4.3 Routing For Performance: The P-Tree$_{AT}$ Algorithm

We extend the P-Tree$_A$ algorithm of the previous section to the timing driven version of the algorithm, P-Tree$_{AT}$ which derives an area/delay tradeoff. The basic structure of P-Tree$_{AT}$ is essentially identical to P-Tree$_A$ except in the primitives we use to manipulate solutions. Recall that both $S_b(v,i,j)$ and $S(v,i,j)$ were scalars in P-Tree$_A$. By contrast, P-Tree$_{AT}$ manipulates sets of *load, required-time* pairs. We refer to such sets as *cq-sets* and introduce new primitives for manipulating these sets. For instance, a load, required-time pair $(c,q) \in S(v,i,j)$ indicates that there exists a legal routing topology rooted at v and driving sinks $i..j$ which has lumped capacitance c at v (also proportional to routing area) and achieves required-arrival time q at v. Note that since we may be performing simultaneous wire sizing, we not only need to determine a routing topology but also the width of each wire segment between Steiner nodes.

As in the preceding chapter, we use the following pruning property [48], [10].

Property 3.4.1 *For $(c,q),(c',q') \in S$, if $c' \geq c$ and $q' < q$ then (c',q') is sub-optimal.*

Intuitively, this notion is clear since a larger capacitance can only slow down a path from the driver. Thus, given a cq-set S, all (c',q') pairs as described by Property 4.1 may be eliminated. As a result we can organize cq-sets in strictly increasing order of c *and* q (recall that we prefer large required times). This ordering captures the notion of an area/required-time (delay) tradeoff and the effect of sub-solutions on ancestors, thus giving capacitance values c the dual role of capturing area overhead and load presented to parents.

To manipulate and combine cq-sets, we need some primitive operations as follows.

join_cq_sets(S_1,S_2): This primitive serves in computing $S_b(v,i,j)$. Its inputs are solution sets S_1 and S_2 both of which give solutions rooted at v, where S_1's solutions drive sinks $i..k$ and S_2's solutions drive sinks $k+1..j$ for some

$k \in \{i..j-1\}$. From these sets it produces a set of solutions rooted at v driving sinks $i..j$, the size of which is proportional to $|S_1| + |S_2|$.

augment_cq_set(S, l, w): This primitive takes cq-set S and computes a new cq-set where each $(c, q) \in S$ is augmented by a wire of length l and width w. This is used in constructing single stem solution sets $S(v, i, j)$.

prune_cq_set(S): This primitive takes a cq-set S and applies Property 4.1 to it, eliminating sub-optimal solutions.

The routine *join_cq_sets*(S_1, S_2) is implemented in a manner similar to the merging of two sorted lists. It performs the same task as was performed in Chapter 2 in computing unbuffered solution sets. Pseudo-code appears in Figure 3.8 (sets S_1 and S_2 are in increasing order of c and q and indexed). The routine produces the new cq-set by first pairing the solutions with minimum c. We then advance through the sets noticing that we can only improve the current solution by improving the required-time of the critical branch – improving the required-time of a non-critical branch results in a sub-optimal solution, i.e. one having more load than necessary.

```
S ← ∅
i ← 1 ; j ← 1
 While (i ≤ |S1| and j ≤ |S2|)
   Let (c1, q1) = S1[i]
   Let (c2, q2) = S2[j]
  S ← S ∪ {(c1 + c2, min(q1, q2))}
   If  (q1 ≤ q2)      /* S1 Critical */
      i ← i + 1
   If  (q2 ≤ q1)      /* S2 Critical */
      j ← j + 1
return S.
```

Figure 3.8: Primitive *join_cq_sets*(S_1, S_2)

The routine *augment_cq_set*(S, l, w) examines each $(c, q) \in S$ and adds to c the capacitance of a wire of length l and width w and subtracts the delay of such a

$S \leftarrow \emptyset$
for $w = 1..W$
    $S \leftarrow S \cup augment\_cq\_set(S(v_i, i, i), d(v, v_i), w)$
$S(v_i, i, i) \leftarrow prune\_cq\_set(S)$

Figure 3.9: Computation of $S(v, i, i)$ in P-Tree$_{AT}$ $\quad (v \neq v_i)$

wire from q to give (c', q') as

$$c' = c + \alpha wl \qquad\qquad q' = q - \frac{\beta l}{w}(\frac{\alpha lw}{2} + c).$$

Finally, we implement $prune\_cq\_set(S)$ by making a linear pass over S in increasing order of c eliminating sub-optimal solutions as we go (by comparing each (c, q) pair with its predecessor).

To get started we need the base case

$$S(v_i, i, i) \leftarrow \{(c_i, q_i)\} \quad \forall i \in \{1..n-1\}.$$

where c_i and q_i are the input capacitance and required arrival time for sink i respectively. Computation of $S(v, i, i)$ where $v \neq v_i$ is shown in Figure 3.9 (W is the largest allowable wire width as a multiple of the base wire width which is normalized to 1).

$S \leftarrow \emptyset$
for $k = i..j-1$
    $S \leftarrow S \cup join\_cq\_sets(S(v, i, k), S(v, k+1, j))$
    $S \leftarrow prune\_cq\_set(S)$
$S_b(v, i, j) \leftarrow S$

Figure 3.10: Computation of $S_b(v, i, j)$ in P-Tree$_{AT}$.

Computation of $S_b(v, i, j)$ is illustrated in Figure 3.10. The algorithm visits the possible partition points k in the subsequence and takes the union of the resulting cq-sets and prunes the result per Property 4.1.

As in the min-area case we compute $S(v, i, j)$ for all $v \in V$ and fixed i, j in a four phase process. During the computation we partition the intermediate solution

```
/* Phase 1 */
LS(v_{r,n}, w) ← S_b(v_{r,n})    ∀r ∈ {1..n}, w ∈ {1..W}
for r = 1 to n
    for c = 2 to n
        for w = 1 to W
            LS(v_{r,c}, w) ← S_b(v_{r,c}) ∪
                        augment_cq_set(LS(v_{r,c-1}, w), d(v_{r,c-1}, v_{r,c}), w)
            LS(v_{r,c}, w) ← prune_cq_set(LS(v_{r,c}, w))
```

Figure 3.11: Computation of $LS(v)$ $\forall v \in V$ P-Tree$_{AT}$

sets into W disjoint subsets to ensure that wire segments between Steiner points are of uniform width.[1] For instance, $LS(v, w)$ is a cq-set rooted at v where the wire to a branching point v' must have width w and v' is constrained to be at v or to v's left in the same row. In the fourth stage of the algorithm we collect in $S(v, i, j)$ all solutions regardless of wire width. Analogously to Lemma 3.4.1, we have the following which gives the sub-solution optimality property necessary for dynamic programming.

Lemma 3.4.3 *The routing tree giving $(c, q) \in S(v, i, j)$ either has v as a branching point (i.e. $(c, q) \in S_b(v, i, j)$),* or *$\exists(c', q') \in S(v', i, j)$ where v' is one of v's neighbors in $GG(N)$, $c = c' + \alpha w\, d(v, v')$, and $q = q' - (\beta d(v, v')/w)(\alpha d(v, v')w + c')$ for some $w \in \{1..W\}$.*

Figure 3.11 gives pseudo-code for the first phase – i.e., the computation of $LS(v, w)$. The other phases follow analogously as in the P-Tree$_A$ algorithm. The others phases follow analogously. Notice that instead of taking scalar minimums, we now apply our cq-set primitives and we now must deal with the parameterization of the sets by w.

Thus, we have shown how to compute cq-sets $S_b(v, i, j)$ and $S(v, i, j)$ for all $v \in V$ and $i \leq j$. Once this is complete, we must consider the cq-set seen at the

[1]The algorithm can be structured to allow multiple widths per wire-segment. However, this is at the expense of higher computational complexity – such fine-tuning might be more appropriate at a later stage on a fixed topology.

driver position v_d. To obtain the final area/delay tradeoff set S_{final} we augment each pair $(c,q) \in S(v_d, 1, n-1)$ by considering the resistance of the driver (much in the same way as we did with an augmenting wire) as follows.

$$S_{final} \leftarrow \{(c, q - r_d \cdot c) | (c,q) \in S(v_d, 1, n-1)\}$$
$$S_{final} \leftarrow prune\_cq\_set(S_{final})$$

Finally, we can show the optimality of the solutions derived by the P-Tree$_{AT}$ algorithm analogously to Lemma 3.4.2.

Lemma 3.4.4 *Consider $(c,q) \in S_{\text{final}}$ computed by the P-Tree$_{AT}$ algorithm. There exists no Rectilinear Steiner tree induced by sink permutation π with Steiner points on the Hanan Grid which yields required-time $q' \geq q$ and capacitance $c' < c$ (recall that load is proportional to area).*

Before proceeding, we would like to point out one of the properties of P-Tree$_A$. It is often the case that there are many distinct topologies induced by a permutation all of which achieve min area. By incorporating the required-time q with the area (and capacitance) in P-Tree$_A$ we can derive the min-delay topology among the min-area topologies without increasing run-time. This is done by using required-time q as the tie-breaker when multiple solutions derive identical area.

3.4.4 Complexity

As previously noted, the P-Tree$_A$ algorithm is dominated by the computation of the S_b sets and has $O(n^5)$ complexity. The complexity of P-Tree$_{AT}$ however, depends on the size of the cq-sets it manipulates reflected in the next lemma.

Lemma 3.4.5 *Algorithm P-Tree$_{AT}$ executes in time $O(\max(Wn^4m, n^5m)) = O(n^5m)$ where m is a bound on the maximum size of any cq-set and W is the number of allowed wire-widths (W assumed to be $O(n)$).*

It can be argued that m is in fact polynomially bounded if the possible capacitive are polynomially bounded integers or can be mapped to such with sufficient precision. This yields a *pseudo-polynomial* complexity [14].

The complexity of the P-Tree$_{AT}$ algorithm can be further formalized when the following assumptions are made.

(1) The pin positions are drawn from a *uniform* l x l grid.

(2) Let c_A be the cost (or total capacitance) of the min-area P-Tree solution. There exists a constant γ, such that solutions with cost greater than γc_A are considered infeasible.

The first assumption does not constrain the problem since l can be made arbitrarily large. It does, however, allow a complexity analysis in terms of l which, in turn, gives some insight into the tradeoff between precision and run-time – i.e., if a problem instance on an lxl is mapped to a coarser l'xl' grid, what is the impact on run-time? The second assumption allows a tighter bound on the run-time, and seems quite reasonable in practice.

We first bound the size of the routing trees in terms of the basic grid length in the following lemma.

Lemma 3.4.6 *Any any set of n points drawn from an l x l grid has a rectilinear Steiner tree of length $O(l\sqrt{n})$.*

Proof: Uniformly divide the lxl grid into n sub-grids of size $(l/\sqrt{n})$x$(l/\sqrt{n})$ or less. Let each point connect to the nearest boundary of the subgrid in which it lies; this connection has length $O(l/\sqrt{n})$ and thus, for all n points the cost is $O(l\sqrt{n})$. Clearly, the n resulting boundary points can be connected with $O(l\sqrt{n})$ additional cost since only segments on the $\sqrt{n}$ boundary rows (columns) need be considered.

□

This is by no means a new observation (in fact, it can be shown that $O(l\sqrt{n})$ is the expected length of the MST – see Chapter 5 of [25] for related information). Since the RMST is within a constant factor of the RSMT, it follows that the topology produced by P-Tree$_A$ is $O(l\sqrt{n})$ in length.

From this analysis (and assumption 2), we can conclude that only trees of total length $O(\gamma l\sqrt{n}) = O(l\sqrt{n})$ are of interest and that this also bounds the number of *distinct* capacitive values c that must be considered for any cq-set. The overall run-time follows directly from this discussion and is given by the following theorem.

Theorem 3.4.1 *Subject to assumption 2 above, P-Tree$_{AT}$ has run-time complexity $O(n^{5.5}l)$ or $O(n^{6.5})$ if $l = O(n)$. Space complexity is $O(n^{4.5}l)$ or $O(n^{5.5})$ if $l = O(n)$.*

3.4.5 Heuristic Limitation of cq-set Size

As is clear from the preceding complexity analysis, cq-sets can grow to be quite large depending on the grid granularity, number of pins and number of wire widths. To combat this we have implemented a simple heuristic to limit cq-sets to be no larger than a pre-specified size. Given a maximum allowable cq-set size of m and a cq-set S with $|S| > m$ we first select $m/2$ entries in S uniformly by their ordering (always including the first entry) and select the remaining $m/2$ by their "marginal benefit"; i.e., the marginal benefit of i'th entry $(c_i, q_i) \in S$ is $(q_i - q_{i-1})/(c_i - c_{i-1})$. This strategy has proved reasonably effective in improving run-time while maintaining high solution quality.

3.5 Experiments

Our main experimental results appear in Table 3.2. Technology parameters are the same as those used in previous works such as [19] and [3] and appear in Table **??**. For each net-size and technology pair 25 nets were routed (with terminal positions generated randomly and uniformly). We first ran the SERT and A-Tree algorithms and their wire-sizing variants on the nets and then used the delay figure achieved by these algorithms as timing requirements for the P-Tree$_{AT}$ algorithm. From this the min-area solutions produced by P-Tree$_{AT}$ achieving the same delay (or better) as SERT and A-Tree were derived.

tech.	MCM	$1.2\mu m$ CMOS	$0.5\mu m$ CMOS
driver res.	25.0Ω	212.1Ω	270.0Ω
wire res.	$0.008\Omega/\mu m$	$0.073\Omega/\mu m$	$0.122\Omega/\mu m$
wire cap.	$0.06fF/\mu m$	$0.083fF/\mu m$	$0.039fF/\mu m$
sink cap.	$1000fF$	$7.06fF$	$1.0fF$

Table 3.1: Technology parameters for experiments.

As can be seen in both the unsized and wire sized cases, SERT consumes substantially more area than a P-Tree solution with equal or better timing. For instance, for 12-pin nets in MCM technology without wire sizing, the SERT solutions used 66% more area on average than the P-Tree solution. In some instances, SERT consumes more than twice the area of the P-Tree solution. A-Tree solutions tend to produce smaller area solutions than SERT (delay figures appear to depend on the technology and net size – sometimes A-Tree out-performs SERT in delay also and vice-versa). As can be seen in the table, A-Tree consumes as much as 17% more area on average for unsized 12-pin nets in $1.2\mu m$ technology.

In the wire sized cases the comparison is with the dynamically wire sized variant SERT and the statically wire-sized variant of A-Tree described in [19]. We again see the substantial overhead of the SERT algorithm and significant overhead of A-Tree with SERT consuming more than 70% additional area for 12-pin $0.5\mu m$ nets and nearly 20% area overhead for A-Tree on those same nets.[2] We note that in the 12-pin cases with wire sizing (and only these cases), the heuristic described in the previous section to limit the size of the cq-sets with $m \approx 50$ was applied.

Another way of evaluating the quality of a routing topology is the AT metric – i.e., the product of area and delay. Results under this metric appear Table 3.3. Results are normalized to the P-Tree$_{AT}$ solution with minimum area-delay product.

[2]In two of the 12-pin $1.2\mu m$ trials and three of the $0.5\mu m$ trials, the min-delay P-Tree$_{AT}$ solution was beaten by A-Tree. In these cases the difference in delay was less than 3% percent on average and never greater than 7%; area overhead was comparable to that in the table. We suspect that this is an artifact of the heuristic used for limiting cq-sets used in these cases. The figures in the table reflect the nets for which P-Tree$_{AT}$ equaled or bettered A-Tree.

		MCM		$1.2\mu m$		$0.5\mu m$	
		SERT	A-Tree	SERT	A-Tree	SERT	A-Tree
w/o	$n = 6$	1.672	1.055	1.353	1.122	1.381	1.076
wire	$n = 9$	1.406	1.060	1.608	1.160	1.533	1.081
sizing	$n = 12$	1.660	1.083	1.627	1.170	1.657	1.158
with	$n = 6$	1.289	1.116	1.461	1.171	1.638	1.171
wire	$n = 9$	1.300	1.105	1.570	1.193	1.700	1.184
sizing	$n = 12$	1.362	1.053	1.566	1.192	1.727	1.199

Table 3.2: Average area overhead vs. P-Tree$_{AT}$ (area of P-Tree$_{AT}$ solution is normalized to 1.0).

		MCM		$1.2\mu m$		$0.5\mu m$	
		SERT	A-Tree	SERT	A-Tree	SERT	A-Tree
w/o	$n = 6$	1.406	1.130	1.536	1.174	1.568	1.119
wire	$n = 9$	1.695	1.161	2.008	1.231	1.751	1.163
sizing	$n = 12$	1.907	1.214	2.005	1.243	1.945	1.226
with	$n = 6$	1.527	1.368	1.780	1.259	1.865	1.293
wire	$n = 9$	1.504	1.278	1.927	1.286	1.875	1.257
sizing	$n = 12$	1.580	1.211	1.844	1.283	1.939	1.231

Table 3.3: Comparison under AT metric (avg.) (AT product of the P-Tree$_{AT}$ solution is normalized to 1.0).

Once again the P-Tree$_{AT}$ algorithm yields significant improvement.

An area/delay tradeoff curve computed by the P-Tree$_{AT}$ is illustrated in Figure 3.12; the corresponding SERT and A-Tree are also plotted (wire sizing incorporated in all cases). In this particular case, A-Tree achieved a better delay than SERT. It is interesting to see that the solution produced by SERT lies near the flat portion of the P-Tree$_{AT}$ curve while the P-Tree$_{AT}$ solution with the same delay is near the steep portion of the curve. The same phenomenon holds to a lesser degree vs. A-Tree.

The next set of experiments studied the min-delay solutions derived by P-Tree$_{AT}$ versus SERT and A-Tree. The results appear in Table 3.4. As can be seen,

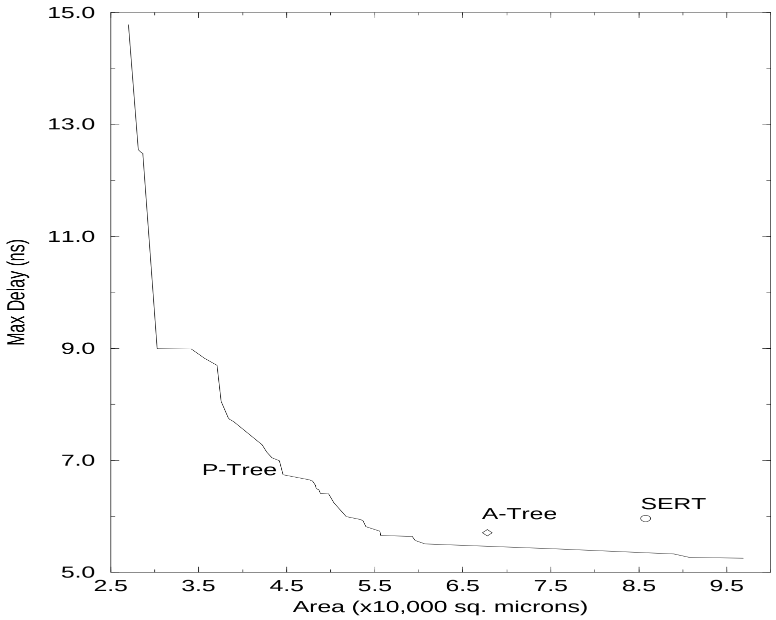


Figure 3.12: Area vs. Delay for a 12 pin MCM net (with wire-sizing)

		MCM		$1.2\mu m$		$0.5\mu m$	
		SERT	A-Tree	SERT	A-Tree	SERT	A-Tree
w/o wire sizing	$n = 6$	1.018	1.301	1.146	1.051	1.102	1.056
	$n = 9$	1.045	1.421	1.271	1.077	1.179	1.065
	$n = 12$	1.074	1.481	1.281	1.087	1.235	1.113
with wire sizing	$n = 6$	1.054	1.048	1.206	1.059	1.372	1.057
	$n = 9$	1.109	1.145	1.265	1.074	1.396	1.049
	$n = 12$	1.128	1.136	1.267	1.084	1.429	1.058

Table 3.4: Average min-delay (delay of P-Tree$_{AT}$ solution is normalized to 1.0).

significant improvements can be made when absolute minimum delay is the goal. Of course, this performance comes at a price in area overhead – i.e., we are at the flattest portion of the tradeoff curve.

As another illustration, Figure 3.13 shows routing topologies produced for a 9-pin $0.5\mu m$ net. The Figure first shows the routing topologies produced by Batched 1-Steiner, SERT and A-Tree. The next three topologies are from the P-Tree$_{AT}$ tradeoff curve: the min area solution, the solution minimizing the AT metric and the min delay solution. All three P-Tree$_{AT}$ produce better delays than B1S, SERT and A-Tree with less area than SERT and A-Tree. It is also interesting to note that the min-area P-Tree solution matches B1S in area while being near the minimal delay and that

	MST+P-Tree$_A$	MST+2P-Tree$_A$	B1S+P-Tree$_A$	B1S+2P-Tree$_A$
Avg. Imp.	−0.418%	−0.164%	0.358%	0.380%

Table 3.5: Routing 20-pin nets for min area. Figures are average percentage improvement vs. B1S.

this solution can be derived by the P-Tree$_A$ algorithm.

Another set of topologies appears in Figure 3.14. The figures show a sampling of the tradeoff curve computed by the P-Tree algorithm for a 12-pin $0.5\mu m$ net; we begin with the minimum area topology and end with the minimum delay topology (maximum delay is the metric). The driver is indicated by a rectangle and in each case the critical sink is circled. The corresponding point in the tradeoff curve is also circled for each topology. This sequence of topologies is a good example of the flexibility of the P-Tree approach.

The next set of experiments studied minimizing area alone. The P-Tree$_A$ algorithm was applied iteratively. In other words, we take a given topology such as the MST, induce a permutation consistent with that topology and apply P-Tree$_A$ to that permutation. The process may be repeated on the new topology. The goal was to how well P-Tree$_A$ compared with the state-of-the-art Batched 1-Steiner (B1S) algorithm of Kahng and Robins [26] and to see how much improvement could be obtained by by using B1S to induce the initial permutation. The experment was on fifty trials of 20 pin nets. The results appear in Table 3.5. The first column shows that the P-Tree$_A$ algorithm when applied to a permutation derived from the Minimum Spanning Tree does not quite match the solution quality of B1S. However, when P-Tree$_A$ is applied to a permutation derived from the result in the first column, the solution quality comes much closer to B1S. Though not quite equaling B1S, these experiments confirm that MST-induced permutations are in fact of high quality. The third column shows the average area reduction achieved by applying P-Tree$_A$ to a permutation induced by the B1S topology itself and the fourth column shows the result of the next iteration. These results indicate that some improvement over B1S is possible (however, at the cost of $O(n^5)$ run time versus $O(n^3)$ for B1S). While the 0.38% average improvement appears quite small it is interesting when put in the context of the results reported by Griffith et. al. in [15]. In that paper, it was reported that for 20 pin nets, the *optimal* solution (derived through a branch-and-bound procedure) improved upon the Batched 1-Steiner solution by only about 0.46% (this can be derived from Table III of [15]). Thus, it would appear that the 0.38% improvement achieved by applying P-Tree is close to the best one could hope for. (This should be only interpreted as the trend since the experiments were on different data sets and optimal solutions were not derived for the instances in Table 3.5.)

The last experiment assessed the impact on delay by constraining topologies to Hanan grid points. We routed 9-pin nets first on the Hanan grid and then on a larger grid with $n-1$ additional rows and columns. We then assessed the minimum delay solution of each run. We performed 50 such runs. In only 17 cases was delay improved and of those cases, the largest improvement was 0.8%. The overall average improvement was 0.05%. We offer this as empirical evidence in support of the suggestion of [5] – that constraining Steiner points to the Hanan grid does not have significant impact on solution quality in performance driven routing.

Finally, we give an idea of the run-time complexity of our algorithms in practice. We implemented the P-Tree algorithms in C on a Sun SPARC 20 workstation.

Representative run-times for P-Tree$_A$ range from 0.05 cpu seconds for $n = 6$ to 1.8 cpu seconds for $n = 15$ and 49 cpu seconds for $n = 30$.

In the case of P-Tree$_{AT}$ without wire sizing, run times ranged from 0.1 cpu seconds for $n = 9$, 7.3 cpu seconds for $n = 12$ and 120 cpu seconds for $n = 20$.

When running P-Tree$_{AT}$ *with* wire sizing, run-times ranged from 0.8 cpu seconds for $n = 6$ to 70 cpu seconds for $n = 12$ and 150 cpu seconds for $n = 15$.

While our algorithms have higher computational complexity the other algorithms evaluated in this paper, we suggest that they are quite practical for the vast majority of net-sizes typically seen in VLSI systems (particularly when the distribution of those net-sizes is taken into account). In addition, we have shown that the payoff for this additional computational complexity in terms of reduced area and improved timing can be very high. Furthermore, the quoted run-times are for a relatively naive implementation of the algorithm. Substantially improved performance can be had by considering techniques which are mentioned in the conclusion of this Chapter.

3.6 Conclusions/Comments

We have introduced two algorithms, the P-Tree$_A$ and P-Tree$_{AT}$ algorithms for finding high-quality Rectilinear Steiner Trees for low area and minimizing area subject to timing constraints. The algorithms find optimal solutions in the exponentially large solution space of routing topologies induced by a previously derived permutation on the sinks of the net. We have demonstrated a technique for finding high-quality permutations which are *consistent* with a given topology such as the Minimum Spanning Tree or a previously computed Steiner Tree. Such permutations are further optimized by a dynamic programming algorithm to minimize the tour length of the permutation.

We have shown that previously proposed performance driven routing techniques produce routing topologies with substantially more routing overhead than our algorithms. We also improve over previous works in minimum delay and under the AT metric. For area minimization we are able to improve on the state-of-the art Steiner Tree heuristic by iteratively applying our algorithm.

In closing we mention a few implementation issues relating to the P-Tree algorithm which are of practical iterest. Additionally, we discuss a few natural P-Tree like problem formulations which can be solved more quickly, but have less expansive solution spaces.

First, we point out that P-Tree$_{AT}$ can be made considerably faster in practice by applying bounding techniques. A particularly promising approach sets an apriori limit on the total cost of any topology it will consider; we gradually relax this limit as we explore the tradeoff curve. By adopting such an approach and computing lower-bounds on the size of the "rest of the tree" and upper-bounds on required arrival time with respect to a particular subsolution, we can detect a vast number of sub-solutions which either exceed the current cost limit or is dominated by a previously discovered solution (i.e., which was computed for a smaller limit on total cost) but cannot be discarded by the simple pruning properties presented in this chapter. The cumulative effect of early detection and removal of such sub-solutions can be quite dramatic with respect to both run time and memory consumption. The idea of this iterative bounding process is illustrated in Figure 3.15. The figure shows the progression as we relax the cost limit. At each step, any sub-solution which can be shown to contribute to only global solutions which lie in the shaded area (as determined by bounding techniques) can be discarded.

Another implementation issue relates to memory usage. Clearly, it is not necessary to keep the solution sets $S_b(v,i,j)$ (the branching point solutions) once we have computed $S(v,i,j)$ (unconstrained solutions). Indeed, this can save some space, but it is not hard to see that we can instead store only the relavent branching point solutions ($S_b(v,i,j)$) and discard the unconstrained solutions ($S(v,i,j)$). For a particular i,j, we compute $S(v,i,j)$ for all v. Each unconstrained solution will have a corresponding branching point solution; we mark those branching point solutions in $S_b(v',i,j)$ for all v'. Once this is done we can discard $S(v,i,j)$ and the unmarked branching point solutions. In practice, the number of relevant branching point solutions is significantly smaller than the number of unconstrained solutions (i.e., many of the unconstrained solutions use precisely the same branching point solution). The approach now is to *re-compute* $S(v,i,j)$ when necessary (e.g., when computing some $S_b(v,i,k)$ where $k > j$). It is easy to see that simple loop interchanging in the computation of $S_b(v,i,k)$ yields an algorithm with the same asymptotic complexity even though we perform such re-computation.

There are several natural P-Tree like problem formulations which can be solved with essentially the same techniques as were used for P-Tree. For instance, computing the optimal tradeoff curve for a *given* abstract topology routed on the Hanan grid can be done in only $O(n^{3.5}l)$ time. When this algorithm is used on a routing graph of size $|G|$ (e.g., perhaps a finer-grain graph than the Hanan grid), we have a run-time of $O(|G|n^{1.5}l)$. Thus, it is natural to use such an algorithm to *refine* a topology produced by the P-Tree algorithm by embedding it on a finer-grained grid.

While the previous approach speeds things up by assuming a *given* topology, it is also possible to accelerate things by considering a routing graph which is asymptotically *smaller* than the Hanan grid while maintaining the full set of candidate topologies induced by the given permutation. For instance, we may consider only the pin positions themselves as candidate Steiner points; such a topology can be thought of as a permutation induced *spanning tree* (though it is important to realize that multiple links may connect to the same point, but not be electrically connected). This problem can be solved in $O(n^{4.5}l)$ time by P-Tree techniques. Note however that we do not have a regular grid as the underlying routing graph in this case; nevertheless, a naive algorithm for computing $S(v, i, j)$ which considers all possible branch points v' explicitly still yields the specified run-time (with the mere computation of $S_b(v, i, j)$ producing the same run-time). Note also that if we add some additional candidate Steiner points, the algorithm still has the same complexity provided we only add $O(n)$ additional points. For instance, we may add a uniform $\sqrt{n}$ by $\sqrt{n}$ grid to give additional flexibility.

Thus, there are a wide variety of algorithms based on the P-Tree approach which can trade off solution space coverage for speed. There are clearly many interesting hybrids of these approaches which are worth exploring as "refinement" procedures. For instance, one may first compute permutation constrained topologies with only $O(n)$ candidate Steiner points. From the resulting tradeoff curve, it is natural to extract one or more abstract topology and route them on a grid with finer granularity.

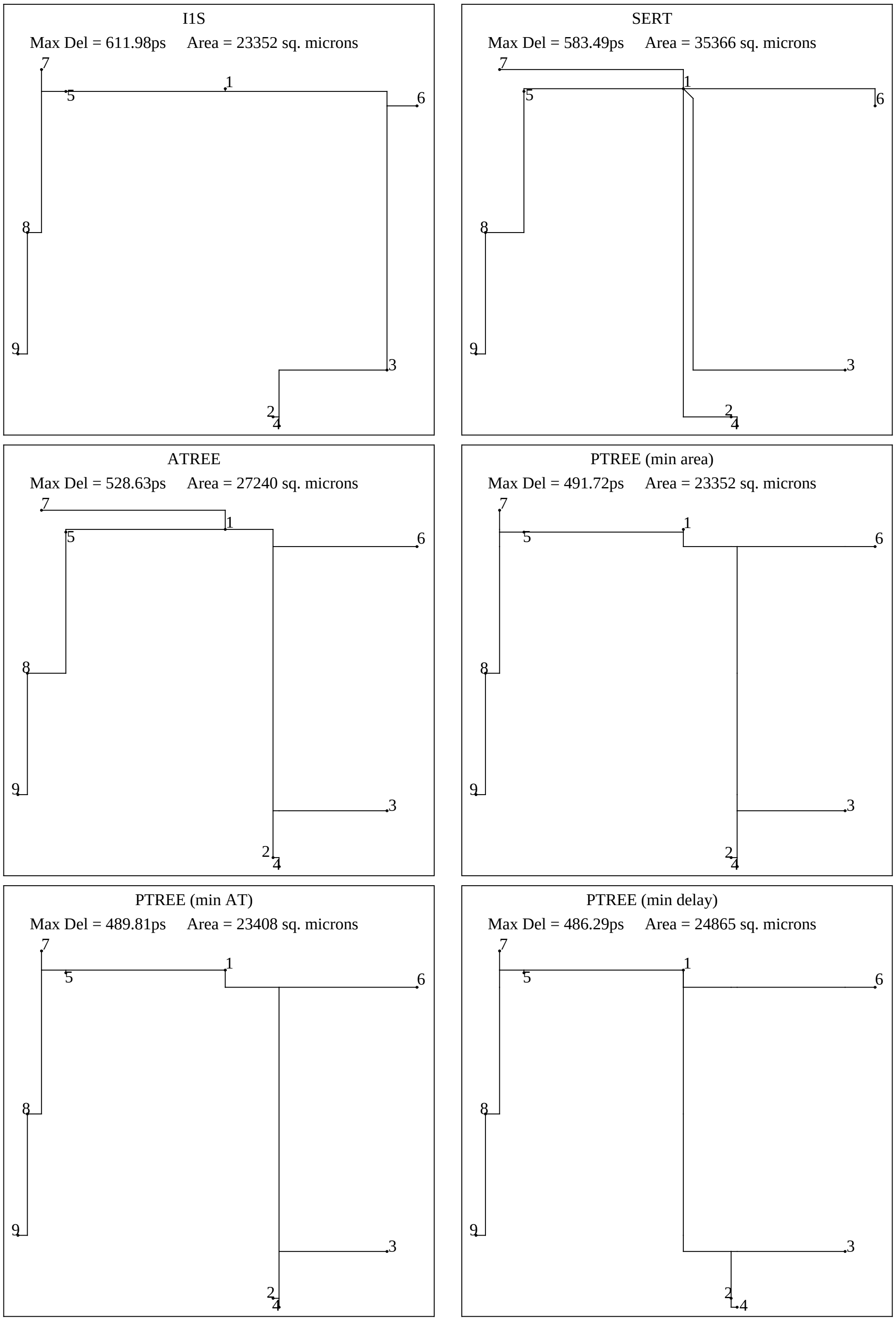


Figure 3.13: Routing Topologies for a 9-pin net in $0.5\mu m$ technology. Driver is pin 1.

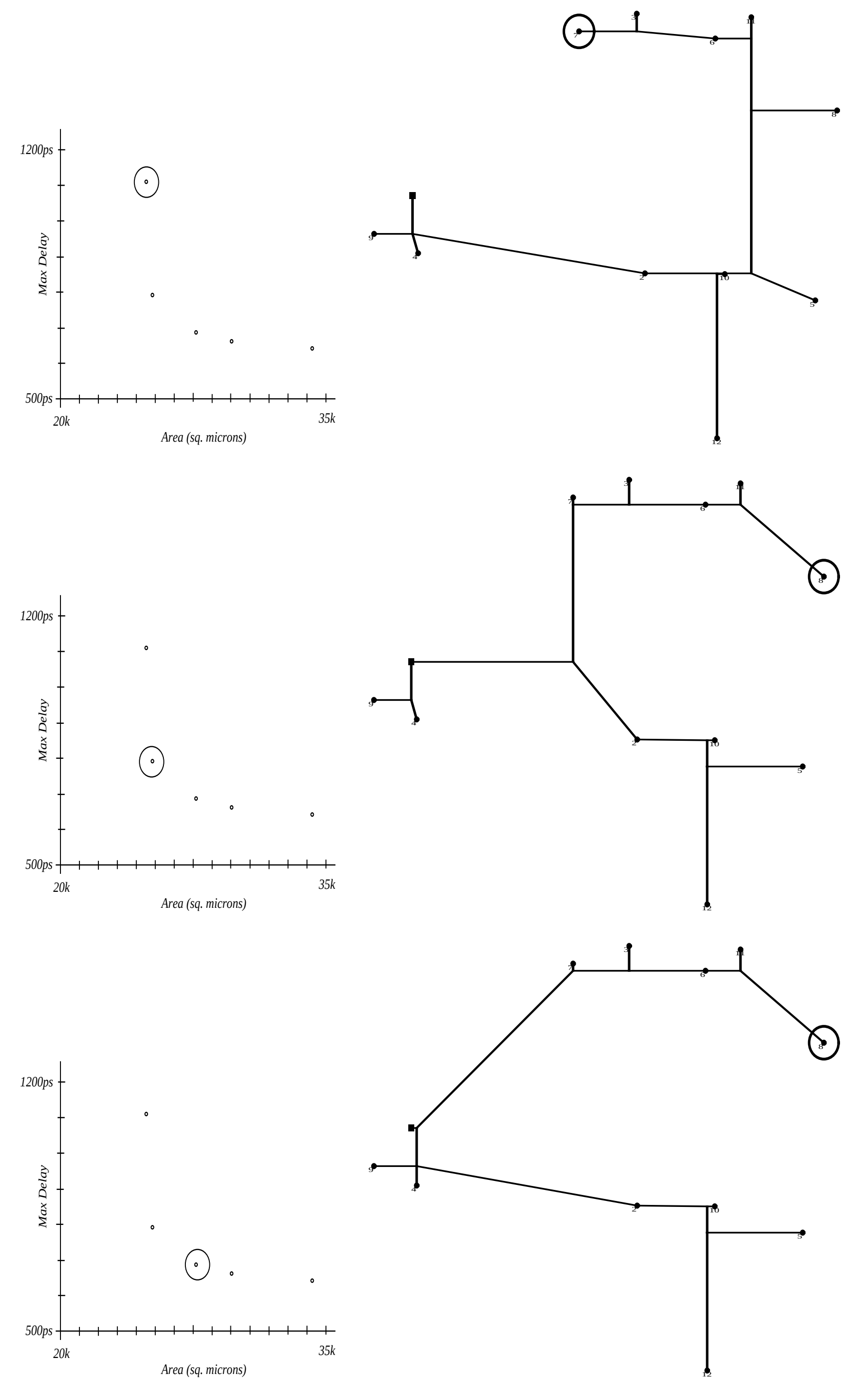


Figure 3.14: P-Tree topologies for a 12-pin net in $0.5\mu m$ technology. Driver is pin indicated by the rectangle. Corresponding point on tradeoff curve and critical sink are circled. Topologies are rectilinear: tree edges are drawn as Euclidean arcs for convenience.

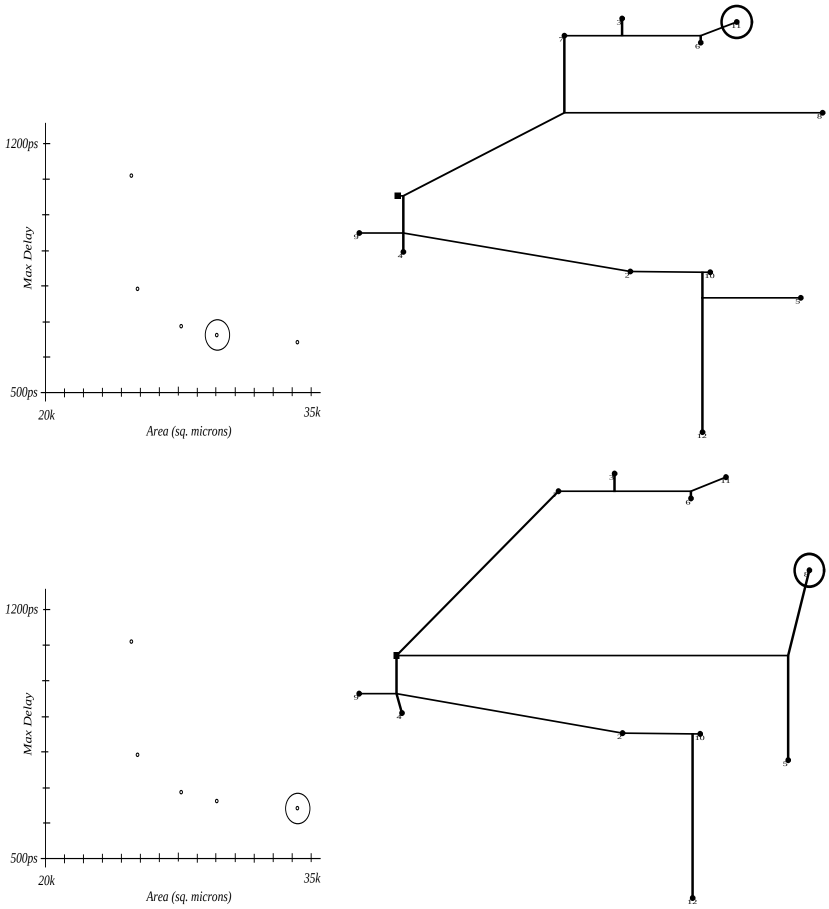


Figure 3.14: (Continued)

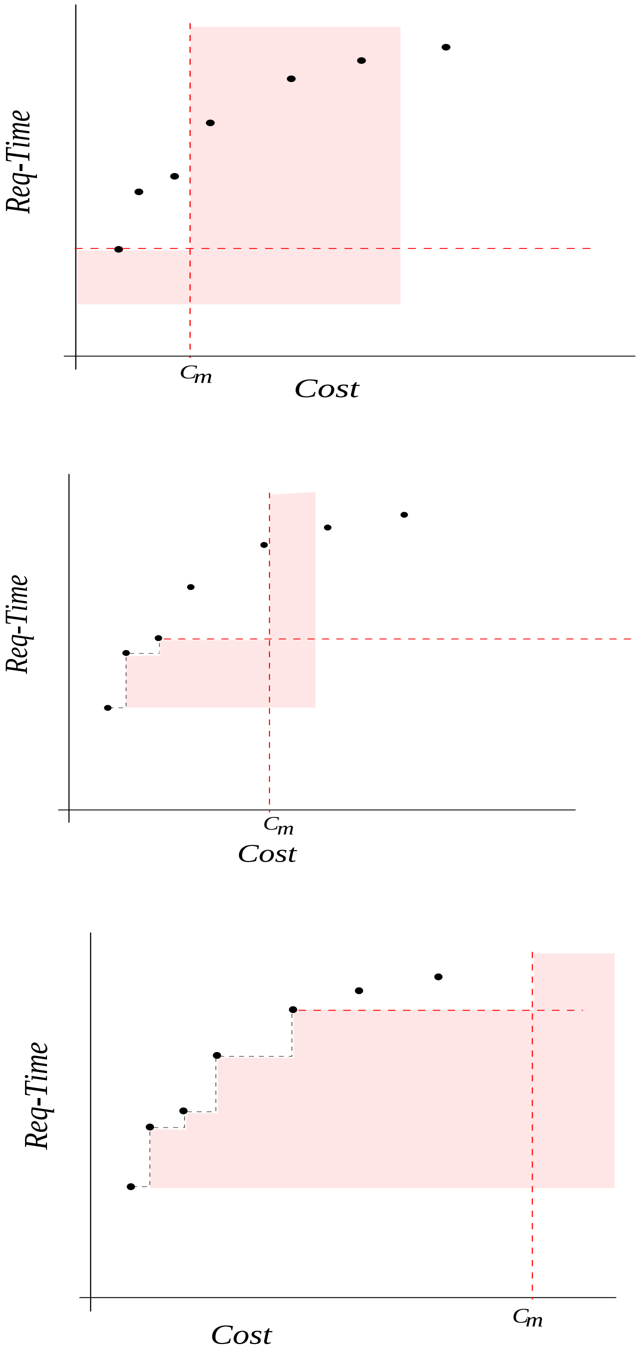


Figure 3.15: Illustration of iterative bounding process.

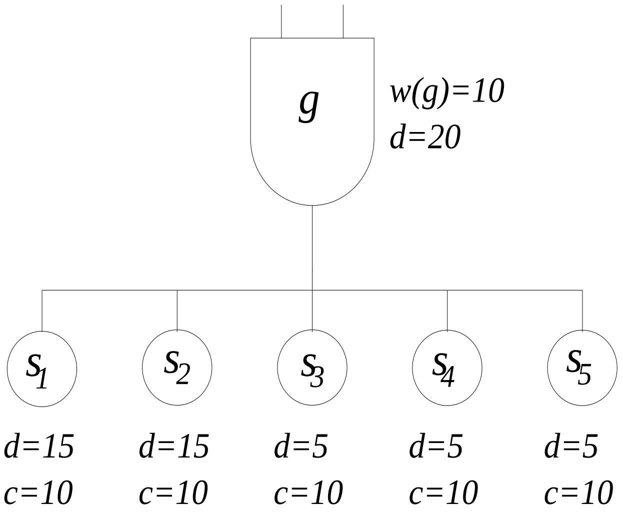
Chapter 4

Timing and Area Optimization by Gate Replication

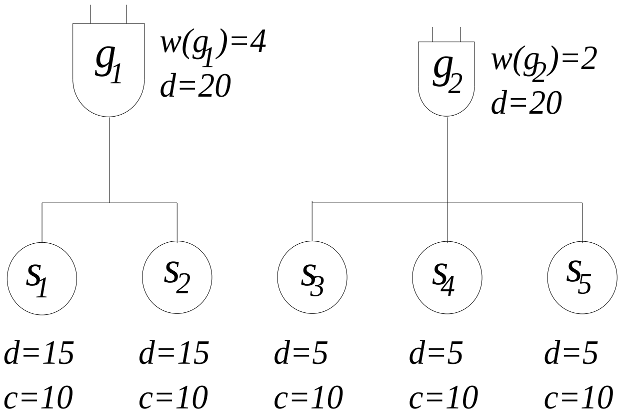
In Chapters 2 and 3 of this thesis, we presented fundamentally *net-oriented* optimization techniques; we presented algorithms for interconnect optimization by buffer insertion, wire sizing and topology synthesis. These techniques were devised largely because of technology trends (i.e., the advent of sub-micron technology) which have made interconnect delay critical. Nevertheless, *circuit-level* optimization techniques remain important and interact with net-oriented optimization. This Chapter presents problem formulations and algorithms for the largely overlooked circuit-level optimization technique of *logic replication.*

Before presenting our results in logic replication, it is instructive to consider the traditional role of *gate sizing* in the typical design flow. Gate sizing (e.g., [39]) have proven to be an effective post-synthesis, circuit-level optimization technique. A circuit can be optimized by carefully sizing the gates to minimize a cost function (e.g., area) subject to a given timing constraint. The continuous variant of the gate sizing problem can be solved optimally in polynomial time with reasonable delay models by convex optimization techniques [39]. On the other hand, the *discrete gate sizing* problem where each gate has a finite library of implementations has been shown to be NP-complete[28].

We propose that gate replication techniques can be applied after the gate sizing phase to further optimize the circuit timing and area (alternatively, iterating sizing and replication phases is another natural possibility). This contrasts with the typical design flow in which no netlist-level optimization occurs after the gate sizing phase.



(a)



(b)

Figure 4.1: Example of Logic Replication

To illustrate how gate replication can optimize a circuit, consider the net in

Figure 4.1(a). For illustrative purposes, we assume that delay is dominated by logic. We denote the width of a gate g as $w(g)$. Let the output resistance of a gate g be $1/w(g)$ and the delay of such a gate be $C/w(g)$ where C is the load on g's output. In the figure, the driving gate has width 10 and drives fanout nodes $s_1..s_5$, each of which is labeled with its input capacitance c and maximum delay d to a primary output (as is g). Intuitively, g is large because it drives a large load and some of its fanout nodes are critical (s_1 and s_2). On the other hand, the resources spent on g are, in a sense, wasted on fanout nodes s_3, s_4 and s_5 – i.e., those paths are made faster than is necessary. Figure 4.1(b) shows an alternative implementation of the net which uses two functionally identical gates g_1 and g_2 in place of g to collectively drive the fanout nodes. By introducing this redundant logic, we are able to isolate the more critical nodes, thereby eliminating the effect of the load presented by the non-critical nodes. This results in a solution with the same timing characteristics as before, but with 4 fewer units of area. Similarly, we can achieve a delay of approximately 17.62 time units by setting $w(g_1) = 7.62$ and $w(g_2) = 2.38$. Such a solution has the same cost as the original solution with improved delay. Note that since $w(g_1) + w(g_2) \leq w(g)$, we ensure that the load presented to ancestors in the circuit is no greater than before, ensuring that we do not negatively influence other paths.

The focus of this Chapter is algorithms to explicitly exploit the phenomenon illustrated in the preceding example. We note that optimization by gate replication is similar in spirit to the buffer insertion techniques presented in Chapter 2 in that both techniques *de-couple* subsets of fanouts from one another. However, gate replication is distinguished by the fact that the number of levels of logic in the design does not increase and that overall area can actually decrease. Additionally, the concept of gate replication introduces the possibility (not explored in this chapter) of *gate relocation.*

The remainder of this chapter is organized as follows: Section 4.1 presents some notational conventions and assumptions; Section 4.2 presents an unconstrained (i.e., logic-level) formulation of the problem and sketches a pseudo-polynomial time algorithm solving it; Section 4.3 gives a layout constrained formulation and sketches an algorithm for it; Section 4.4 discusses the interaction of these techniques with gate sizing; we conclude in Section 4.5.

4.1 Notation

Before giving problem formulations and algorithms, we present some notational conventions:

$w(g)$	width of gate g
$d(g), d(s)$	maximum delay from components g and s respectively to a primary output.
$d_{max}(S)$	$\max_{s \in S}(d(s))$
$s_{crit} \in S$	$\arg\max_{s \in S}(d(s))$
$C(S)$	input capacitance of fanout node s.
$c(S)$	$\sum_{s \in S} c(s)$

For simplicity, we assume that the output resistance of a gate g is proportional to $1/w(g)$ and that the delay through a gate is proportional to the product of its output resistance and the load. However, the algorithms do not rely on these models; any monotonically decreasing function of $w(g)$ is acceptable for output resistance and any monotonically increasing function of load capacitance is acceptable for gate delay; a constant term for intrinsic delay is easily incorporated. Also, for notational simplicity we assume that the minimum size gate has width 1. Additionally, the impact of additional routing overhead is not discussed, but may be taken into account for the layout constrained formulation of the problem.

4.2 Unconstrained Formulation

We now present the formulation of a combinatorial problem capturing the optimization of the example of the previous section. For this formulation we assume that $w(g)$ is an integer multiple of the basic gate width of 1. This ensures that we may efficiently examine all possible ways of splitting the available resources between

g_1 and g_2.

Formulation 4.2.1 Unconstrained 2-Way Fanout Partitioning.
Given: *driving gate* g *with width* $w(g) = k$, *fanout set* $S = \{s_1, ...s_n\}$ *and target delay* d.
Output: *If such a solution exists, return* g_1 *and* g_2 *where* $w(g_1), w(g_2) \in \{1..k\}$ *and* $w(g_1) + w(g_2) \leq k$ *and disjoint sets* $S_1, S_2 \subset S$ *s.t.* $S_1 \cup S_2 = S$ *and the resulting delay* $\max(d(g_1), d(g_2))$ *is no greater than target delay* d *when* g_1 *and* g_2 *drive* S_1 *and* S_2 *respectively. Mathematically, we have the constraint*

$$\max(d_{max}(S_1) + \frac{c(S_1)}{w(g_1)}, d_{max}(S_2) + \frac{c(S_2)}{w(g_2)}) \leq d.$$

A solution to Formulation 4.2.1 will provide the basis for an algorithm finding the minimum delay solution by performing binary search on d until the minimum feasible d is found with acceptable precision.

Alternatively, the technique we propose to solve Formulation 4.2.1 can also find the minimum area solution satisfying a target delay (for instance, the delay $d(g)$ in the original sized circuit).

To solve Formulation 4.2.1, it is useful to consider the following simpler problem.

Formulation 4.2.2 *Suppose we are given the values* $w(g_1), w(g_2)$ *and the two fanout nodes* s_{crit} *and* s', *find* S_1 *and* S_2 *satisfying target delay* d *subject to the following constraints:*

$$s_{crit} \in S_1$$

$$s \in S_2$$

$$s' \in S_2 \Rightarrow d(s') \leq d(s).$$

In other words, we assume that s_{crit} is driven by g_1 and that s a critical fanout for g_2.

A solution to Formulation 4.2.2 leads to a solution to Formulation 4.2.1 by trying the different candidates for $w(g_1), w(g_2)$ (of which there are $k-1$) paired with the different candidates for s.

It can be shown by reduction from *Subset Sum* that Formulation 4.2.2 is NP-hard (as is Formulation 4.2.1). However, *Subset Sum* (and the related *Knapsack problem*) have not been shown to be *strongly* NP-hard; in fact, pseudo-polynomial [14] dynamic programming algorithms have been devised for these problems (see, for example [34]) and such an algorithm can be used directly to solve Formulation 4.2.2. This can be seen by noticing that once s (and s_{crit}) are known, the "capacitance budgets" for S_1 and S_2 are described by the following inequalities

$$c(S_1) \leq d \cdot w(g_1) - w(g_1)d(s_{crit})$$

$$c(S_2) \leq d \cdot w(g_2) - w(g_2)d(s).$$

We state S_1 as $S_1 = S_1' \cup S_1''$ where $S_1'' = \{s' | d(s') > d(s)\} \cup \{s_{crit}\}$. Since S_1 determines S_2 (and vice-versa), it is sufficient to find S_1' (where $s \notin S_1'$) maximizing

$$c(S_1') \leq w(g_1) - w(g_1) \cdot d(s_{crit}) - c(S_1'').$$

This is a variant of the Knapsack problem and can be solved in pseudo-polynomial time (i.e., the run-time is polynomial in the load values of S when represented as integers with sufficient precision).

The algorithm described thus far assumes that gate g is replaced by exactly two gates functionally equivalent to g. If the number of new gates is unknown, the problem begins to resemble Bin-Packing for which there are no known pseudo-polynomial algorithms. On the other hand, there are very effective, provably good, approximation schemes for Bin-Packing which could conceivably be adopted here.

4.3 Layout Constrained Formulation

A criticism of the formulation of the preceding section is that it is under-constrained when we take into account the physical placement of the driving gate and the fanout nodes. For reasons of routing overhead and congestion, it is desirable to ensure that the new fanout sets obey some spatial locality constraints so that the new nets are not competing for routing resources to distribute the same signal. We propose that a natural and reasonably general way to introduce such a constraint is to impose an ordering on the fanout nodes S. Fanout sets driven by the same gate will be constrained to be drawn from the subsequences of the ordering. This leads us to our final problem formulation.

Formulation 4.3.1 Layout Constrained Multi-way Gate Replication.
Given S, g and delay target d as before, an ordering π on S, d, partition π into contiguous subsequences covering S such that the total cost of the replicated gates driving the subsequences does not exceed $w(g)$ and the resulting delay is not greater than d.

Clearly, the quality of a solution to Formulation 4.3.1 depends on the permutation π as was the case with the P-Tree algorithm of Chapter 3. Thus, it is natural to apply the same techniques for finding sink permutations based on Minimum Spanning Tree decomposition and tour length optimization.

Once the ordering is established, we can efficiently solve Formulation 4.3.1 as a shortest path problem. First note that there are only $O(n^2)$ subsequences we need to consider. Each such subsequence is represented by its beginning and ending indices, (i, j), and is referred to as S_{ij}. We can compute the minimum width gate required to drive S_{ij} as

$$w(g_{ij}) = \max(1, \frac{c(S_{ij})}{d - d_{max}(S_{ij})}) \tag{4.1}$$

where 1 is the minimum gate width. (This is for the continuous case; in the discrete case, we would take the ceiling of the above equation).

We now construct a directed weighted graph G on nodes labeled $1..n+1$. For each pair of nodes $i, j \in \{1..n\}, i \leq j$, we introduce an edge $(i, j+1)$ having weight $w(g_{ij})$. The following lemma follows directly from this graph construction.

Lemma 4.3.1 *There exists a path from node* 1 *to node* $n+1$ *of total weight not greater than* $w(g)$ *if and only if there exists a multi-way partitioning of* π *achieving the specified delay with budget* $w(g)$*.*

Thus, by solving a shortest path problem in a directed acyclic graph on n nodes with $O(n^2)$ edges, we determine the feasibility of target delay d. The edges used in the shortest path induce the partition of the fanouts. The shortest path computation can be done in $O(n^2)$ time by topological ordering since the graph is acyclic. Again we use standard binary search techniques to find the minimum d using the shortest paths algorithm as a subroutine.

An interesting feature of this algorithm is that it is entirely independent of the delay model. In particular, the algorithm does not rely on equation 4.1 and one can use virtually any method to determine the gate size $w(g_{ij})$ necessary to drive sinks $i..j$. This analysis may include a preliminary routing or even some level of simulation.

4.4 Interaction with Gate Sizing

The algorithms we have described thus far focus on single gates. In this sense, they perform a kind of "peep-hole" optimization. However, for an entire combinational circuit they may be used by applying the algorithms to individual nets in bottom up order. Once this process is complete, we may return to the gate sizer to resize the new netlist to either reduce the area required to meet the timing spec, or to achieve a previously unachievable timing spec.

The peep-hole optimization of a net driven by gate g in the initial circuit may either minimize delay subject to total area not exceeding $w(g)$ or minimize area subject to delay not exceeding $d(g)$. It may be the case that the latter approach has advantages since it may result in a certain amount of "inertia" by reducing the load seen by $g's$ fanins. The relationship between these two approaches is a topic of worth exploring.

4.5 Conclusions and Comments

This chapter provided the theoretical foundations for systematic gate replication to improve delay and area required for a given timing requirement. These techniques complement the already popular gate sizing process.

We propose that the improvement afforded by these techniques is likely to be significant based on the fact that a large percentage of nets in typical VLSI circuits have reasonably large fanout and that the delay properties (or timing requirements) of the sinks of a net intuitively seem likely to be significantly different from one another.

As we alluded to in the early portion of this chapter, we note also that the potential of redundant logic goes beyond the techniques described here by considering gate relocation. In particular, the reader may have observed that there is no real need for copies of a gate to be adjacent to one another in the resulting layout. This observation provides some freedom in placing the newly introduced gates to further improve the timing, area and perhaps even routability characteristics of the circuit. This interaction with placement is one possible topic for further research.

Chapter 5

Conclusions and Comments

The results of this work are summarized as follows. We presented an algorithm for optimal buffer insertion and wire sizing for a given routing topology; this algorithm generalizes the previous work of van Ginneken [48] by incorporating a more accurate buffer delay model by manipulation of piece-wise linear functions and by demonstrating efficient implementation of the *min power subject to timing constraints* formulation. We then presented the *P-Tree* algorithm for synthesis of high performance, low cost rectilinear Steiner trees. Lastly, we proposed *logic replication* as a novel and potentially powerful timing optimization technique. Algorithms were presented for natural formulations of the logic replication problem.

A recurring theme throughout this thesis is the importance of realistic and practical problem formulations. For instance, performance optimization without regard to cost overhead (e.g., power or area) can often lead to impractical solutions. The techniques we have proposed explicitly consider the cost versus performance tradeoff by adopting a "min cost subject to timing constraints" formulation; more generally, both the buffer insertion/wire-sizing and P-Tree algorithms compute a suite of solutions which exhibit a cost/performance tradeoff. Similarly, incorporation of realistic delay models is critical for real world practicality. Accordingly, we have adopted RC delay models and avoided geometric models of delay.

There is currently concensus in both the CAD and semiconductor industries that interconnect and, more generally, physical design will play an increasingly critical role in realizing the performance potential of new VLSI technologies. It is our hope that this work has contributed to the algorithmic foundations necessary for realizing this potential.

Finally, we mention some generalizations of the work presented here and some important open problems. First, it should be noted that the buffer insertion techniques of Chapter 2 can be integrated with the topology synthesis techniques of Chapter 3 [31]. The complexity of this simultaneous optimization is higher than either of the seperate techniques, but remains polynomial. As noted at the end of Chapter 3, there are a number of natural “P-Tree like” algorithms which can trade some solution space coverage or precision for run time efficiency (e.g., by using a coarser routing graph). Such techniques are certainly relavent when considering simultaneous buffer insertion and topology synthesis. We also point out that work needs to be done to determine the effectiveness of gate replication in practice. Motivated by the impressive results in circuit partitioning using replication [33], we also note that gate or block replication has potential beyond timing optimzation; in particular, routability can conceivably be improved by careful replication. Both the theoretical and practical aspects of such routability oriented replication deserve more attention. Several other important problems still need attention: whole-circuit optimization by net-oriented techniques, performance driven routing of multi-source nets, integration of performance driven Steiner tree algorithms with global routing.

Bibliography

[1] C.L. Berman, J.L. Carter, 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,* pp 69-99. MIT Press, March 1989.

[2] K. D. Boese, A. B. Kahng, B. A. McCoy, G. Robins, "Near-Optimal Critical Sink Routing Tree Constructions," *IEEE Trans. on CAD,* v. 14 no. 12 (1995) pp. 1417-1436.

[3] K. D. Boese, A. B. Kahng, G. Robins, "High-Performance Routing Trees With Identified Critical Sinks," *Proc. ACM/IEEE Design Automation Conf.,* 1993, pp. 182-187.

[4] K. D. Boese, A. B. Kahng, B. A. McCoy, G. Robins, "Fidelity and Near-Optimality of Elmore-Based Routing Constructions," *Proc. Proc. IEEE Intl. Conf. Computer-Aided Design*, 1993.

[5] K. D. Boese, A. B. Kahng, B. A. McCoy, G. Robins, "Rectilinear Steiner Trees with Minimum Elmore Delay," *Proc. ACM/IEEE Design Automation Conf.,* 1994, pp. 381-386.

[6] J.J. Cong, K.S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed RC delay model," *Proc. ACM/IEEE Design Automation Conf.,* 1993 pp. 606-611

[7] J.J. Cong, K.S. Leung, "Optimal Wiresizing Under Elmore Delay Model," *IEEE Trans. on CAD,* v. 14 no. 3 (1995) pp. 321-336.

[8] J.J. Cong, C.-K. Koh, "Simultaneous driver and wire sizing for performance and power optimization" *IEEE Transactions on VLSI Systems,* Dec. v. 2 no. 4 (1994) pp 408-25

[9] T. H. Cormen, C. E. Leiserson, R. A. Rivest, "Introduction to Algorithms," 1990, MIT Press and McGraw-Hill Book Co., pp. 558-562.

[10] S. Dhar, "Delay Optimization Algorithms For Tree Models of MOS Circuits," Ph.D dissertation, Dept. of Electrical Engineering, Washington University Stevens Institute of Technology, 3 September 1987.

[11] W. E. Donath, R. J. Norman, B. K. Agrawal, S. E. Bello, S.Y. Han, J. M. Kurtzberg, P. Lowy, R. I. McMillan, "Timing Driven Placement Using Complete path Delays," *Proc. ACM/IEEE Design Automation Conf.,* 1990, pp. 84-89.

[12] S. E. Dreyfus, R. A. Wagner, "The Steiner problem in graphs," *Networks*, 1 (1972), pp. 195-208.

[13] W.C. Elmore, "The Transient Response of Damped Linear Network with particular Regard to Wideband Amplifiers," *J. Applied Physics* 19 (1948), pp 55-63.

[14] M. R. Garey, D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," *W. H. Freeman,* San Francisco (1979).

[15] J. Griffith, G. Robins, J. S. Salowe, T. Zhang, "Closing the Gap: Near-Optimal Steiner Trees in Polynomial Time," *IEEE Transactions on Computer-Aided Design,* v. 13, no. 11, Nov. 1994, pp. 1351-1365.

[16] R. Gupta, B. Tutuianu, B. Krauter, L.T. Pillage, "The Elmore delay as a bound for RC trees with gerneralized input signals," *Proc. 32nd ADM/IEEE Design Automation Conference*, June 1995. pp. 364-369.

[17] M. Hanan, "On Steiner's Problem With Rectilinear Distance," *SIAM J. Applied Math.,* 14 (1966), pp. 255-265.

[18] N. Hedenstierna and K.O. Jeppson, "CMOS Circuit Speed and Buffer Optimization," *IEEE Transactions on Computer-Aided Design,* Mar. 1987, pp. 270-281.

[19] T. D. Hodes, B. A. McCoy, G. Robins, "Dynamically-Wiresized Elmore-Based Routing Constructions," *Proc. IEEE Intl. Symp. Circuits and Systems*, 1994.

[20] X. Hong, T. Xue, E. S. Kuh, C. K. Cheng, J. Huang, "Performance-Driven Steiner Tree Algorithms for Global Routing," *Pro. ACM/IEEE Design Automation Conf.,* June 1993, pp. 177-181.

[21] F. K. Hwang, D. S. Richards, P. Winter, "The Steiner Tree Problem," Elsevier Science Publishers, (1992), pp. 213-214.

[22] F.K. Hwang, "On Steiner Minimal Trees with Rectilinear Distance," *SIAM J. Applied Math.* 30 (1976), pp. 104-114.

[23] M. A. B. Jackson, E. S. Kuh, M. Marek-Sadowska, "Timing-Driven Routing for Building Block Layout," *Proc. IEEE Intl. Symp. Circuits and Systems,* 1987, pp. 518-519.

[24] M. A. B. Jackson, E. S. Kuh, "Performance-Driven Placement of Cell-Based IC's," *Proc. ACM/IEEE Design Automation Conf.,* 1989, pp. 370-375.

[25] A. B. Kahng, G. Robins, "On Optimal Interconnections for VLSI," *Kluwer Academic Publishers,* Boston (1995).

[26] A. B. Kahng, G. Robins, "A New Class of Iterative Steiner Tree Heuristics With Good Performance," *IEEE Trans. Computer-Aided Design*, 11 (1992), pp. 893-902.

[27] L.N. Kannan, P.R. Suaris, H.-G Fang, "A Methodology and Algorithms for Post-Placement Delay Optimization," *Proc. ACM/IEEE Design Automation Conf.,* 1994, pp 327-332.

[28] W.N. Li, "Strongly NP-hard discrete gate sizing problems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Aug. 1994, pp. 1045-51.

[29] J. Lillis, C. K. Cheng, T. T. Lin, "Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model," *Proc. 1995 IEEE Intl. Conf. Computer-Aided Design*, 1995, pp. 138-143.

[30] J. Lillis, C.K. Cheng, T.T. Lin, "Optimal and Efficient Buffer Insertion and Wire Sizing," *Proc of IEEE 1995 Custom Integrated Circuits Conference* 1995, pp 259-262.

[31] J. Lillis, C.K. Cheng, T.T. Lin, "Simultaneous Routing and Buffer Insertion for High Performance Interconnect," *Proc. of the Sixth Great Lakes Symposium on VLSI,* 1996, pp 148-153.

[32] S. Lin and M. Marek-Sadowska, "A fast and Efficient Algorithm for Determining Fanout Trees in Large Networks," *Proc. of* 1*st European Design Automation Conference,* 1990, pp 539-544.

[33] L.-T. Liu, M.-T. Kuo, C.-K. Cheng, T.C. Hu, "A replication cut for two-way partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* May 1995, vol. 14, (no. 5), pp. 623-30.

[34] U. Manber, "Introduction to Algorithms: a Creative Approach," *Addison-Wesley,* Reading, Mass. (1989).

[35] L. Pileggi, "Coping with RC(L) Interconnect Design Headaches," *Proc. International Conference on Computer-Aided Design*, Santa Clara, Nov. 1995.

[36] S. Prasitjutrakul, W. J. Kubitz, "A Timing-Driven Global Router for Custom Chip Design," *Proc. IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1990, pp48-51.

[37] J. Rubinstein, P. Penfield, and M.A. Horowitz, "Signal Delay in RC Tree Networks," *IEEE Trans. on CAD* 2(3) (1983), pp 202-211.

[38] T. Sakurai, "A Unified Theory for Mixed CMOS/BiCMOS Buffer Optimization," *IEEE Journal of Solid-State Circuits,* 27(7):1014-1019, July 1992.

[39] S.S. Sapatnekar, V.B. Rao, P.M. Vaidya, and S.-M. Kang, "An exact solution to the transistor sizing problem for CMOS circuits using convex optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Nov. 1993, vol.12, (no.11), pp. 1621-34.

[40] S.S. Sapatnekar, "RC Interconnect Optimization under the Elmore Delay Model," *Proc. ACM/IEEE Design Automation Conf.,* 1994, pp. 387-391.

[41] J.C. Shah, S.S. Sapatnekar, "Wiresizing and Buffer Sizing for Power-Delay Trade-offs Using a Sensitivity Based Heuristic," Technical Report ISU-CPRE-95-SS01, Dept. Electrical and Computer Engineering, Iowa State University, Ames IA 50011.

[42] K.J. Singh, A. Sangiovanni-Vincentelli, "A Heuristic Algorithm for the Fanout Problem," *Proc of 27th ACM/IEEE Design Automation Conference,* 1990, pp 357-360.

[43] S. Sutanthavibul, E. Shragowitz, "An Adaptive Timing-Driven Layout for High Speed VLSI," *Proc. ACM/IEEE Design Automation Conf.,* 1990, pp. 90-95.

[44] A. Srivinasan, K. Chaudhary, E. S. Kuh, "RITUAL: A Performance Driven Placement Algorithm for Small-Cell ICs," *Proc. IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, Nov. 1991, pp. 48-51.

[45] Synopsys 3.1 Release Manual: Appendix B, "Static Timing Analysis."

[46] H.J. Touati, "Performance-Oriented Technology Mapping," Ph.D dissertation, Memorandum No. UCM/ERL M90/109, Dept. of Electrical Engineering and computer Science, UC Berkeley, 28 November 1990.

[47] H. Vaishnav, M. Pedram, "Routability-Driven Fanout Optimization." *Proc. ACM/IEEE Design Automation Conf.*, 1993, pp. 230-235.

[48] L.P.P.P van Ginneken, "Buffer Placement in Distributed RC-tree Networks for Minimal Elmore Delay," *Proc. International Symposium on Circuits and Systems,* 1990, pp 865-868.

[49] A. Vittal, M. Marek-Sadowska, "Minimal Delay Interconnect Design Using Alphabetic Trees," *Proc. ACM/IEEE Design Automation Conf.,* 1994, pp. 392-396.

[50] N.H.E. Weste, K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1993, pp. 231-237.

[51] F. F. Yao, "Computational Geometry," Ch. 7 in *Handbook of Theoretical Computer Science. Vol A.*, Elsevier Science Publishers, 1990.