Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model

John Lillis, Chung-Kuan Cheng, *Senior Member, IEEE*, and Ting-Ting Y. Lin, *Member, IEEE*

***Abstract*—We present efficient, optimal algorithms for timing optimization by discrete wire sizing and buffer insertion. Our algorithms are able to minimize a cost function subject to given timing constraints; we focus on minimization of dynamic power dissipation, but the algorithm is also easily adaptable to, for example, area minimization. In addition, the algorithm efficiently computes the complete, optimal power-delay trade-off curve for added design flexibility. An extension of our basic algorithm accommodates a generalized delay model which takes into account the effect of signal slew on buffer delay which can contribute substantially to overall delay. To the best of our knowledge, our approach represents the first work on buffer insertion to incorporate signal slew into the delay model while guaranteeing optimality. The effectiveness of these methods is demonstrated experimentally.**

NOMENCLATURE

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	Polarity of buffer b; $p_b = 1$ indicating b is an inverter, $p_b = 0$ otherwise.
q_v	Required arrival time of sink node v.
W	Largest possible wire width ($1 \cdots W$ are possible).
B	Buffer library.
leaves(T)	Set of leaves of tree T.

I. INTRODUCTION

TIMING optimization techniques for VLSI circuits have received much attention in recent years due to increasingly aggressive designs and the impact of technological trends such as shrinking geometries. Among these techniques are performance driven placement and routing, gate sizing, buffer insertion (often referred to as fanout optimization in pre-layout works), and wire sizing. In this work, we focus on wire sizing and buffer insertion

Wire Sizing: Automatic sizing of wire widths is an attractive technique for timing optimization in signal nets, particularly with the advent of submicron technology. The benefit of wire sizing lies in the fact that, with shrinking geometries, wire resistance is now a significant contributor to overall delay. As a result, it makes sense to tune the widths of wires to balance the trade-off between added capacitance and decreased resistance. Wire sizing can be of significant benefit for both on-chip and for inter-chip (e.g., MCM) interconnects.

Cong, Leung, Zhou, and Koh provided several studies of wire sizing in [4], [2], and [3] and demonstrated the potential of wire sizing in improving delay. In these works, the problem was formulated as the task of minimizing the weighted sum of the source-to-sink Elmore delays for a set of identified critical sinks in a given routing tree. 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 $O(n^3 r)$. Cong *et al.* also attack 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.

Later, in [12], 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. He noted that the key 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 by Cong *et al.* does not apply when the length of all wire segments is not identical. In the same paper, Sapatnekar proposed a geometric programming formulation of the maximum delay, *continuous* wire-sizing problem followed by a mapping heuristic to discretize the solution.

Later, in [8], 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

Manuscript received October 20, 1995; revised November 30, 1995. This work was supported in part by Grants from the NSF project MIP-9315794 and California MICRO program.

J. Lillis and C.-K. Cheng are with the Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114 USA.

T.-T. Y. Lin is with the Department of Electrical and Computer Engineering, University of California, San Diego, La Jolla, CA 92093-0407 USA.

Publisher Item Identifier S 0018-9200(96)02468-7.

TABLE I

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

yielded a polynomial time minimum delay wire sizing dynamic programming algorithm. However, power considerations were not explicitly taken into account.

Buffer Insertion: Research on buffer insertion includes the early works of Berman *et al.*, [1], Touati [16], and van Ginneken [17]. Other contributions in this area include [9], [14], [8], and [7]. With the exceptions of [17], [7], and more recently [8], these works have focussed on timing optimization prior to layout by imposing buffer trees on the network. 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 [7], and potential routability problems created by pre-layout buffer trees as mentioned in [1].

As a result of these practical considerations, we focus on a post-layout methodology where topological information is available. Previous work on post-layout buffer insertion includes [17], 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.

In [8], a delay-optimal algorithm for simultaneous buffer insertion and wire sizing was given. However, neither power (nor area) considerations nor signal slew were taken into account.

Contributions of This Paper: In this paper, we present efficient algorithms for wire sizing, buffer insertion, and both techniques simultaneously. Our main contributions are summarized as follows.

- We give optimal, polynomial-time algorithms for the min power wire sizing problem and the simultaneous wire sizing, buffer insertion problem. This includes computation of the entire power-delay curve and a novel data-structure for efficiently pruning suboptimal solutions.
- We incorporate signal slew into the buffer delay model by manipulation of piecewise 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., [6]) 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 Table I.

In the table, n is the number of sinks in the net, m is the number of sizeable wire segments, 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 are pseudopolynomial. 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—usually a constant factor slower.

To the best of our knowledge, this work represents the most efficient optimal algorithms to date for these problems. We also improve on the results of [8] in terms of run-time when minimizing maximum delay independent of power is the goal.

Our 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 [17] 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 suboptimal solutions and thereby drastically reduce the size of solution sets. This property is identified efficiently by use of a novel tree data-structure. Generalizations of these techniques are developed to handle the case where slew is taken into account.

The remainder of the paper is organized as follows. Section II gives delay models and problem formulations. Section III gives the overall algorithmic framework. Section IV addresses the min-delay/max-required time formulation. Section V generalizes the algorithm to minimize power subject to timing constraints. Section VI generalizes the algorithm further to account for the contribution of signal slew to delay by the manipulation of piecewise linear functions. Section VII gives experimental results and we conclude in Section VIII.

II. Models and Problem Formulation

A. Delay Models

As in previous works, we adopt the Elmore delay model [5] for interconnect delay and standard RC models for buffer delay.

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. The computation of these delays is detailed in the following.

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

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

where α and β are characteristic constants.[1]

To compute the Elmore delay of a wire e_v in tree T, we first recursively define $c(T_v)$, the total lumped capacitance of T_v, as follows:

$$c(T_v) = \begin{cases} c_v, & \text{if } v \text{ is a sink node} \\ c_b, & \text{else if buffer } b \text{ placed at } v \\ c(T_{l(v)}) + c(T_{r(v)}) & \\ \quad + c_{e_{l(v)}} + c_{e_{r(v)}}, & \text{otherwise.} \end{cases} \tag{1}$$

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 subtrees, $c_{e_{l(v)}}$ and $c_{e_{r(v)}}$. Given this notation, the Elmore delay of wire e_v, is defined as

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

Similarly, the delay through a buffer b at node v in a basic model is determined by the parameters $c(T_v)$, b's *intrinsic* (load independent) *delay* d_b and *output resistance* r_b. The delay through the buffer 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 well-known *isolation property* of buffers exhibited in (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 by the buffer.

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, $D_{L\text{prev}}$—i.e., the RC delay of the driving buffer. Thus we denote the augmented delay equation as

$$\text{buf\_delay}_{\text{slew}}(b, c_l) = \text{buf\_delay}(b, c_l) + \lambda_b D_{L\text{prev}}. \tag{2}$$

This and similar models have been proposed in various contexts (e.g., [15], [7]). An extension of our algorithms to accommodate this delay model is discussed in Section VI.

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

B. 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 is particularly useful since it allows a straightforward application of our algorithms to optimize a combinational network by proceeding in bottom-up order. 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 nonnegative, the tree T is said to meet its timing requirements. 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 $\max \text{delay} = -q(T)$.

C. Minimizing Power Subject to Timing Requirements

The dynamic power dissipation for CMOS technology P_d is given in [18] as

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

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.

If we let 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 trade-off 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 assumption that design techniques have been employed to eliminate or drastically reduce short-circuit power dissipation.

III. 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 pseudocode in Fig. 1. The algorithm computes the solution sets $S_{\text{bot}}(v)$ and $S_{\text{top}}(v)$. The set $S_{\text{bot}}(v)$ 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 $S_{\text{top}}(v)$ 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.

Algorithm GDP(T,B,W)

Foreach node $v \in T$ in topological order from leaves to root
  If v is a leaf
    Compute $S_{\text{bot}}(v) \leftarrow$ *Base_Case*(v)
  Else
    Compute $S_{\text{bot}}(v) \leftarrow$ *Bottom_Solutions*$(v, S_{\text{top}}(l(v)), S_{\text{top}}(r(v)))$
  If v is not the root
    Compute $S_{\text{top}}(v) \leftarrow$ *Top_Solutions*$(v, S_{\text{bot}}(v))$
  Else /* v is the root */
    Compute *Optimal_Soln*$(v, S_{\text{bot}}(v))$

Fig. 1. General algorithm structure.

The four procedures *Base_Case( )*, *Bot_Solutions( )*, *Top_Solutions( )*, and *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 (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 detail the simplest scenario: maximizing the required arrival $q(T)$ time 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 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.

IV. 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 $S_{\text{bot}}(v)$ and $S_{\text{top}}(v)$ are each partitioned into two disjoint subsets as

$$S_{\text{bot}}(v) = S^{+}_{\text{bot}}(v) \cup S^{-}_{\text{bot}}(v)$$
$$S_{\text{top}}(v) = S^{+}_{\text{top}}(v) \cup S^{-}_{\text{top}}(v).$$

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 *noninverted* and the sets with superscript "−" contain solutions where the incoming signal *is* assumed to be inverted.

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

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

An important initial observation made by van Ginneken [17] is the following.

Property 4.1: For $(c, q), (c', q') \in S$, if $c' \geq c$ and $q' < q$ then (c', q) is suboptimal.

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

Property 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 to compute *Base_Case*(v) is simply sets $S^{+}_{\text{bot}}(v) \leftarrow (c_v, q_v)$ and $S^{-}_{\text{bot}}(v) \leftarrow \emptyset$ since no valid solution will have an inverted signal at a sink.

Computation of *Bottom_Solutions( )* is described in pseudocode in Fig. 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 time and ensures that Property 4.2 holds. The key is that this is a linear time operation and the size of the resulting set is linear as observed by van Ginneken [17]. Next, we find the optimal buffer configurations in lines 15–20 by pairing buffers 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 Fig. 3. We examine all pairings of widths w for wire e_v (having length l_{e_v}) with solutions (c, q) at v. Since the loads c' are of the form $c + wZ$ where Z is fixed and $w \in \{1 \cdots W\}$, 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 4.2 to ensure that the set is in strictly increasing order of both c and q.

Finally, we must implement *Optimal_Soln*$(v, S_{\text{bot}}(v))$. This is done simply by pairing all of the previously computed *nonbuffered* $(c, q) \in S^{+}_{\text{bot}}(v))$ (since the signal leaving the driver is noninverted) with the properties (i.e., output resistance) of the driver and selecting the solution with the largest resulting required arrival time.

Algorithm: *Bottom_Solutions*$(v, S_{top}(l(v)), S_{top}(r(v)))$

```
/* First compute unbuffered solutions */
S+bot(v) ← ∅
Let Sl = S+top(l(v))
Let Sr = S+top(r(v))
/* Sl, Sr are indexed and ordered by c */
i ← 1 ; j ← 1
While (i ≤ |Sl| and j ≤ |Sr|)
    Let (cl, ql) = Sl[i]
    Let (cr, qr) = Sr[j]
    S+bot(v) ← S+bot(v) ∪ {(cl + cr, min(ql, qr))}
    If (ql ≤ qr)    /* Left Critical */
        i ← i + 1
    If (qr ≤ ql)    /* Right Critical */
        j ← j + 1
Compute S−bot(v) analogously
/* Now compute buffered solutions */
Foreach buffer b ∈ B
    If b is an inverter
        Find (c, q) ∈ S−bot(v) s.t.
          q+b = q − db − rbc is maximized
    Else
        Find (c, q) ∈ S+bot(v) s.t.
          q+b = (q − db − rbc) is maximized
    Analogously compute q−b
S+bot(v) ← S+bot(v) ∪ {(cb, q+b)|∀b ∈ B}
S−bot(v) ← S−bot(v) ∪ {(cb, q−b)|∀b ∈ B}
Prune S+bot(v) and S−bot(v) by Property 5.2
```

Fig. 2. *Bottom_Solutions( )* routine for max required time.

Algorithm: *Top_Solutions*$(v, S_{bot}(v))$

```
S+top(v) ← ∅
Foreach (c, q) ∈ S+bot(v), w ∈ {1..W} in increasing
        order of c' = c + αwl_ev
    S+top(v) ← S+top(v) ∪ {(c', q − elmore(ev))}
    /* elmore delay evaluated at width w */
Compute S−top(v) analogously
Prune S+top(v) and S−top(v) by Property 5.2
```

Fig. 3. *Top_Solutions( )* routine for max required time.

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

Comments: For simplicity, we have presented our algorithm in terms of a binary tree, but note that the algorithm is easily applied to general trees. One straightforward method to achieve this is to convert a nonbinary 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 zero. 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 generalizable to accommodate multiple sizeable segments in a single wire 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 if desired. 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 subsequent algorithms in this paper.

A. 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 grid-width" to analyze the complexity.

Property 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 $w_i \in \{1 \cdots W\}$ 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 take on any integer value in range of $m \cdots mW$. This gives an upper-bound of mW on the sizes of the load-required_time 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 size, we have $O(n^2W^2)$ since $n = m$ in such a case.

Scenario 2) is a generalization of the situation for the algorithm of van Ginneken [17]. Since $W = 1$ in this case, computation of S_{top} sets is trivial. Thus, the size of the S_{bot} sets is the key factor in the run-time. We first state the following properties alluded to earlier.

Property 4.4: For $S_{\text{bot}}(v)$, let S_l and S_r be the S_{top} sets of v's left and right children respectively of the same polarity. The following inequality holds: $|S_{\text{bot}}(v)| \leq |S_l| + |S_r| + |B|$.

Property 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 $O(n|B|(n + |B|n)) = O(n^2|B|^2)$.

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 load-required_time 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.

V. Minimizing Power for Given Timing Constraints

We now extend the algorithm to accommodate dynamic power considerations. For clarity, we present this and subsequent algorithms without regard to signal polarity. Application of the ideas of the previous section is straightforward. We note also that the extension presented in this section can easily be modified by minimizing the area subject to timing constraints.

The first issue is how to parameterize the solution sets. Now we are not only concerned with the load and required-time of a sub-solution, but also the power it consumes.

Therefore, solution sets $S_{\text{bot}}(v)$ and $S_{\text{top}}(v)$ 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, $(p, S) \in P_{\text{bot}}(v)$ 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 for dynamic power dissipation, capacitance is the correct measure and c_v is the "power" associated with sink v, *Base_Case*(v) simply sets $S_{\text{bot}}(v) = \{(c_v, \{(c_v, q_v)\})\}$.

Pseudocode for *Bottom_Solutions( )* is given in Fig. 4. We visit all possible values of total power consumption at v. These values are from among the buffered and unbuffered configurations. We introduce the notion of a "nonbuffer" ϕ 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 expensive sorting operation.

Top_Solutions( ) is implemented in a similar manner; its pseudocode appears in Fig. 5.

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 4.2. However, an additional pruning condition is utilized in Figs. 4 and 5 on lines 9 and 6 respectively. This pruning is captured in the following property.

Property 5.1: For solution (c, q) consuming power p, if $\exists$ solution (c', q') where $p' < p$, $c' \leq c$, and $q' \geq q$, then the solution (c, q) is suboptimal.

Algorithm: *Bottom_Solutions*$(v, S_{top}(l(v)), S_{top}(r(v)))$

1. **Let** $B' = B \cup \{\phi\}$
   /* ϕ indicates "no buffer", $c_\phi = 0$ */
2. $S_{bot}(v) \leftarrow \emptyset$
3. **Foreach** triple $(p_l, S_{p_l}) \in S_{top}(l(v))$, $(p_r, S_{p_r}) \in S_{top}(r(v))$, $b \in B'$ 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 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 $(p', S_{p'}) \in S_{bot}(v)$ */
10. **If** $(p, S_p) \in S_{bot}(v)$ (previous triple gave same p)
11. $S_p \leftarrow S_p \cup S'$
12. Prune S_p by property 5.2
13. Else
14. $S_{bot}(v) \leftarrow S_{bot}(v) \cup \{(p, S')\}$

Fig. 4. *Bottom_Solutions( )* routine for low power.

Algorithm: *Top_Solutions*$(v, S_{bot}(v))$

1. $S_{top}(v) = \{(p, \emptyset) | p \text{ is a possible power}\}$
2. **Foreach** pair $w \in 1..W$, $(p_{bot}, S_{p_{bot}}) \in S_{bot}$ in increasing order of $p = p_{bot} + \alpha w l_{e_v}$
3. **Foreach** $(c, q) \in S_{p_{bot}}$
4. $S_p \leftarrow S_p \cup \{(c + \alpha w l_{e_v}, q - \text{elmore}(e_v))\}$
   /* elmore delay evaluated at width w */
5. Prune S_p by Property 5.2
6. $S_p \leftarrow S_p \setminus \{(c, q) \in S_p | \exists (c', q') \in S_{p'}, p' < p, c' \leq c, q' \geq q\}$
   /* where $(p', S_{p'}) \in S_{top}(v)$ */
7. **If** $(S_p \neq \emptyset)$
8. $S_{top}(v) \leftarrow S_{top}(v) \cup \{(p, S_p)\}$

Fig. 5. *Top_Solutions( )* routine for low power.

The application of this property has proven essential in giving reasonable running times in practice. Efficient detection of Property 5.1 is addressed subsequently.

We implement *Optimal_Soln*$(v, S_{\text{bot}}(v))$ simply by selecting the lowest power unbuffered solution at the root giving required-time $q(T) \geq 0$ when paired with the driver. Alternatively, this set of unbuffered solutions gives the full trade-off curve.

Detection of Property 5.1: When computing the load-required_time set S_p for power p in the previous algorithms, we have already computed the load-required_time sets $S_{p'}$ for all $p' < p$. We now want a data-structure to efficiently determine, for each $(c, q) \in S_p$, if Property 5.1 holds. Since the solution sets can grow to be of substantial size, a linear scan to detect this property would likely be a disaster.

Since we visit the power values in order, we know that the entries in the data structure are for power values $p' < p$. Thus, the data structure need only concern itself with c and q values.

Thus, we need a data structure which efficiently supports the following operations:

- *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.

Such a data-structure solves a special case of the *orthogonal range query* problem from computational geometry (see e.g., [19]). Our problem is a special case in the sense that we need

not retrieve or count all (c', q')'s satisfying the property, and 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 the 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-D 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 by load values c. At each node t of the search tree, we store the load value c *and* the *largest* q value at t or in the left subtree t. We refer to this value as $t.q_{l\_\max}$. Given this augmentation, *insert( )* is easily implemented recursively and *sub_opt( )* can be implemented by examining the four following cases with respect to c, q (given), and $t.c$ and $t.q_{l\_\max}$ stored at the current node in the tree (boundary conditions are not given for clarity)

$c < t.c, q < t.q_{l\_\max}$	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.

A. Run Time

With respect to wire-sizing alone, i.e., $|B| = 0$, we notice that $p = c$ for every power-load-required_time triple 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 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 IV-A, 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).

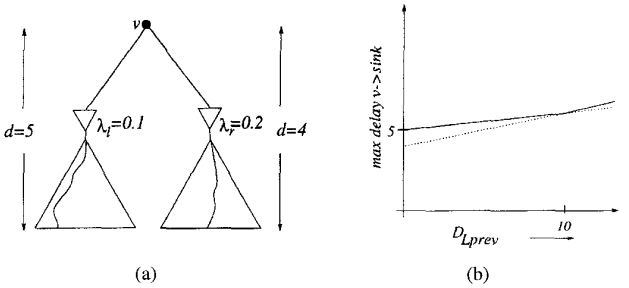


Fig. 6. Piecewise linear function modeling effect of signal slew.

VI. 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 piecewise linear functions to model the effect of signal slew.

Overview: By (2), buffer delay is augmented by the term $\lambda_b D_{L\text{prev}}$ (recall that λ_b is a characteristic constant of buffer b and that $D_{L\text{prev}}$ 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 $D_{L\text{prev}} = x$?"

Since $\lambda_b D_{L\text{prev}}$ is linear in $D_{L\text{prev}}$, we utilize *piecewise 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 piecewise linear function; $f(x) = q$ is the optimal required time q at v for load c and $D_{L\text{prev}} = x$.

We illustrate the modeling of delay by a piecewise linear function in Fig. 6. Fig. 6(b) shows the piecewise linear delay function f at node v in Fig. 6(a). The left and right subtrees have maximum delays of five and four units, respectively, when $D_{L\text{prev}} = 0$. 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 Fig. 6(b) correspond 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 $D_{L\text{prev}}$ can result in different critical paths.

We represent a piecewise 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 piecewise linear functions is based on three basic operations:

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

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, pwl_max$(f_1, \ldots, f_k)$. 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 piecewise 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 piecewise linear functions.
3) Where we eliminated suboptimal solutions (c, q) by Properties 4.2 and 5.1, we now eliminate suboptimal *portions* of solutions (c, f) by analogous properties. These suboptimal solutions now give required arrival time of $-\infty$ for certain values of $D_{L\text{prev}}$.

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

Pruning Operations: 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 $S_{1,2} = (S_1 \cup S_2) \backslash S^*$ where S^* was a set of provably suboptimal solutions by Property 4.2; i.e., $(c, q) \in S^*$ implied that $\exists (c', q') \in S_{1,2}$ s.t. $c' \leq c$ and $q' \geq q$.

We generalize this concept for *load-required_time_func* sets. The property analogous to Property 4.2 is the following:

Property 6.1: Let load-required_time set $S = \{(c_1, f_1), \cdots, (c_m, f_m)\}$ be ordered by load; i.e., $c_i < c_{i+1}$. For $(c_i, f_i) \in S$ let f_i' be defined as

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

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 Fig. 7 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 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 suboptimal and may be eliminated. The analogous property for the slew sensitive generalization is as follows.

Property 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 that $p_i < p$ and $c_i \leq c$. Further, let $f_{\max} = \text{pwl\_max}(f_i)$ over all $(p_i, c_i, f_i) \in S'$. In a manner similar to Property 6.1, we may replace f with

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

Algorithm: *Merge_Load-Func_Sets*(S_1, S_2)

```
Let S' ← S_1 ∪ S_2
f_max(x) = −∞   ∀x
Foreach (c, f) ∈ S' in increasing order of c
    Let f'(x) = f(x)    if f(x) > f_max(x)
        f'(x) = −∞      otherwise
    S' ← S' \ {(c, f)}
    If ∃x s.t. f'(x) ≠ −∞
        S' ← S' ∪ {(c, f')}
    /* Otherwise, (c, f') is useless */
    f_max ← pwl_max(f_max, f)
Return S'
```

Fig. 7. Algorithm for merging *load-func* sets (presumably consuming the same power).

Algorithm: *Bottom_Solutions*$(v, S_{\text{top}}(l(v)), S_{\text{top}}(r(v)))$

```
B' ← B ∪ {φ} where φ indicates "no buffer", c_φ = 0
S_bot(v) ← ∅
Foreach triple (p_l, S_{p_l}) ∈ S_top(l(v)), (p_r, S_{p_r}) ∈ S_top(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 7.2
        If (∃(p, S_p) ∈ S_bot)
            S_bot ← S_bot − {(p, S_p)}
            S'_p ← Merge_Load-Func_Sets(S', S_p)
            S_bot ← S_bot ∪ {(p, S'_p)}
        Else    /* New power value */
            S_bot ← S_bot ∪ {(p, S')}
```

Fig. 8. *Bottom_Solutions( )* routine for low power and signal slew.

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

A generalization of the data-structure in Section V is be presented later in this section to efficiently implement the pruning of Property 6.2.

Finally, we give pseudocode for *Bottom_Solutions( )* and *Top_Solutions( )* in Figs. 8 and 9, respectively. The routines follow the previous versions quite closely. However, because Properties 6.1 and 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 (Fig. 2, lines 7–12). This can be seen in the *for loop* of *Bottom_Solutions( )* in Fig. 8—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—and this may hold for all i's. However, in practice, these sets tend to

Algorithm: *Top_Solutions*$(v, S_{\text{bot}}(v))$

1. $S_{\text{top}}(v) \leftarrow \emptyset$
2. **Foreach** pair $w \in 1..W$, $(p_{\text{bot}}, S_{p_{\text{bot}}}) \in S_{\text{bot}}$ in increasing order of $p = p_{\text{bot}} + \alpha w l_{e_v}$
3. **Foreach** $(c, f) \in S_{p_{\text{bot}}}$
4. $S_p \leftarrow S_p \cup \{(c + \alpha w l_{e_v}, \text{pwl\_add\_scalar}(f, -\text{elmore}(e_v))\}$ (at width w; $S_p = \emptyset$ initially)
5. Prune sub-optimal solutions from S_p by Property 7.1
6. Prune sub-optimal solutions from S_p by Property 7.2
7. **If** $(S_p \neq \emptyset)$
8. $S_{\text{top}}(v) \leftarrow S_{\text{top}}(v) \cup \{(p, S_p)\}$

Fig. 9. *Top_Solutions( )* routine for low power and signal slew.

Algorithm: *Prop7.2*(t, c, f)

```
if t=NULL return f
else if c < t.c
    return Prop7.2(t.left, c, f)
else if c ≥ t.c
    Define f' as:
        f'(x) = f(x) if f(x) > t.f_{L_max}(x)
        f'(x) = -∞ otherwise
    if (f'(x) = -∞ ∀x  or  t.c = c)
        return f'
    else
        return Prop7.2(t.right, c, f')
```

Fig. 10. Implementation of Property 6.2.

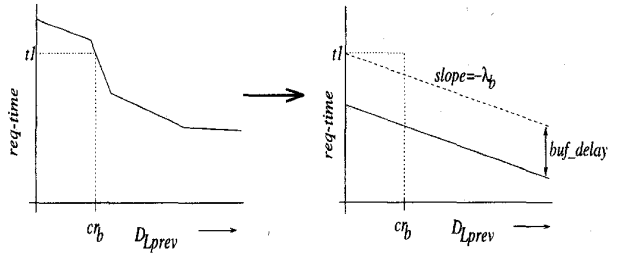


Fig. 11. Effect of inserting a buffer.

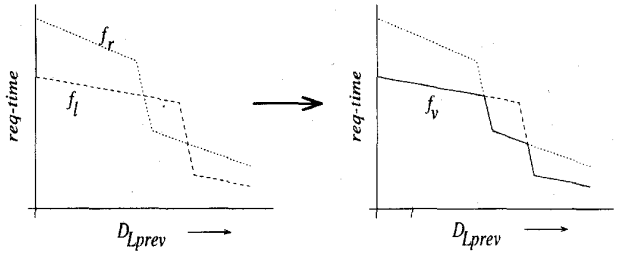


Fig. 12. Combination of left and right solutions.

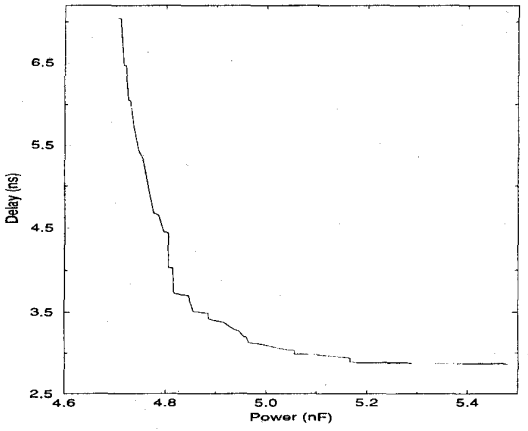


Fig. 13. Power-Delay curve for a 20 sink net.

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.

Examples: To give intuition on the operation of the algorithm, we now give some illustrations. Fig. 11 shows function f having load c before and after buffer b is considered for insertion at v as is performed in *Bottom_Solutions( )*. Since b will drive load c, $D_{L\text{prev}}$ 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 $-\lambda_b$.

Fig. 12 illustrates the operation of combining left and right ("top") solutions. For some values of $D_{L\text{prev}}$, 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 function to capture the combination of these two solutions.

Implementation of Property 6.2: We now give a generalization of the data-structure of Section V to implement Property 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 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 $(t{\cdot}c, t{\cdot}f_{l\_\max})$ at each tree node t rather than $(t{\cdot}c, t{\cdot}q_{l\_\max})$. As before, the search tree is ordered by c. The piecewise linear function $t \cdot f_{l\_\max}$ is $\text{pwl\_max}(f_i)$ over all f_i where (c_i, f_i) resides either at t $(c_i = t \cdot c)$ or in its left subtree.

Pseudocode implementing Property 6.2 is given in Fig. 10—i.e., given (c, f), we return the portions of f which are *not* suboptimal in the form of f'. Updating the data-structure can be done by recursively traversing the search tree and 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 piecewise linear operations (which has been very small in our experiments).

TABLE II
BASIC ALGORITHM VERSUS SLEW ALGORITHM FOR VARIOUS SENSITIVITIES

	K=1		K=2		K=4		K=8		K=16		K=32		MAX
n	B	S	B	S	B	S	B	S	B	S	B	S	CPU
10	1.94	1.94	1.95	1.95	1.96	1.96	1.97	1.97	2.0	2.0	2.07	2.07	0.2
15	2.64	2.64	2.65	2.65	2.66	2.66	2.87	2.70	3.19	2.77	3.90	2.87	0.8
20	2.99	2.93	3.04	2.99	3.13	3.02	3.30	3.07	3.66	3.16	4.37	3.47	1.3
25	3.59	3.59	3.60	3.60	3.62	3.62	3.67	3.67	3.94	3.78	4.70	3.95	3.8
30	3.42	3.39	3.49	3.40	3.63	3.42	3.91	3.46	4.47	3.56	5.59	3.73	6.8

VII. 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 nonuniform 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 trade-off curves, and the effect of signal slew.

We used five different buffer types; the smallest (1X) buffer having $r_b = 3170\ \Omega$, $c_b = 10$ fF, $d_b = 300$ ps 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 0.8, smallest 0.1). Our experiments used a variety of wire widths from 0.5 μm to 5 μm (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 zero).

Fig. 13 shows the optimal power versus 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 II. Here we show the importance of taking slew into account for buffer insertion. We performed experiments on nets ranging from 10 to 30 sinks. The other variable was a scaling factor K, which is a coefficient for λ_b. We replace each λ_b with $K\lambda_b$. For each K we have two columns: B is the result of running the basic algorithm for min delay and evaluating the delay of the resulting tree taking slew into account and S is the result of the extended algorithm of Section V. The right-most column is the worst run-time among all experiments in that row. As K and n grow, we see large variation between observed delays, approaching 50% in one case. For the 10 sink net, the critical path never included any buffers accounting for the identical delays.

VIII. CONCLUSION

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

REFERENCES

[1] C. L. Berman, J. L. Carter, and K. F. Day, "The fanout problem: From theory to practice," in *Advanced Research in VLSI: Proc. 1989 Decennial Caltech Conf.*, C. L. Seitz Ed., MIT Press, Mar. 1989, pp. 69–99.

[2] J. J. Cong and K. S. Leung, "Optimal wiresizing under Elmore delay model," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 3, pp. 321–336, 1995.

[3] J. J. Cong and C.-K. Koh, "Simultaneous driver and wire sizing for performance and power optimization" *IEEE Trans. VLSI Syst.*, vol. 2, no. 4, pp. 408–425, Dec. 1994.

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

[5] W. C. Elmore, "The transient response of damped linear network with particular regard to wideband amplifiers," *J. Applied Physics*, vol. 19, pp. 55–63, 1948.

[6] N. Hedenstierna and K. O. Jeppson, "CMOS circuit speed and buffer optimization," *IEEE Trans. Computer-Aided Design*, pp. 270–281, Mar. 1987.

[7] L. N. Kannan, P. R. Suaris, and H.-G. Fang, "A methodology and algorithms for post-placement delay optimization," in *Proc. ACM/IEEE Design Automation Conf.*, 1994, pp. 327–332.

[8] J. Lillis, C. K. Cheng, and T. T. Lin, "Optimal and efficient buffer insertion and wire sizing," in *Proc. Custom Integrated Circuits Conf.*, 1995, pp. 259–262.

[9] S. Lin and M. Marek-Sadowska, "A fast and efficient algorithm for determining fanout trees in large networks," in *Proc. 1st European Design Automation Conf.*, 1990, pp. 539–544.

[10] J. Rubinstein, P. Penfield, and M. A. Horowitz, "Signal delay in RC tree networks," *IEEE Trans. Computer-Aided Design*, vol. 2, no. 3, pp. 202–211, 1983.

[11] T. Sakurai, "A unified theory for mixed CMOS/BiCMOS buffer optimization," *IEEE J. Solid-State Circuits*, vol. 27, no. 7, pp. 1014–1019, July 1992.

[12] S. S. Sapatnekar, "RC interconnect optimization under the Elmore delay model," in *Proc. ACM/IEEE Design Automation Conf.*, 1994, pp. 387–391.

[13] J. C. Shah and S. S. Sapatnekar, "Wiresizing and buffer sizing for power-delay tradeoffs using a sensitivity based heuristic," Tech. Rep. ISU-CPRE-95-SS01, Dept. Electrical and Computer Engineering, Iowa State University, Ames IA 50011.

[14] K. J. Singh and A. Sangiovanni-Vincentelli, "A heuristic algorithm for the fanout problem," in *Proc. 27th ACM/IEEE Design Automation Conf.*, 1990, pp. 357–360.

[15] Synopsys 3.1 Release Manual: Appendix B, "Static timing analysis."

[16] H. J. Touati, "Performance-oriented technology mapping," Ph.D. dissertation, Memo. UCM/ERL M90/109, Dept. of Electrical Engineering and Computer Science, UC Berkeley, Nov. 28, 1990.

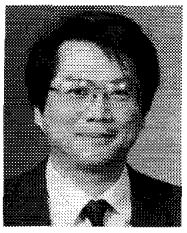
[17] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *Proc. Int. Symp. on Circuits and Systems*, 1990, pp. 865–868.

[18] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Reading, MA: Addison-Wesley, pp. 231–237, 1993.

[19] F. F. Yao, "Computational Geometry," *Handbook of Theoretical Computer Science*, Elsevier Science, ch. 7, vol. A, 1990.

John Lillis received the B.S. degree in computer science from the University of Washington, Seattle, in 1989 and the M.S. degree in computer science from the University of California, San Diego, in 1992. He is currently working toward the Ph.D. degree in the Computer Science Department at the University of California, San Diego.

His research interests are in computer aided design of VLSI circuits and combinatorial optimization.



Chung-Kuan Cheng (S'82–M'84–SM'95) received the B.S. and M.S. degrees in electrical engineering from National Taiwan University, and the Ph.D. degree in electrical engineering and computer sciences from University of California, Berkeley, in 1984.

From 1984 to 1986 he was a senior CAD engineer at Advanced Micro Devices Inc. In 1986, he joined the University of California, San Diego, where he is currently an Associate Professor in the Computer Science and Engineering Department. His research interests include network optimization and design automation on microelectronic circuits.

Dr. Cheng is an Associate Editor of IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN.

Ting-Ting Y. Lin (S'84–M'87) received the B.S. degree from National Chiao-Tung University, Hsinchu, Taiwan, R.O.C. and the Ph.D. degree in computer engineering from Carnegie Mellon University, Pittsburgh, PA.

She is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of California, San Diego. Her research interests include design automation, VLSI testing, fault modeling, and system dependability.

Dr. Lin is a member of Sigma Xi.