BUFFER INSERTION IN LARGE CIRCUITS USING LOOK-AHEAD AND BACK-OFF TECHNIQUES

A Thesis

by

MANDAR WAGHMODE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2005

Major Subject: Computer Engineering

BUFFER INSERTION IN LARGE CIRCUITS USING LOOK-AHEAD AND BACK-OFF TECHNIQUES

A Thesis

by

MANDAR WAGHMODE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Weiping Shi
Committee Members,	Jiang Hu
	D. M. H. Walker
Head of Department,	C. N. Georghiades

December 2005

Major Subject: Computer Engineering

ABSTRACT

Buffer Insertion in Large Circuits
Using Look-ahead and Back-off Techniques. (December 2005)
Mandar Waghmode, Bachelor of Engineering, University of Pune, India
Chair of Advisory Committee: Dr. Weiping Shi

Buffer insertion is an essential technique for reducing interconnect delay in sub-micron circuits. Though it is a well researched area, there is a need for robust and effective algorithms to perform buffer insertion at the circuit level. This thesis proposes a new buffer insertion algorithm for large circuits. The algorithm finds a buffering solution for the entire circuit such that buffer cost is minimized and the timing requirements of the circuit are satisfied. The algorithm iteratively inserts buffers in the circuit improving the circuit delay step by step. At the core of this algorithm are very simple but extremely effective techniques that constructively guide the search for a good buffering solution. A flexibility to adapt to the user's requirements and the ability to reduce the number of buffers are the strengths of this algorithm. Experimental results on ISCAS85 benchmark circuits show that the proposed algorithm, on average, yields 36% reduction in the number of buffers, and runs three times faster than one of the best known previously researched algorithms.

To my parents

ACKNOWLEDGMENTS

I am greatly indebted to my advisor, Dr. Weiping Shi. Sir, I could not have realized my potential without your invaluble guidance, consistent encouragement and emphasis on quality of the research contribution. My words are simply insufficient to express gratitude towards you.

Special thanks to Dr. Jiang Hu and Zhuo Li for the brain-storming sessions we had during the course of this research and to Cliff Sze for making their research available for comparison.

I also extend my sincere gratitude to many people who made my masters program at Texas A&M University one of the most enriching experiences I have ever had. Quality and Excellence is literally overflowing in my teachers, mentors and collegues here. I am specially grateful to Dr. Sunil Khatri and Dr. M. Ray Mercer in this regard.

Last, but not the least, without constant support and encouragement of my parents and friends, it was not possible for me to come thus far. They have always been and will always be there for me.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A. A Little Digression

A one-day game of cricket, first fifteen overs and oh, how delightful is the stroke-play of the likes of India's "Little-Master-Sachin" or Australia's "Always-Delivers-Gilchrist"! The field restrictions are in place, and the batsmen have the luxury of taking risks in order to score at a rapid pace. In contrast, in the middle of the innings, the focus is on being watchful, taking fewer risks and building a solid inning. That is why the likes of India's "The-Wall-Dravid" and Pakistan's "Mammoth-Inzi excel" in the middle order. And then come the stars of the slog overs like South Africa's "Zulu-Klusner" smashing the ball mercilessly all over the field. There is nothing to lose in the slog overs and making the most of the remaining deliveries is the key. Do these cricket strategies suggest how to perform buffer insertion? How uncomplicated the life of a VLSI CAD engineer would be if buffer insertion could be done effectively with such simple strategies!

Similarly, in the game of chess, masters think of their own as well as opponent's possible future moves and strategize accordingly. The more they can look ahead into the future moves, the greater is their mastery. Early in the game, few future moves need to be visualized. Later on in the game, one has to be more careful, and need to think of many more possible moves. Is this strategy applicable to buffer insertion?

As interconnect delay poses a limit to the performance of VLSI circuits, the cost of required buffering resources to meet the timing constraints is exploding [7].

The journal model is *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*

Therefore, we want to minimize the number of buffers inserted in the circuit while satisfying the timing requirements. It is akin to the goal of scoring maximum runs in the alloted fifty overs of a one-day cricket game. Then, are the cricketing strategies applicable to buffer insertion? Let us see. Start from a state where no buffers are inserted in the circuit and the slack is worst. Then continue adding buffers one by one till the timing requirements are satisfied. Early on, don't scratch your head much because the chances of inserting a buffer in the wrong net are low. Relative improvement in slack per newly added buffer is high. Later on, when the slack has improved sufficiently, the returns from each newly added buffer are lower. At this point, buffers should be inserted carefully because the chances of inserting buffers in the wrong place are high.

Well, the next question is – how can we know if it is advantageous to insert a buffer in a particular location? Like a chess master, this can be achieved by looking ahead. Look a few steps into the future, evaluate which move is likely to give higher slack improvements when some more buffers get added, and fix the most advantageous move. If you try to look ahead into all permutations and combinations of the future moves, you will exhaust the whole solution space and get the exact optimum solution but it will need enormous amounts of time to solve the problem. So to borrow from a chess master's strategy, early on, you need not look much ahead. Later on, you need to look ahead more and evaluate the possible moves carefully.

The next question is – does this strategy of *looking ahead just as much as needed* yield an efficient way to solve buffer insertion problem? No – one crucial item is still missing from the strategy. We have one luxury in buffer insertion that the cricketers or chess-masters don't have. What if somebody tells the batsman that he can choose a few bad shots in his innings, those deliveries will be bowled at him again and he can play them afresh? Or if somebody told a grand-master that he can to choose

some bad moves, take them back and play alternative moves instead? This is what we call the *back-off* strategy, where we determine the least effective moves that we made while inserting buffers, and revert them.

The back-off strategy adds a great amount of flexibility and effectiveness to the overall buffer insertion algorithm. It not only improves the cost performance of the algorithm but also the run-time of the algorithm. The cost improvement seems intuitive while the run-time improvement doesn't. The reason for the run-time improvement is that the back-off strategy gives us the power to be much more aggressive while inserting buffers. With the back-off strategy in our arsenal, we can cut back on the number of look-ahead levels while inserting buffers and this results in a much faster algorithm. It also gives rise to a flexible algorithm because we can use these two strategies one after other with varying intensities at different stages of the algorithm. It provides us a basic framework that can be used to trade off the solution cost and run-time performance. These two simple ideas give us *LAB* (Look-ahead And Back-off), the new buffer insertion algorithm that is proposed in this work.

B. Motivation

Buffer insertion is a very effective technique for reducing interconnect delay. Buffer insertion for a single net or interconnect tree is a well-researched problem. L.P.P. van Ginneken [1] proposed an $O(n^2)$ time dynamic programming algorithm in 1991 to maximize the slack of the net. Since then, his algorithm has become a classic in this field and a substantial body of research has developed on the basis of van Ginneken's algorithm. The work of [8] suggested a wire segmenting algorithm to be used as a precursor to van Ginneken's algorithm resulting in faster run-time. Lillis *et al.* [2] extended the framework to minimize buffer cost while satisfying the

timing requirements. Li *et al.* [9] improved the time bound on van Ginneken's algorithm to $O(n \log n)$. The authors of [10] prove that optimizing the total cost given arbitrary buffer costs is a NP-hard problem, and also suggest techniques to improve the efficiency of Lillis' algorithm. Previous researchers [16, 14, 15] have taken other approaches to solve different variants of the buffer insertion problem like simultaneous routing, simultaneous gate sizing, and inclusion of slew and signal integrity constraints.

In real applications, however, the primary objective is to reduce path delay in combinational circuits rather single net delay. Therefore, buffer insertion should be performed at the circuit level rather than at the net-level. This calls for efficient algorithms at the circuit level that capitalize on the progress made by the faster net-level buffer insertion algorithms cited above. The motivation of this research work is to develop such a circuit level buffer insertion algorithm that uses efficient net-level algorithms as its subroutines and builds upon them.

A simple-minded approach could be to apply van Ginneken's algorithm, one net at a time from primary outputs to primary inputs. Although this approach guarantees that the slack at the primary input nodes is maximized, too many buffers will be used since van Ginneken's algorithm does not control buffer cost. To use Lillis' framework, which controls buffering resources, will also run into problems due to the re-convergences which are frequently encountered in combinational circuits.

C. Previous Work

A Lagrangian relaxation based algorithm for circuit level buffer insertion was proposed in [3, 4]. A restrictive assumption, that buffers are placed equidistant from each other is used in [3]. In practice, however, the availability of space dictates whether a buffer

can be inserted in a particular location. The work of [4] tries to get around this restrictive assumption but resulting algorithms do not scale very well. In [3], the CPU time is prohibitive even without cost optimization. With cost optimizations and the restrictive assumptions are removed, it is likely to get much worse.

A path based buffer insertion algorithm is proposed in [5] that builds on the dynamic programming approach of [1, 2]. It inserts buffers on statically computed critical paths in the order of their criticality. However, due to its local focus on critical paths, it may actually insert buffers that are locally effective on a particular path but less effective when the whole circuit is considered. Highly critical paths in the circuit require as many buffering resources as possible, and there is less scope to reduce the buffer cost in such areas. Comparatively, the less critical areas of the circuit offer more scope for cost optimization, but they get a lower priority in this scheme.

A network flow based algorithm is suggested in [6]. It tries to identify the nets which should be given priority for inserting buffers by using the min-cut idea in network flow problems. Though this idea is likely to overcome the disadvantages of path based methods, this work assumes that buffers are placed to decouple certain parts of the nets irrespective of layout space availability and also assumes that the buffers are placed equidistantly in the interconnect. As mentioned earlier, such a placement may not be possible in a pre-routed circuit. The quality of the solution deteriorates when the buffer positions are adjusted during the legalization step.

[11, 12, 13] also address the circuit level buffer insertion problem coupled with other problems such as accurate delay modeling and transistor sizing.

D. Organization of the Thesis

The rest of the chapters in this thesis are organized as follows. Chapter II presents the problem statement. Chapter III explains the main idea of the look-ahead and back-off strategies of the algorithm. Chapter IV describes the methods to further speedup the algorithm. The experimental results for LAB algorithm are presented in Chapter V. This heuristic approach of using look-ahead and back-off strategies is the main contribution of this thesis.

A framework to find the exact optimum solution to the buffer insertion problem was also developed in the early part of this research. This framework is presented in Chapter VI. Another earlier work, in which delay models and insertion algorithm were developed for interconnects with boosters [17], is also presented in Chapter VII. Readers interested only in LAB can skip Chapter VI and Chapter VII. Chapter VIII concludes the findings of this work and describes avenues for future work.

CHAPTER II

PROBLEM DESCRIPTION AND PRELIMINARIES

A. Problem Description

We represent a combinational circuit as a Directed Acyclic Graph (DAG) $G = (V, E)$. The set $V = V_t \cup V_n$ is a set of vertices (nodes) in the graph, where V_t comprises of primary input, primary output, gate input and gate output nodes in the circuit, and V_n is set of internal nodes in the interconnect routing trees. The set of edges E consists of the edges in the interconnect routing trees and input-to-output edges connecting the input and output nodes of the gates. Fig. 1 shows an example combinational circuit and the corresponding DAG representation.

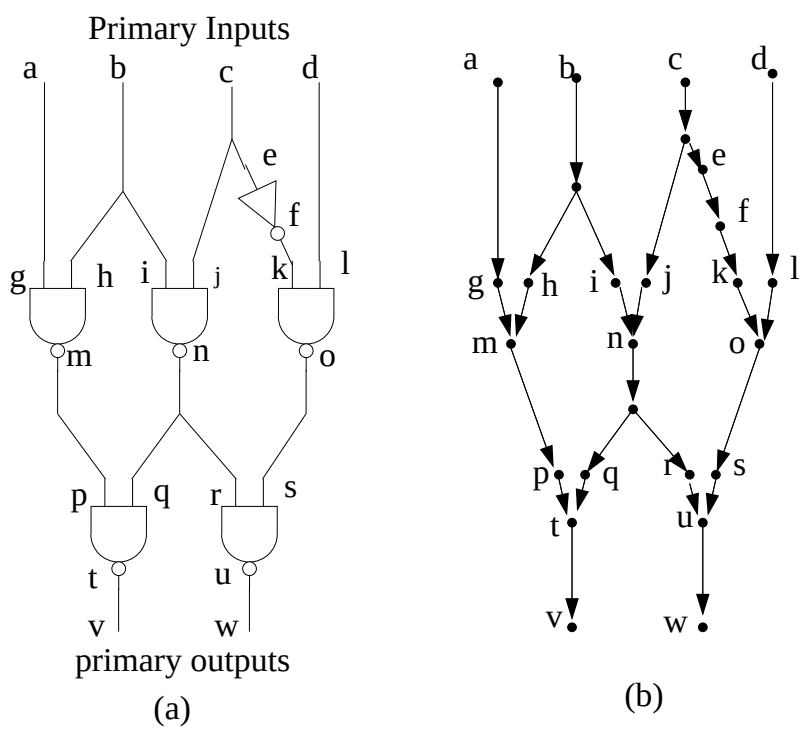


Fig. 1. (a) Example combinational circuit. (b) Corresponding DAG representation.

A buffer library $\mathbf{B}$ is also provided as a part of problem statement. The locations where buffers can be inserted are given as a function $f : V_n \rightarrow 2^{\mathbf{B}}$. Under this definition, each node in the interconnect routing tree allows certain types of buffers, or no buffer. Each buffer type $b_i \in \mathbf{B}$ is modeled by driving resistance $R(b_i)$, input capacitance $C(b_i)$, intrinsic delay $D(b_i)$ and cost $W(b_i)$. The cost of a buffer can be either area or power or any other criteria, depending on the optimization objective.

Each interconnect edge e is modeled as a π type RC circuit and is associated with resistance $R(e)$ and capacitance $C(e)$. Each gate is modeled in similar manner as a buffer. Thus each edge e connecting a gate input to a gate output is associated with delay $D(e)$, each gate input node v is associated with input capacitance $C(v)$ and each gate output node v is associated with driving resistance $R(v)$.

Following previous researchers [1, 2, 8], the Elmore delay model is used for interconnect and a linear delay model is used for gates and buffers. For each edge $e = (v_i, v_j)$, signals travel from v_i to v_j. The Elmore delay of e is $D(e) = R(e)\left(\frac{C(e)}{2} + C(v_j)\right)$, where $C(v_j)$ is the downstream capacitance at v_j. For any gate or buffer b at vertex v_j, the gate or buffer delay is $D(v_j) = K(b) + R(b) \cdot C(v_j)$, where $C(v_j)$ is the downstream capacitance at v_j. For a gate input node v, the capacitance viewed from upstream is $C(v)$. Similarly, the capacitance viewed from upstream for inserted buffer b is $C(b)$.

For any subgraph $G' = (V', E')$, the set of its input nodes $I(G')$ is such that no edge in E' is directed towards the nodes in $I(G')$. Similarly, a set of its output nodes $O(G')$ is such that no edge is directed away from the nodes in $O(G')$. As an example, for $G' = G$, $I(G')$ is a set of all primary inputs and $O(G')$ is a set of all primary outputs.

Consider a subgraph $G' = (V', E')$ where $V' \subset V$, $E' \subset E$ such that if $v \in V'$ is a node in an interconnect routing tree in G, then the whole routing tree is in G'. A *buffer assignment* is a function $\alpha(G') : V'_n \rightarrow \mathbf{B} \cup \{\emptyset\}$ that specifies the type

of buffer inserted for each node in V'_n, where V'_n is a set of legal buffer locations in the interconnect routing tree. Since each of these assignments is a candidate for the optimal one, we also refer them as *candidates*. $W(\alpha(G'))$ denotes the total buffer cost of $\alpha(G')$.

If a directed path exists from node u to node v in G', then delay of the path from u to v under assignment α is defined as

$$D(u, v, \alpha) = \sum_{e=(v_j,v_k)} (D(v_j) + D(e)),$$

where the sum is over all edges e in the path from u to v.

The Required Arrival Time (RAT) $Q(u)$ at node u is a user-specified value if u is a primary output node. Otherwise, RAT at node u under $\alpha(G')$ is defined as follows.

$$Q(u, \alpha(G')) = \min_{v \in O(G')} \{Q(v) - D(u, v, \alpha(G'))\}$$

Also, the Arrival Time (AT) $T(u)$ at a node u is a user specified value if u is a primary input node. Otherwise, AT at node u under $\alpha(G')$ is defined as follows.

$$T(u, \alpha(G')) = \max_{v \in I(G')} \{T(v) + D(v, u, \alpha(G'))\}$$

The Slack at a node u under $\alpha(G')$ is defined as follows.

$$S(u, \alpha(G')) = T(u, \alpha(G')) - Q(u, \alpha(G'))$$

Also, the slack of subcircuit G' under $\alpha(G')$ is given as

$$S(\alpha(G')) = \min_{v \in I(G')} S(v, \alpha(G'))$$

A buffer assignment $\alpha(G)$ *satisfies* the timing requirements of the circuit if $S(\alpha(G)) > 0$. The optimum solution to this problem is a buffer assignment that has minimum cost among all the assignments, and satisfies the timing requirements

of the circuit.

The main algorithm presented in this work is a heuristic approach that tries to minimize the buffer cost without guaranteeing an exact optimum solution. Additionally, a framework to find the exact optimum solution was also developed in the early part of this research. This framework is also presented in Chapter VI.

B. Preliminaries

Consider the dynamic programming based approaches of [2] and [10] to find the optimal cost buffer assignment for a tree structure. Any assignment of a routing subtree is represented as a triplet (Q, C, W) in these algorithms where Q is RAT at the root of the subtree, C is the capacitive load presented to the upstream and W is the cost of the buffers inserted under the assignment. Let us generalize this representation for any circuit subgraph. In general, to adopt a dynamic programming approach, an assignment for any circuit subgraph can be completely expressed as a vector with the following parameters.

- Timing parameters:

  This parameter is either the RAT or the AT, depending upon the direction in which the subcircuits are being processed in the dynamic programming approach. Also, depending on the nature of the subgraph under consideration, the number of nodes for which timing parameters need to be represented in the vector may differ. As is evident from the definitions of RAT and AT, greater the RAT or lesser the AT, better is the assignment in terms of timing.

  For example, in the framework of [2], since the subgraph under consideration is a tree and it is processed from leaves towards root, at RAT at exactly one node is sufficient to represent a candidate. Later on, in Chapter VI, we will

develop a framework which uses RAT/AT at more than one node to represent a candidate.

- Loading or Driving Parameters:

  These parameters are either capacitive load seen by the upstream or driving resistance seen by the downstream. Similar to timing parameters, the nature and number of these parameters in the vector depends on the direction of processing and the nature of the subgraph. For example, in the framework of [2], capacitive loading at only one node is sufficient to represent a candidate.The greater the driving resistance or lesser the capacitive loading, the better is the assignment.

- Cost:

  This is simply the total cost of buffers inserted under an assignment. The lesser the cost, the better the assignment in terms of cost.

Given two assignments α_1 and α_2, we say α_1 *dominates* α_2, if α_1 has lower cost, better timing and better loading/driving parameters compared to α_2. Note that the specifics specifics of the representation of an assignment for various types of subcircuits will be discussed in subsequent chapters as required.

The set of *non-redundant assignments* for any subgraph G', denoted as $N(G')$ is a set of assignments such that no assignment in $N(G')$ dominates any other assignment in $N(G')$ and any buffer assignment for G' is dominated by some assignment in $N(G')$.

CHAPTER III

LOOK-AHEAD AND BACK-OFF STRATEGIES

A. The Concept

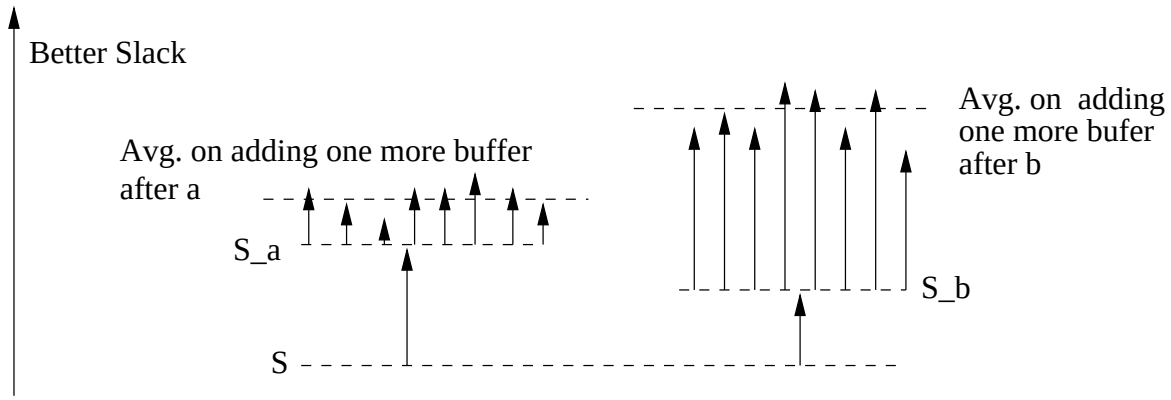


Fig. 2. Look-ahead strategy for buffer insertion.

As shown in Figure 2, suppose we have added a certain amount of buffers already in the circuit, and the resulting circuit slack is S. If we add one more buffer in position a, the circuit slack becomes S_a. Alternatively, if we add a buffer in position b, the circuit slack becomes S_b. Also, let S_b be lower (i.e. worse) than S_a. But if we choose to insert buffer at a and then try inserting one more buffer elsewhere in the circuit, the average slack improvement is much lower compared to the case where we would have chosen to insert buffer at b. Thus, just by looking one level ahead, we can make a better decision about which buffer should be inserted. For sake of simplicity, let us say we have only one buffer type. Then, the *look-ahead level* is simply the number of additional buffers we try while evaluating the future effectiveness of inserting a buffer in a particular location.

Figure 3 shows the cost vs. delay profile of a sample circuit with 2 nets. It

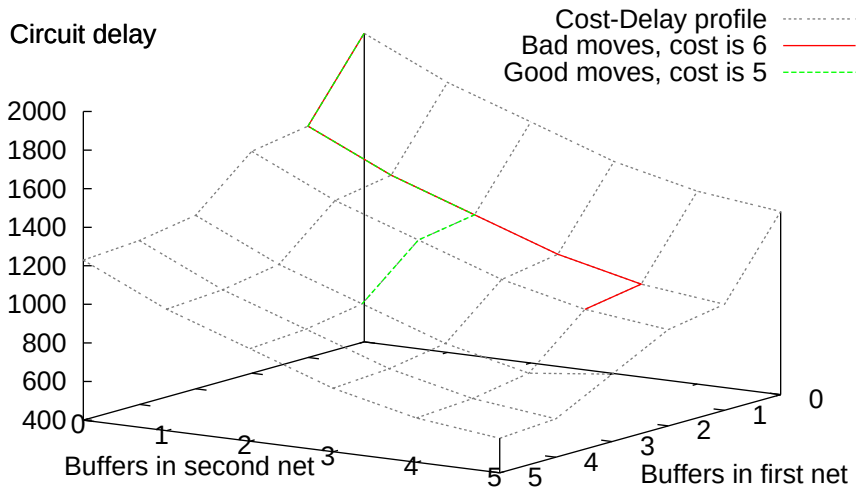


Fig. 3. Example cost vs. delay profile, and benefit of looking ahead.

illustrates that without lookahead, 6 buffers are required to satisfy the timing requirements whereas with lookahead, 5 buffer are required and thus one buffer could have been saved by looking ahead one level.

As we discussed in the introduction to this thesis in Chapter I, the concepts of look-ahead and back-off are as simple as evaluating the consequences of a move before committing to it in chess or allowing the grand master to take back their moves. But to describe them formally, let us define a few more terms.

B. Few More Definitions

As a side note, given the RAT at each leaf node of a net, non-redundant assignments for the whole net can be expressed by $(RAT, Cost)$ i.e. (Q, W) pairs. Also, there

is a unique non-redundant assignment for each possible value of W. Let us use the symbol η to denote a net. Lillis' algorithm [2] along with predictive pruning [10] is a core subroutine used in LAB for the purpose of finding $N(\eta)$ given the RAT at each leaf of η.

Let us now define two operations, namely *incrementing and decrementing an assignment*. Let $\alpha(G)$ be an assignment of the whole circuit graph G, and W be the cost of buffers inserted under this assignment in net η. Let $N(\eta)$ be the set of non-redundant candidates when RAT $Q(v)$ at each sink v of η is $Q(v) = Q(v, \alpha(G))$. Also, let $\alpha_a(\eta)$ be the assignment in $N(\eta)$ with next higher cost than W. *Incrementing* $\alpha(G)$ *over* η, represented as $\{\alpha(G)\}^{+\eta}$, is defined as follows:

$$\{\alpha(G)\}^{+\eta} = \{\alpha(G \setminus \eta)\} \cup \alpha_a(\eta)$$

Similarly, let $\alpha_b(\eta)$ be the assignment in $N(\eta)$ with next lower cost than W. *Decrementing* $\alpha(G)$ *over* η, represented as $\{\alpha(G)\}^{-\eta}$, is defined as follows:

$$\{\alpha(G)\}^{-\eta} = \{\alpha(G \setminus \eta)\} \cup \alpha_b(\eta)$$

C. Look-ahead

Now that an increment operator is defined for an assignment, a subroutine to find next assignment by looking l levels ahead, in its simplest form, can be represented by the pseudo-code in Figure 4.

Note that when we decide the next assignment after a look-ahead step, we are only incrementing the number of buffers or the cost of buffers inserted in the selected net and we are not fixing the exact positions for these inserted buffers. This is because optimal placement of buffers for a net may be drastically different for different buffer costs. In other words, buffer positions chosen under a lower cost non-redundant

```
Let α(G) be current assignment, l be look-ahead level and n be
  number of nets in G.
for each net η ∈ G do
  trial assignment α'(G) = α(G)
  Cumulative Slack Improvement I(η) = 0
  for each combination c in ^nC_l combinations of nets in G do
    for each net η in combination c do
      α'(G) = {α'(G)}^{+η}
    end for
    I(η) = I(η) + {S(α'(G)) − S(α(G))}
  end for
end for
Choose net η' : I(η') = max_{η∈G}{I(η)}
Next assignment after look-ahead α(G) = {α(G)}^{+η'}
```

Fig. 4. Look-ahead in its simplest form.

assignment may not prove to be good choices for a higher cost assignment. Also, RAT at the nodes in the circuit keeps changing as we go on inserting or removing buffers. Hence we just specify the cost of buffers inserted in each net under an assignment rather than the exact buffer positions in the net.

For a simple case of only one buffer type, it can be seen from the pseudo-code that the number of circuit slack computations performed in one look-ahead iteration get multiplied by n when l is increased by one. Therefore, more approximations are required to make the approach practically applicable. These will be discussed in the

following chapter.

D. Back-off

Similar to the look-ahead strategy, we can look-back a few levels to decide the buffers that are least effective and remove them. The idea here is to let the look-ahead do the job of careful selection, and back-off by a few steps to correct the decisions taken by look-ahead which proved less effective in retrospect. Therefore we need a faster back-off routine which need not be vary careful while removing buffers but should be effective enough to allow aggressive and faster look-ahead. Thus, the back-off routine in its simplest form can be represented by the pseudo-code in Figure 5.

```
Let α(G) be current assignment, n be number of nets in G.
for each net η ∈ G do
   Trial assignment α'(G) = {α(G)}^{-η}
   Reduction in slack F(η) = {S(α(G)) − S(α'(G))}
end for
choose net η' : F(η') = min_{η∈G}{F(η)}
Next assignment after back-off α(G) = {α(G)}^{-η'}
```

Fig. 5. Back-off in its simplest form.

As seen above, we are not fixing any buffer positions while deciding the assignment after back-off but just decrementing the cost or number of buffers inserted in a particular net. Similar to look-ahead, we employ some clever tricks in back-off as well to make it faster without loosing its effectiveness. These speed up techniques are discussed in the following chapter.

CHAPTER IV

SPEED-UP TECHNIQUES

From the pseudo-code presented in Figures 4 and 5, we can see that the most compute intensive task is that of finding circuit slack while evaluating the effectiveness of the possible moves. To be more specific, an assignment specifies the cost or number of buffers to be inserted in each net. The circuit slack is then computed by processing the nets, with Lillis' (Q, C, W) framework [2, 10], in their topologically sorted order. Thus, more the number of circuit slack computations performed by the algorithm, more is the run-time of the algorithm. Further looking closely at a single circuit slack computation, we can see that non-redundant assignments for a 2-pin net are independent of the RAT at its sink. Therefore, non-redundant candidates for a 2-pin net can be computed in the preprocessing step even before starting with the algorithm. Thus, most of the CPU time in one circuit slack computation is spent in processing multi-pin nets with (Q, C, W) framework.

Thus the key to speed-up this scheme is to reduce the number of circuit slack computations and more specifically the number of non-redundant candidate computations on multi-pin nets. The speed-up techniques presented in this chapter are very effective. There can be more than one ways of employing these techniques and hence the implementation presented here is intended only to serve as an example.

A. Reducing the Number of Buffer Positions Evaluated

As described in the previous chapter, we need not evaluate the effect of adding a buffer in each and every net. Adding a new buffer in one of the critical nets is most likely to provide the most advantageous move. Therefore, the proposition is to determine the critical nets after adding each buffer and restricting the algorithm to try only the

critical nets for the next move.

With each new buffer added in the circuit, the nets that were previously critical may become non-critical and vice versa. Since the critical nets are not statically determined at the beginning of the algorithm, the algorithm does not lose its global view by restricting itself to critical nets. Thus the sacrifice in terms of quality or buffer cost is not as significant compared to the gain in CPU time. Moreover, the number of nets evaluated for a possible move can be changed depending on the stage the algorithm is in.

Also, rather than just adding the buffer that gives the best improvement in each iteration of look-ahead, we can add more than one buffer in one iteration. The proposition is to determine the critical nets after adding each new buffer and continue adding the second best buffer and then the third best buffer and so on as long as the buffers are being added on dynamically determined critical nets. Again, the algorithm can be more or less conservative about the number of buffers being added in one iteration, depending on the stage it is in. Also, the later back-off iterations will more than likely correct the moves in case they later prove to be less effective.

B. Reducing the Effort of Processing Multi-pin Nets

Since we add or remove buffers one at a time, previously computed RAT information can be reused. Thus, if we change the number of buffers inserted in a particular net, then the RAT needs to be updated only for the nodes in the fan-in cone of the net in question.

Moreover, we can compromise the computation of the exact slack value wherever exact slack information is not needed. For example, while backing-off during the early steps of the algorithm, the purpose is only to judge less effective moves and an

approximate circuit slack value can be used for this purpose. Also, if l is the look-ahead level, then the circuit slack of the assignment found by adding l^{th} additional buffer is needed only for estimation purposes, and hence can be approximated.

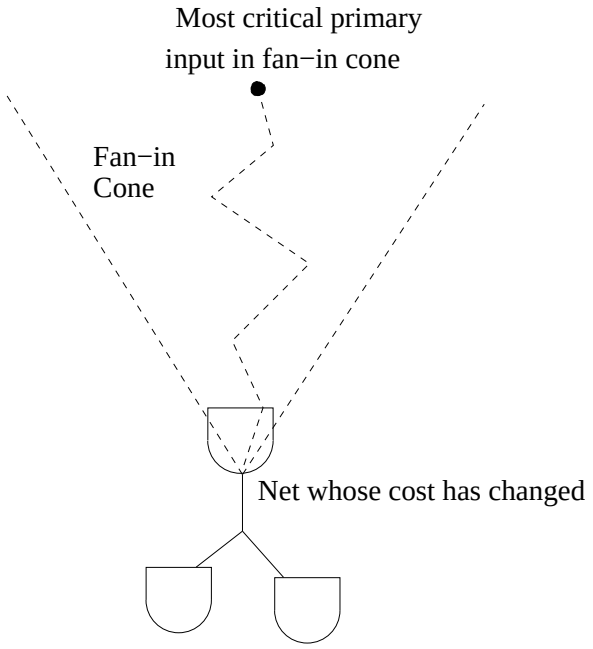


Fig. 6. Estimating circuit slack approximately.

More specifically, after an assignment has been incremented or decremented over net η, referring to Figure 6, the proposition is to process only the most critical path in the fan-in cone of η to get the new approximate or very likely value of circuit slack. This way, we can get approximate circuit slack by processing significantly fewer nets.

Also, if the given circuit has larger nets with many buffer positions, then such interconnect trees can be partitioned by fixing a buffer temporarily in a legal position that partitions the tree proportionately. Such restrictions can be placed in the early stages of the algorithm in order to speed up the computation. These restrictions can be removed later in order to achieve better quality.

C. Fast and Greedy Net-based Buffer Insertion

Rather than looking ahead fewer levels, a more aggressive strategy is to determine the critical nets in the circuit and to populate the critical nets greedily with buffers till the given slack requirement is achieved. In this manner, we can reduce the number of circuit slack computations significantly but at the cost of large number of buffers. Since we invoke the back-off strategy to remove unnecessary buffers, we can afford to flood the circuit with such aggressive insertion of buffers in the initial stages of the algorithm. Also, a careful selection of critical nets can prevent such a greedy insertion from adding a lot of less effective buffers. A routine employed for this purpose in the current implementation is sketched in Figure 7.

Note that this sketch is just one way of using the idea of flooding the circuit with buffers in a greedy manner i.e. just looking at local improvements in slack at the source of critical nets. Aggressive and fast buffer insertion can be performed in some other manner as well.

D. Faster Identification of Some Back-off Moves

Let $\alpha(G)$ be the current assignment of the whole circuit graph and $\alpha(\eta) = \alpha(G \cap \eta)$ represent the partial buffer assignment in net η under $\alpha(G)$. Let node u be the root or source node of η. Also, let $\alpha'(\eta)$ be the non-redundant assignment having next lower cost than $W(\alpha(\eta))$. Consider a condition such that:

$$S(u, \alpha(G)) > (S(\alpha(\eta)) - S(\alpha'(\eta)))$$

If this condition is true for some net $\eta \in G$, then it can be easily seen that there is enough surplus slack at node u such that decrementing $\alpha(G)$ over η will not change the circuit slack. Thus, the decision of adding the last buffer in η had no effect on

```
Let α(G) be current assignment and S' be desired circuit slack.
while S(α(G)) < S' do
  find set of critical nets R
  for Each critical net η ∈ R do
    α(η) = α(G ∩ η), Local improvement L(η) = S({α(η)}^{+η}) − S(α(η))
  end for
  Cumulative improvement I = 0.
  Sort the nets in R in the decreasing order of L(η)
  for each net η in sorted order do
    α(G) = {α(G)}^{+η}, I = I + L(η)
    if (I > (S' − S(α(G)))) then break the for loop
  end for
end while
Next assignment after fast insertion is α(G).
```

Fig. 7. Aggressive net-based buffer insertion.

the circuit slack and hence η can be safely chosen for back-off. Note that the effort of computing circuit slack for all possible back-off moves and then choosing the best move is saved by identifying the back-off moves with above criterion.

Also, an option of applying this criterion more aggressively, i.e. choosing nets for back-off that are likely to have little if any effect on circuit slack based on such a local comparison, yields additional speedup in the earlier stages of algorithm.

E. Overall Flow

In this and previous chapters, we have referred to utilizing different strategies at different stages in the algorithm. Figure 8 shows the overall flow of the algorithm that brings together various strategies discussed till now. As seen in Figure 8, trade-offs can be made in the performance of the algorithm with two input parameters i.e. look-ahead level and cost of buffers inserted with net-based insertion. This flow just serves as an example. More flexibility can be provided in terms of more input parameters and controlling the interleaving of the look-ahead and back-off routine and their intensities.

In our experiments with ISCAS85 benchmark circuits, looking ahead just by one level yielded tremendous improvements in cost with very efficient run-times. Also, inserting 80% of the buffers with the net-based greedy buffer insertion did not harm the cost-performance of the algorithm. Thus referring to Figure 8, $l = 1$ and $p = 0.8$ gave a good balance between run-time and buffer cost for experimental ISCAS85 test-cases. The experimental results are presented in Chapter V.

```
Input parameters:  l=Look-ahead level, p=Cost inserted with
  net-based insertion as a fraction of initial cost estimate W_i.
Let α(G) be an assignment such that W(α(G)) = 0
while S(α(G)) < 0 do
  Update α(G) with net-based insertion and back-off.
end while
Initial cost estimate W_i = W(α(G))
while W(α(G)) > (p × W_i) do
  Decrement α(G) with back-off.
end while
while S(α(G)) < 0 do
  Update α(G) with look-ahead and back-off.
end while
return α(G)
```

Fig. 8. Top-level view of the algorithm.

CHAPTER V

EXPERIMENTAL RESULTS OF LAB ALGORITHM

The newly proposed Look-ahead and Back-off (LAB) algorithm is compared with path based buffer insertion (PBBI) algorithm of [5]. Table I shows the ISCAS85 benchmark circuits and their respective sizes in terms of number of source and sink nodes. The test-cases are created by scaling the actual layouts performed in 0.18μ technology, so as to create the need for buffering. Only one buffer type is used for these experiments.

Table I. Size of the benchmark circuits.

Ckt	#Sources	#Sinks	#Buffer Locations
c432	196	343	868
c499	243	440	1216
c880	443	775	1632
c1355	587	1096	1868
c1908	913	1523	4037
c2670	1502	2292	7192
c3540	1719	2961	7729
c5315	2485	4509	11403
c6288	2448	4832	10865
c7552	3720	6253	16758

Timing constraints are determined by maximum achievable slack with given legal buffer positions. The RAT at the primary output nodes is computed according to the maximum achievable slack.

Figures 9, 10(a) and 10(b) present the experimental statistics showing the effect of various intensities of look-ahead and back-off on the buffer cost. Figure 9 shows the percentage improvement in slack per additional percentile of buffer cost for the test circuits. It can be seen that due to the tightest possible timing constraints, almost 40% of the buffers added in the later stage result only in 10% improvement in slack.

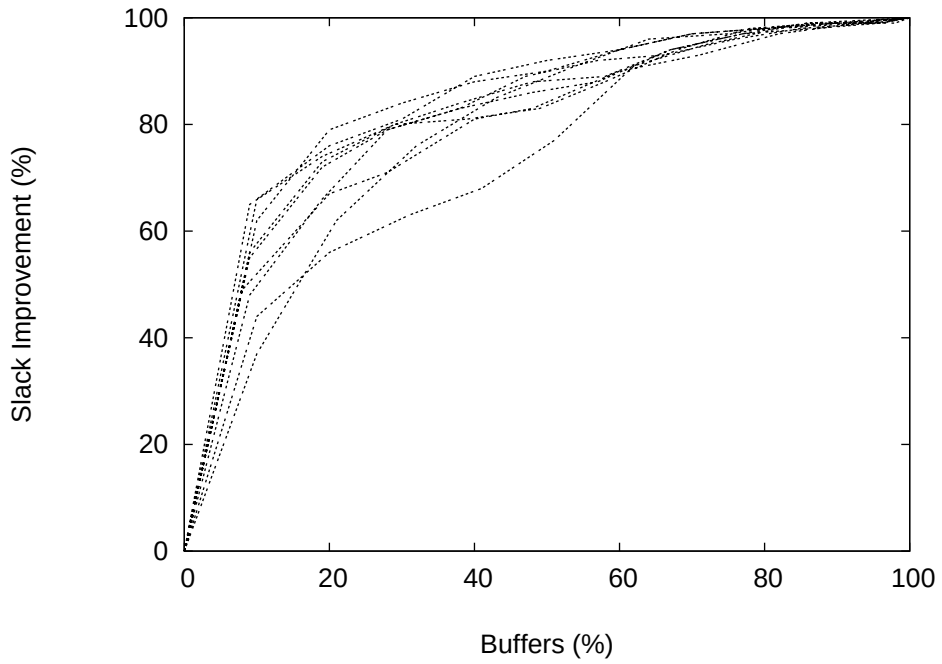


Fig. 9. Additional cost vs slack improvement for test-cases.

Figure 10(a) shows improvement in cost performance as we go on increasing the look-ahead level. The results in Figure 10(a) are obtained without back-off just for comparison purpose. It can be seen that the improvement saturates after look-ahead level 1 for most of the test-cases.

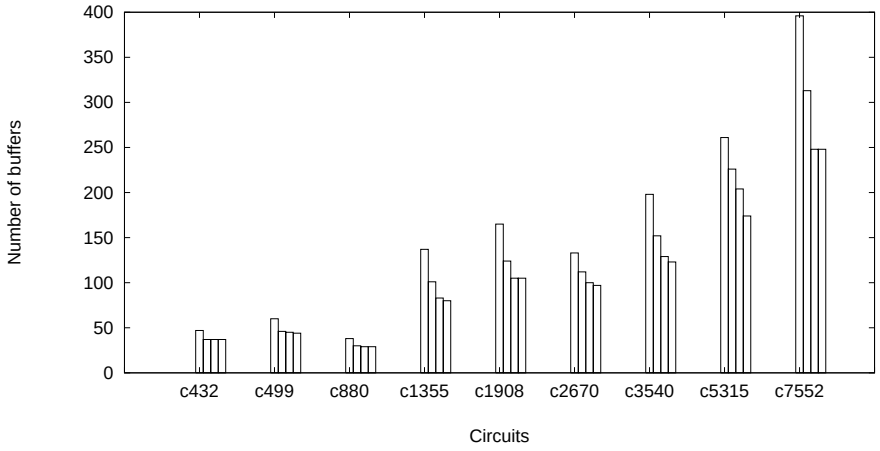
Figure 10(b) shows the cost performance for various look-ahead levels with back-off. It can be seen that the difference in the number of buffers inserted for different look-ahead levels is very less compared to Figure 10(a). Also, performance of greedy

net-based insertion with back-off is close to higher look-ahead levels. Hence net-based insertion can be used to insert 80% to 90% of the buffers initially and higher look-ahead levels can be used to insert remaining buffers. This results in the desired balance of run-time and buffer cost minimization.

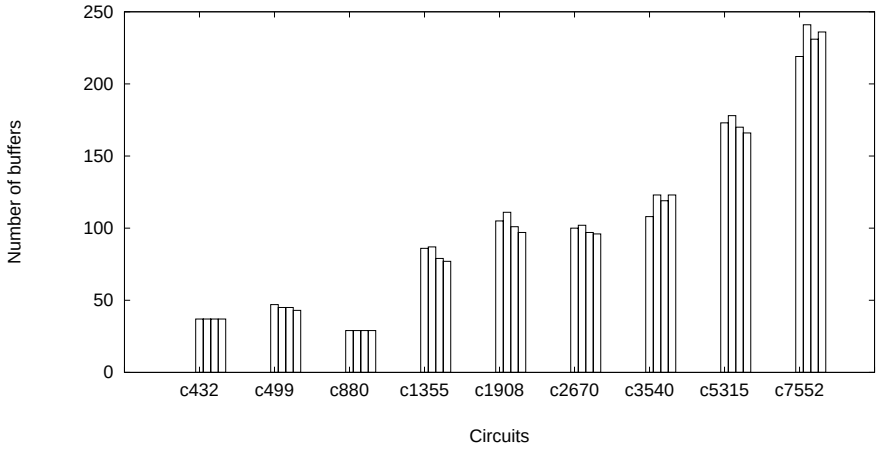
Table II shows the comparison of LAB and PBBI [5] algorithm with respect to number of buffers inserted and the CPU time. Referring to Figure 8, the experimental results are obtained with $l = 1$ and $p = 0.8$ for LAB. On an average, LAB gives 36% reduction in number of buffers inserted and 3× speed-up as compared to PBBI.

Table II. Comparison against a contemporary algorithm.

Ckt	PBBI [5]		LAB (New)			
	#Buffers	Time(s)	#Buffers	Time(s)	Reduction	Speed up
c432	61	0.4	37	1.5	39.3%	0.26x
c499	69	0.7	47	1.2	31.8%	0.58x
c880	48	1.9	29	0.7	39.5%	2.71x
c1355	143	3.9	78	3.6	45.4%	1.08x
c1908	137	16.9	96	7.6	29.9%	2.22x
c2670	187	63.2	94	10	49.7%	6.32x
c3540	202	85.2	109	21.6	46%	3.94x
c5315	269	194.4	164	21	39%	9.25x
c6288	508	182	433	149	14.91%	1.22x
c7552	282	429.4	208	49.5	26.2%	8.67x
Average					36.2%	3x



(a) Without back-off.



(b) With back-off.

Fig. 10. Comparison of cost performance of net-based insertion and look-ahead levels 0,1 and 2 for different test-cases.

CHAPTER VI

A FRAMEWORK TO OBTAIN EXACT OPTIMUM

In this chapter, we present an algorithm that solves the problem optimally. Since the worst-case running time is exponential, we call this algorithm as a subroutine only for circuit with no more than 30 gates.

A. Representation of a Candidate

A subgraph $G_I = (V_I, E_I)$ of G is called an *input subgraph* of G if $I(G_I) \subset I(G)$ and for every node $u \in V_I$, if there is a directed path from node v to u in G, then $v \in V_I$. Also, a subgraph $G_O = (V_O, E_O)$ of G, is called an *output subgraph* of G if $O(G_O) \subset O(G)$ and for every node $u \in V_O$, if there is a directed path from node u to v in G, then $v \in V_O$. Figure 11 shows an example input subgraph represented by the dotted area and its set of output nodes (h, n, e).

Consider the algorithms of [2] and [10] that find a optimal cost buffer assignment for a tree structure. In these algorithms, to represent a buffer assignment for a subtree rooted at a node v, Required Arrival Time only at node v is sufficient. This is because edges merge with each other when a tree is traversed from leaf nodes towards root node. But in case of a DAG, no matter how the graph is traversed, the edges merge as well as fork away from each other. Therefore, in this framework, assignment is represented as follos.

Consider a graph $G = (V, E)$, its input subgraph G_I and set of its output nodes $O(G_I) = \{v_1, v_2, \ldots, v_k\}$. Arrival time value $T(v_i, \alpha(G_I))$ for any node $v_i \in O(G_I)$ under an assignment $\alpha(G_I)$ can be computed since arrival time value at each primary input of G is known. Thus, provided that $O(G_I) \subset V_t$, a candidate $\alpha(G_I)$ can be

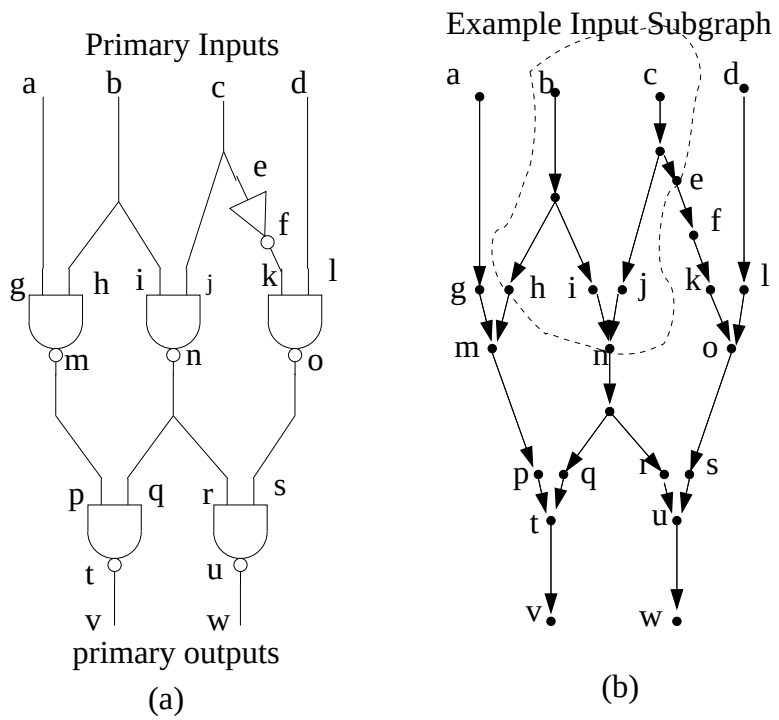


Fig. 11. Sample input subgraph and its set of output nodes $\{h, n, e\}$.

represented as a vector

$$W(\alpha(G_I)), T(v_1, \alpha(G_I)), \ldots, T(v_k, \alpha(G_I)).$$

Similarly, for output subgraph G_O and corresponding set of inputs $I(G_O) = \{v_1, v_2, \ldots, v_k\}$, if $I(G_O) \subset V_t$, any buffer assignment $\alpha(G_O)$ can be represented as a vector

$$W(\alpha(G_O)), T(v_1, \alpha(G_O)), \ldots, T(v_k, \alpha(G_O)).$$

$\alpha_1(G_I)$ *dominates* $\alpha_2(G_I)$ if $W(\alpha_2(G_I)) \geq W(\alpha_1(G_I))$, and $T(v, \alpha_2(G_I)) \geq T(v, \alpha_1(G_I))$ for each $v \in O(G_I)$. Similarly, $\alpha_1(G_O)$ *dominates* $\alpha_2(G_O)$ if $W(\alpha_2(G_O) \geq W(\alpha_1(G_O))$, and $Q(v, \alpha_2(G_O)) \leq Q(v, \alpha_1(G_O))$ for each $v \in I(G_O)$.

B. Propagating the Candidates

Similar to the dynamic programming algorithms for a single net [1, 2], our algorithm traverses the graph and computes the set of non-redundant assignments for the traversed subgraph.

To traverse the whole graph, we have a choice about the direction in which we can traverse the graph. We can start from the primary inputs where the input subgraph has no edges and grow the input subgraph to eventually traverse the whole circuit graph. Alternatively, we can start from the primary outputs and grow the output subgraph to eventually traverse the whole graph. Propagation using input subgraph is described in detail the following discussion. Since similar scenarios exist for propagation using output subgraph, it is addressed in short at the end of the section. Also, the criteria used to choose one method over the other for solving a particular problem are discussed.

Following is a simple flow of the algorithm with propagation using input subgraph.

```
1: G_I = (V_I, E_I), V_I = Set of primary inputs.
2: while G_I ≠ G do
3:   Select subcircuit G' to grow G_I.
4:   Propagate candidates of G_I over G'
5:   Update G_I and N(G_I).
6: end while
7: Solution = candidate with minimum cost in N(F)
```

Consider an input subgraph $G_I^a = (V_I^a, E_I^a)$ and a subgraph $G' = (V', E')$ such that $I(G') \subseteq O(G_I^a)$ and a subgraph $G_I^b = (V_I^b, E_I^b)$,where $E_I^b = E_I^a \cup E'$, is also an

input subgraph. Also, let $I(G') = \{i_1, \ldots, i_l\}$ and $O(G') = \{o_1, \ldots, o_m\}$.

Provided that $O(G_I^a) \subset V_t$, an assignment $\alpha(G_I^a)$ can be expressed in terms of arrival time values at nodes in $O(G_I^a)$. With this set of arrival time values and provided that $O(G') \subset V_t$, any buffer assignment $\alpha(G')$ for the subgraph G' can be expressed as $\alpha(G') = \{W(\alpha(G')), T(o_1, \alpha(G')), \ldots, T(o_m, \alpha(G'))\}$. Then the resultant assignment for G_I^b will be as follows:

$$W(\alpha(G_I^b)) = W(\alpha(G_I^a)) + W(\alpha(G'))$$

$$T(u, \alpha(G_I^b)) = \begin{cases} T(u, \alpha(G')) & \text{if } u \in O(G') \\ T(u, \alpha(G_I^a)) & \text{otherwise} \end{cases}$$

The subgraph G' chosen for growing the input subgraph can simply be a single net. But G' can also be chosen with added intelligence to facilitate lesser cardinality and efficient computation of set of resultant non-redundant candidates.

C. Computing Non-redundant Candidates for a Net

Consider a net represented as a routing tree η with leaf nodes $l_1, l_2, \ldots, l_k$. A 2-pin net is a special case tree with only one leaf.

When forward direction of propagation is used, RAT at leaf nodes of the net is not available. Hence (Q, C, W) framework of [2, 10] can not be used to compute the non-redundant candidates for the net. When propagating in forward direction, the framework we use to represent the candidate for the subtree η' rooted at node v is a vector $W(\alpha(\eta')), C(\alpha(\eta')), T(l_1, \alpha(\eta')), \ldots, T(l_k, \alpha(\eta'))$, where $T(l_i, \alpha(\eta'))$ is the delay from v to leaf l_i, and $C(\alpha(\eta'))$ is the downstream capacitance seen from node v under $\alpha(\eta')$.

Basic operation of this framework is same as (Q, C, W) framework except the computation of delay values while propagating the candidates. Similar to the (Q, C, W)

framework, candidates are generated by traversing the net from the leafs toward the root. Also, three basic operations during the traversal are adding a wire, adding a buffer and merging two subtrees [10]. When propagating the candidates in these scenarios, $W(\alpha(\eta'))$ and $C(\alpha(\eta'))$ are computed in the exact same manner as the (Q, C, W) framework. To propagate delay values however, while adding a wire or a buffer, delay values $T(l_i, \alpha(\eta'))$ are incremented by wire or buffer delay respectively. In merging operations, the delay values remain unchanged.

D. Pruning Techniques

Basic mechanism for pruning the assignments is comparing the assignments for the input or output subgraph against each other and deleting the inferior ones by the pruning criterion described earlier. In the following, we discuss pruning techniques that help to prune many more assignments effectively. We define following additional terms for node $u \in V_t$.

$$Q_{best}(u) = \max_{\alpha \in N(G)} \{Q(u, \alpha)\}$$

$$T_{best}(u) = \min_{\alpha \in N(G)} \{T(u, \alpha)\}$$

$$Q_{worst}(u) = \begin{cases} \text{RAT(u)} & \text{if } u \in V_{po} \\ max(T_{best}(u), Q(u, \alpha(\eta))) & \text{otherwise} \end{cases}$$

where η is the net rooted at node u, and $\alpha(\eta$ is such that $RAT(v) = Q_{worst}(v)$ for each leaf v of η and $W(\alpha(\eta)) = 0$.

- Q_{best} based filtering:

Consider an input subgraph G_I and its set of output nodes $O(G_I)$. Then, an assignment $\alpha(G_I)$ can be deleted if for some $v_i \in O(G_I)$:

$$T(v_i, \alpha(G_I)) > Q_{best}(v_i)$$

From the definition of Q_{best}, it can be easily seen that such an assignment certainly won't satisfy the timing requirements of the circuit. This condition is checked in propagation step itself and generation of such assignments having insufficient buffering is prevented. Also, whenever the input subgraph engulfs a primary output node v_i, all the assignments that survive Q_{best} based filtering satisfy the $RAT(v_i)$. Hence node v_i can be dropped from the candidate representation so that the exact value of $T(v_i)$ will be disregarded while comparing the assignments with each other resulting in further pruning.

- Q_{worst} based filtering:

Consider an input subgraph G_I and its set of output nodes $O(G_I)$. Consider an assignment $\alpha(G_I)$ such that for some $V_i \in O(G_I)$,

$$T(v_i, \alpha(G_I)) < Q_{worst}(v_i)$$

This condition suggests that sufficient buffers have been added in the transitive fan-in cone of node v_i under $\alpha(G_I)$ and even no buffering in the net rooted at node v_i is sufficient to meet the timing requirements. Thus, $\alpha(G_I)$ is propagated only with the minimum cost assignment on the net rooted at node v_i and generation of unnecessarily buffered assignments is prevented.

- Cost stepping:

In stead of generating all the candidates, we put a limit on cost of a candidate such that the candidates with cost more than the limit will not be generated in the current

iteration of candidate propagation. If the solution is not found in the current iteration, the limit is increased by a certain amount and the candidates under the new limit are evaluated. This is repeated till a solution is found. This simple technique prevents the computation of assignments with cost more than optimal.

- Computing Q_{best} and T_{best}:

To find Q_{best} we run van Ginneken's algorithm [1] on each net from primary output towards primary input, using the RAT for all primary output nodes. Unlike Q_{best}, computation of T_{best} is affected by the re-convergence in the circuit. Therefore, we need to keep track of re-convergences in the circuit to compute exact values of T_{best}. Alternatively an approximate value of T_{best} which is lesser than the exact value can be computed by ignoring the effect of re-convergences and following a procedure similar to Van Ginneken's algorithm.

E. Propagation Using Output Subgraph

Propagation using output subgraph is similar to that using input subgraph described in detail above. In this case, initially $I(G_O)$ is the set of primary outputs and eventually after traversing the whole $G_O = G$.

Consider an output subgraph $G_O^a = (V_O^a, E_O^a)$ and a subgraph $G' = (V', E')$ such that $O(G') \subseteq I(G_I^a)$ and a subgraph $G_O^a = (V_O^a, E_O^a)$, where $E_O^b = E_O^a \cup E'$, is also an output subgraph. Also, $I(G_O^a) \subset V_t$ and $I(G') \subset V_t$. Also, let $I(G') = \{i_1, \ldots, i_l\}$ and $O(G') = \{o_1, \ldots, o_m\}$.

Consider an assignment $\alpha(G_O^a)$ expressed in terms of required arrival time values at nodes in $I(G_O^a)$. With this set of required arrival time values and provided that $I(G') \subset V_t$, any buffer assignment $\alpha(G')$ for the subgraph G' can be expressed as $\alpha(G') = \{W(\alpha(G')), Q(i_1, \alpha(G')), \ldots, Q(i_m, \alpha(G'))\}$. Then the resultant assignment

for G_O^b is computed as follows:

$$W(\alpha(G_O^b)) = W(\alpha(G_O^a)) + W(\alpha(G'))$$

$$Q(u, \alpha(G_O^b)) = \begin{cases} Q(u, \alpha(G')) & \text{if } u \in I(G') \\ Q(u, \alpha(G_O^a)) & \text{otherwise} \end{cases}$$

To compute nonredundant candidates for a net, (Q, C, W) framework of [2, 10] is used. Counterpart of Q_{best} based filtering is T_{best} based filtering. Note that the effectiveness of T_{best} based filtering is reduced because of using an approximate value of T_{best} which is lesser than its exact value, but it does not result in deletion of an assignment that should not have been deleted. Also note that there is no counterpart of the Q_{worst} based filtering.

F. Selecting Method of Propagation

Note that the circuit DAG could be viewed to be made up of 2 types of trees. First type is the interconnect routing trees of a multi-pin net (multi-pin tree). The root of a such a multi-pin tree is towards the primary input side and leaves are towards primary output side. Second type is the tree formed by 2-pin nets (2-pin tree) and input-to-output edges within the gates merging at gate output nodes. The root of such a 2-pin tree is towards the primary output side and leaves are towards primary input side. Also, every gate output node that is a root of some multi-pin tree is also a root of some 2-pin tree.

While propagating the candidates, merging of edges is desired than forking because forking causes longer vectors to represent the candidate and consequently larger cardinality of set of non-redundant candidates. Thus, if input subgraph is used for propagation, edges in multi-pin trees fork away from each other and cause the number of candidates to blow up. Similarly, if output subgraph is used for propagation, 2-pin

trees cause the number of candidates to blow up.

Thus, the relative sizes of the multi-pin and 2-pin trees provide the criteria to choose one method of propagation over the other. Disadvantage of using input subgraph for propagation is it can not handle bigger multi-pin nets effectively. Disadvantage of using output subgraph for propagation is less effective pruning. In general, if number of buffer locations in 2-pin nets is more than that in the multi-pin nets, we choose input subgraph for propagating the candidates.

G. Practicality

For this framework, the worst-case running time is exponential and hence this framework can not be run for circuits having more than 30 gates. But it is likely to improve the cost performance when used in conjunction with path based algorithms like [5]. Initially, such an algorithm can be used to insert buffers on critical paths and consequently partition the bigger circuit along these critical paths to arrive at smaller subcircuits. Then the proposed framework can be used to obtain optimum buffering for smaller subcircuits.

CHAPTER VII

BOOSTER MODELING AND INSERTION

A device named booster was introduced in [17] to drive long on-chip interconnect wires. Compared to traditional buffers, boosters sometimes offer better delay [17]. Furthermore, boosters offer unique advantages as they can be used for bidirectional wires, and multiple boosters can be used to drive the same wire without risking the possibility of short circuits.

Models to estimate delay of buffered interconnect and techniques to optimally insert buffers are mature but there is no delay model or algorithm for boosters. Some guidelines on where to insert boosters on a uniform interconnect are given in [17], but many factors, such as driver strength, non-uniform interconnect parasitic and sink capacitance are ignored. Therefore, in order to use boosters efficiently, it is necessary to develop a reasonable delay model and corresponding insertion algorithms.

In this chapter, a delay model is proposed, based on Elmore delay [18] and tree link partitioning [19], to estimate interconnect delay when one or multiple boosters are present on a 2-pin net. In section B, booster insertion algorithm, that uses newly developed delay model, is proposed. The algorithm adopts the dynamic programming approach to find optimal booster insertion in polynomial time based on the delay model. The experimental results of the new models and the new algorithm are presented in section C. It shows that the proposed delay model is sufficiently accurate for physical synthesis and proposed algorithm is efficient.

A. Delay Models

Booster is attached to an interconnect wire, as shown in Figure 12(a). Figure 12(b) shows the stages through which the booster goes when the interconnect switches from

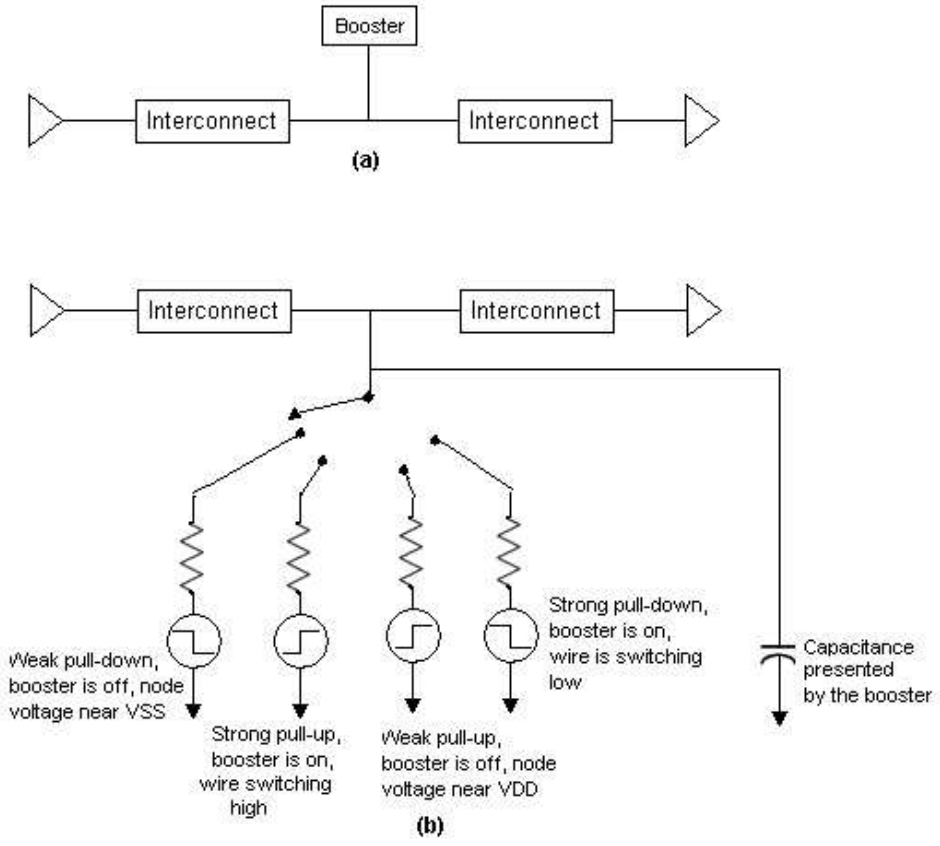


Fig. 12. (a) Booster placed in an interconnect. (b) Operation of booster.

logic low to logic high and back to low. When the signal rises on the interconnect, the booster detects the rise early and applies a strong pull-up to boost the signal strength. When signal potential reaches close to Vdd, booster turns off the strong pull-up and applies a weak pull-up to just sustain the signal level. Similarly, when the signal falls on the interconnect, booster detects the fall early and applies a strong pull-down on the wire. When the signal potential reaches close to Vss, booster turns off strong pull-down and applies a weak pull-down. The weak pull-up and pull-down, applied by the booster when switching is not occurring on the signal gives a better noise immunity. Whenever booster is said to be *on* while applying a strong pull-up/pull-

down, and said to be *off* otherwise. Driving strength of the booster is different in its on and off states but its input capacitance is always felt on the interconnect.

There are several differences between boosters and traditional buffers/repeaters:

- Boosters do not distinguish upstream or downstream. Therefore we can use boosters to drive bi-directional wires.
- Boosters are not always active. Therefore we can use multiple boosters to drive a net having a non-tree topology without the risk of short circuits.
- Boosters do not cut the interconnect wire. Therefore, the intrinsic delay that appears in buffered interconnects can be avoided by using boosters.

1. Single Booster

To model the behavior of boosters, its operation is divided into two states, its on state and its off state, and interconnect delay for these stages are calculated separately. The node voltages are approximated as piecewise linear and this approximation is applied to combine delays of separate stages and find the final delay at the sink.

Figure 13(a) shows a booster placed on a 2-pin net whose parasitic is represented by a π model. Before the booster is triggered on, its driving strength is ignored and as shown in Figure 13(b), it is represented on the interconnect just by its input capacitance.

For second stage of the booster's operation, i.e. the case after booster is triggered on, two different models are proposed here. In the first model, after the booster is triggered on, as shown in Figure 13(c), it is represented by a driving resistor corresponding to its strong pull-up or pull-down. In the second model, as shown in Figure 13(d), a further simplifying assumption is made to combine two drivers on the

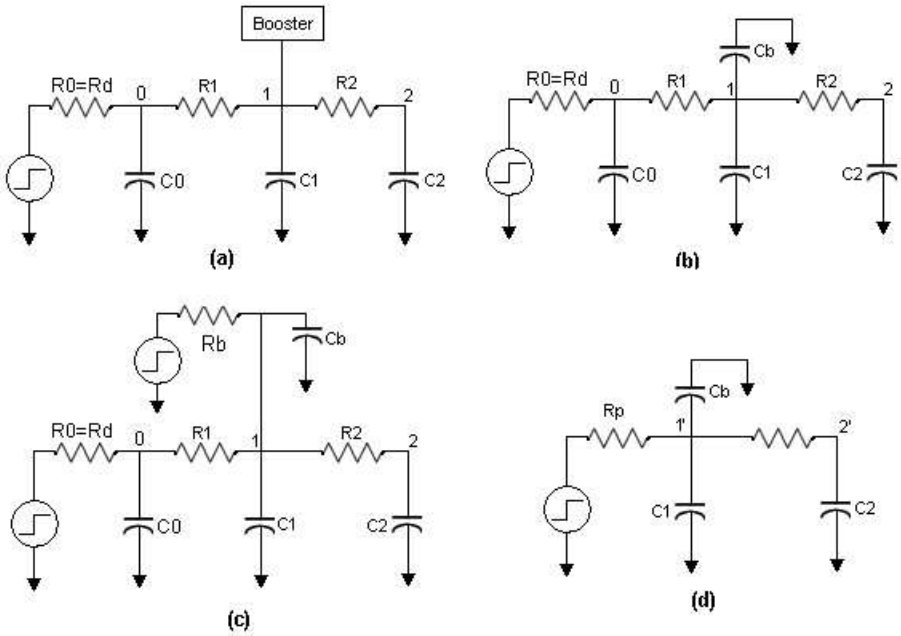


Fig. 13. (a) Original circuit. (b) Circuit before triggering of booster. (c) Circuit after booster has triggered for method 1. (d) Circuit after booster has triggered for method 2.

interconnect into a single driver. This resultant driver R_p is equivalent to a parallel combination of main input side driver $(R_d + R_1)$ and booster driving resistance R_b.

Furthermore, the voltage of every node in the circuit is approximated as a piecewise linear function as shown in Figure 14. Consider the case of rising signal. Before the booster is triggered on, the voltage at a node increases with a certain slope from Vss to Vdd. When the booster is triggered on, the voltage at the node starts increasing at a higher slope because of booster's additional drive.

In the method (Method 1) of Figure 13(c), since there are two drivers driving the same interconnect, we use the tree-link partitioning technique of [19] to find the Elmore delay. Second method (Method 2), i.e. the method of Figure 13(d), is a further approximation, where multiple drivers on the interconnect are combined

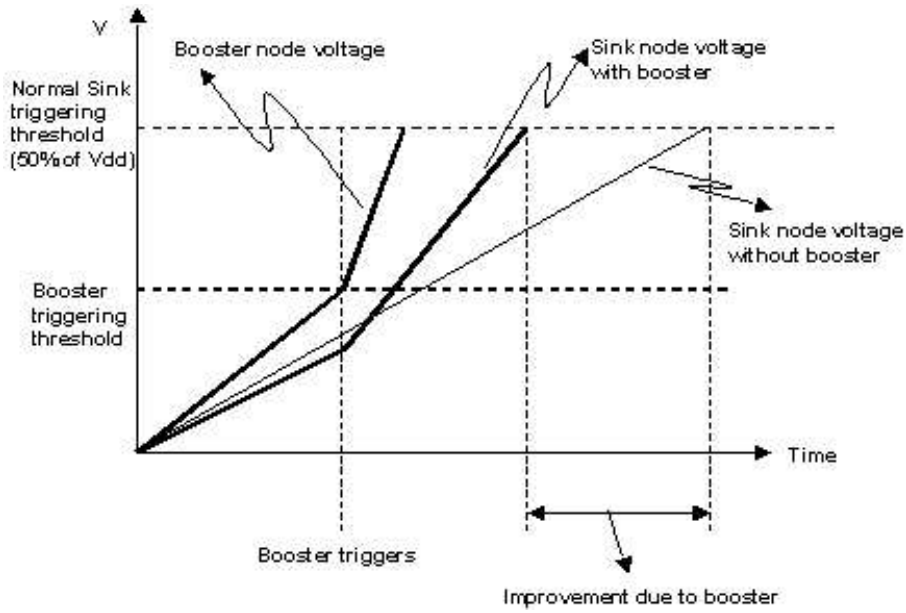


Fig. 14. Effect of booster on sink node voltage.

into one driver. Since there is only one driver in this method, simple Elmore delay calculation applies. Table III describes the terminology used in rest of this chapter.

Threshold voltage of a booster is set below threshold voltage of a traditional buffer so that booster can detect the signal transition early and boost the signal strength according to the direction of switching. For example, threshold voltage of a booster can be 40% and 60% Vdd, for rising and falling transitions respectively. The scale-down factor t_f is defined as $t_f = t_p/50$. For example, if triggering threshold of the booster is 40% of Vdd, then $t_f = 0.8$.

The delay for the booster node to reach triggering threshold is calculated as

$$T_{V1=TTV} = t_f(R_0(C_0 + C_1 + C_2 + C_b) + R_1(C_1 + C_2 + C_b))$$

Table III. Booster modeling notations.

R_d	Driver resistance
R_b	Driving resistance of booster
C_b	Input capacitance of booster
R_p	New driving resistance after booster triggers
R_i	Resistance between node $i-1$ and i
C_i	Ground capacitance at node i
$T_{(Vi=V)}$	Time required for node i to reach voltage V
TTV	Triggering Threshold Voltage of the booster
D_b	Triggering time of the booster
D_{S1}	Delay to sink considering circuit before booster triggers
D_{S2}	Delay to sink considering circuit after booster triggers
t_f	Scale-down factor for deciding booster triggering time
t_p	Booster threshold voltage as a percentage of Vdd
t_i	Intrinsic delay of booster

The triggering time of the booster is given as

$$D_b = T_{V1=TTV} + t_i$$

Elmore delay at the sink considering the circuit before booster triggers i.e. Figure 13(b) will be

$$D_{S1} = R_0(C_0 + C_1 + C_2 + C_b) + R_1(C_1 + C_2 + C_b) + R_2C_2$$

In first method in which two drivers are considered to be driving the interconnect, i.e. the circuit in Figure 13(c), treating R_1 as a link, the delay according to Tree Link

Partitioning method is given as

$$D_{S2} = R_b(C_1 + C_2 + C_b) + R_2C_2 + (\frac{R_0C_0 - R_b(C_1 + C_2 + C_b)}{R_0 + R_1 + R_b} \times R_b)$$

In second method in which two drivers are combined into a single driver, i.e. the circuit in Figure 13(c), Elmore delay at the sink is given as

$$D_{S2} = R_p(C_1 + C_2 + C_b) + R_2C_2$$

Now, according to piecewise linear approximation, the delay at the sink in presence of a booster is given as,

$$T_{(V2=0.5VDD)} = D_b + (1 - D_b/D_{S1})D_{S2}$$

Note that this model assumes that once the booster is turned on, it remains on till the signal level reaches a full rail value. Also, the effect of weak pull-up and pull-down when booster is off is ignored in this model.

2. Multiple Boosters

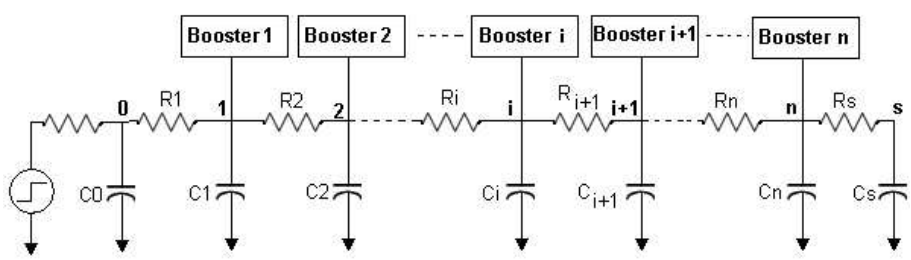


Fig. 15. Multiple boosters on a single wire.

Figure 15 shows the case when multiple boosters are inserted in the interconnect. Method 2 of single booster model described above, i.e. combining multiple drivers

into one, is extended to model multiple boosters on 2-pin interconnect. The only difference is this case from that of single booster is, instead of a sink that triggers at normal 50% threshold, another booster triggering at early threshold can appear in the downstream of one booster. As the triggering threshold of all the boosters is same, the situation can be simplified as shown in Figure 16.

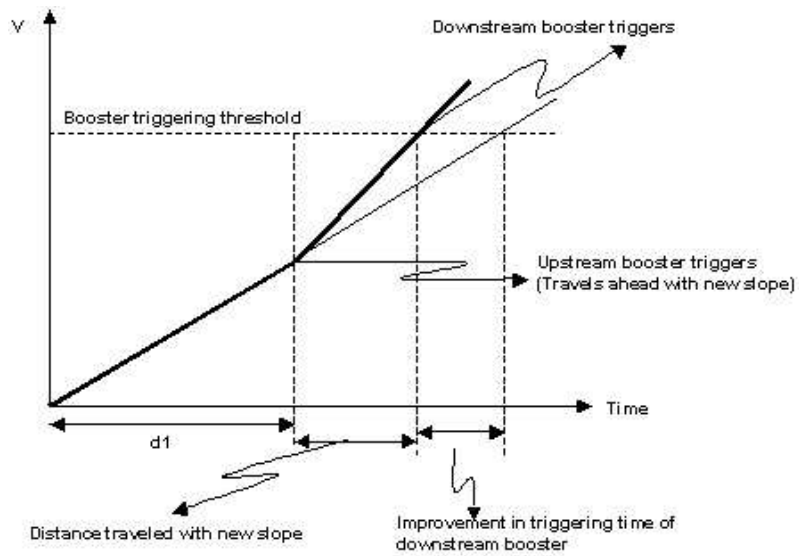


Fig. 16. Effect of upstream booster on downstream booster node.

Consider two adjacent boosters at nodes i and $i+1$. To calculate the triggering time of the downstream booster in presence of an upstream booster, let

d_0 = Time when upstream booster node reaches threshold

d_1 = Triggering time of upstream booster, $d_0 + t_i$

d_2 = Elmore delay from node i to node $i+1$

$\delta = R_p$(Total downstream capacitance at node i)

Then the improved time when downstream booster reaches threshold is given as

Improved downstream booster triggering time =

$= d_1 + (\text{distance traveled with new slope}) \times (\text{new slope})$

$= d_1 + ((1 - \frac{d_1}{d_0 + (t_f \times d_2)}) \times t_p)(\frac{d_2 + \delta}{50})$

Note that only adjacent upstream booster is considered while calculating the improved triggering time of a downstream booster. In other words, we are making a assumption that booster at node i is on till node booster at node $i + 1$ triggers and the rest of the upstream boosters are off when node i booster triggers. As we proceed from the source towards sink, we finally get the triggering time of the last booster. While proceeding from the last booster to the sink, the threshold time is calculated according to equations in subsection 1.

B. Insertion

Using new delay models and following a dynamic programming approach [1], we can find placement of boosters on a single line such that delay from source to sink is minimized. Consider a general situation of Figure 16 where nodes 1 to n represent possible booster positions on the interconnect wire. The algorithm represents the candidate solutions at any node i by a (Q, C, j) triplet, where Q is the slack at that node, C is the downstream capacitance at that node i, and j is the nearest downstream booster node which can take the values $i \ldots n$. The algorithm starts from the sink and traverses towards the source, calculating non-redundant set of candidate solutions in terms of (Q, C, j) triplets and pruning inferior solutions on the way. A candidate (Q_1, C_1, i) is redundant if there is another candidate (Q_2, C_2, i) such that $Q_1 \leq Q_2$ and $C_1 \geq C_2$. Since there are at most n values of C, it is easy to see that the maximum possible number of non-redundant candidates at any node

is at most n^2. Whereas, a brute force method will need to consider all 2^n possible booster placement combinations. The algorithm works as follows. Initially for the sink, we have one candidate solution $(Q, C, \emptyset)$, where Q is the required arrival time at the sink and C is the sink capacitance. The algorithm then moves towards the source calculating solutions for each node i considering the options of presence and absence of the booster at node i and using the model in the previous section. After pruning the redundant candidates, we move upwards to process booster position $i - 1$. The solution having the maximum Q among the final solutions at the source node gives the minimal delay. The time complexity of the algorithm is bounded by $O(n^3)$.

C. Experimental Results

The experimental results for the models described in section A are presented here in comparison to SPICE. Two different boosters with different driving resistance and switching thresholds were used for these experiments.

1. Single Booster

The "Booster Placement" field in Table IV indicates the location of the booster from the source as a fraction of total wire length. As seen from the representative experimental cases in Table IV, as the position of the booster is varied, the trend in change of delay predicted by the model matches closely with SPICE though not exactly. Also, the difference in the delay values predicted by the model and that by SPICE for the same booster placement is comparable to the difference between Elmore delay and the delay predicted by SPICE for general interconnects [20].

Table V represents the average readings for 10 random cases in our experiments, each case having 10 possible booster locations and number of boosters limited to one.

The average percentile difference between SPICE delay values of the optimal position predicted by SPICE and that predicted by the models is very less. For uniform wire, the performance of Method 1 and 2 has no significant difference. For non-uniform wires, Method-1 performs better than Method-2 as the loading of the booster due to upstream capacitance is ignored in the Method-2. The time taken by the models to run is significantly less than that taken by SPICE.

2. Multiple Boosters

For these experiments, the boosters are placed equidistantly on a single wire for the sake of convenience though this is not an optimal fashion to place the boosters. As seen from Table VI, the trend of change in delay for the model, as more boosters are placed on the wire, follows SPICE readings closely.

Table IV. Single booster at different positions on a single wire.

Booster Placement	Delay in model(ps)		Delay in SPICE (ps)
	Method 1	Method 2	
Sample case: Uniform Wire			
0.1L	919	916	671
0.2L	887	882	662
0.3L	863	857	659
0.4L	847	841	666
0.5L	841	834	683
0.6L	842	836	705
0.7L	848	842	734
0.8L	859	853	759
0.9L	871	865	778
Sample case: Non-uniform Wire			
0.1L	696	687	538
0.2L	688	661	537
0.3L	673	641	542
0.4L	667	635	564
0.5L	670	637	595
0.6L	678	644	630
0.7L	686	648	653
0.8L	696	651	670
0.9L	704	653	682

Table V. Booster delay models compared with SPICE simulations.

	Method 1	Method 2	SPICE
Difference from optimal (Uniform wire)	2.38%	2.73%	-
Difference from optimal (Non-uniform wire)	1.8%	7.2%	-
CPU Time	0.01s	0.01s	24s

Table VI. Multiple boosters on a single wire.

Number of boosters	Delay in model (ps)	Delay in SPICE (ps)
Representative Case		
1	878	726
2	828	723
3	819	737
4	828	752

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

This thesis proposed a buffer insertion algorithm called LAB (Lookahead and Backoff) for combinational circuits such that the timing requirements are satisfied and buffer cost is minimized. Experimental results on ISCAS85 circuits show that compared to a recently proposed algorithm in the research community, Lookahead-backoff algorithm can reduce the number of buffers by 15% to 50% and runs up to 9x faster on bigger ISCAS85 benchmark circuits.

The main contributions of this work are the ideas of look-ahead and back-off that can be employed to guide the solution search efficiently. These ideas are very simple in terms of implementation and highly flexible in terms of user's constraints on quality of the solution and run time. These ideas provide a general infrastructure to guide the solution search for combinational optimization problems and can be applied to optimization problems other than buffer insertion.

Another strength of LAB is that it builds on top of well-researched net-level algorithms and does not have any restrictive modeling assumptions. Consequently, any future advances in the dynamic programming based net-level algorithms can be seamlessly integrated into LAB.

As a part of future work, testing LAB with even bigger circuits and extending it with further speed-up techniques is essential to make the algorithm more robust and efficient. Also, to broaden its applicability, it can be extended as a general framework for gate sizing along with buffer insertion.

Along with LAB, two of the earlier works, namely exact optimum framework for buffer insertion problem and delay models and insertion algorithm for boosters are also presented in this work. In the framework proposed to obtain exact optimum solution

to buffer insertion problem, the worst-case running time is exponential and hence this framework can not be run for bigger circuits. But it can be used in conjunction with path based algorithms like [5] to improve the cost-performance. Experimental results for the proposed booster delay models and insertion algorithm show that proposed models closely follow the SPICE predictions, and are suitable for physical synthesis. But only 2-pin nets were considered for these delay models and future work is needed to extend these models for any general interconnect topology.

REFERENCES

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

[2] J. Lillis, C. K. Cheng and T.-T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE Trans. Solid-State Circuits,* vol. 31, no. 3, pp. 437–447, March 1996.

[3] I-Min Liu, A. Aziz, D.F. Wong and H. Zhou, "An efficient buffer insertion algorithm for large networks based on Lagrangian relaxation," in *Proc. Int. Conf. on Computer Design*, 1999, pp. 210–215.

[4] I.-M. Liu, A. Aziz, and D. F. Wong, "Meeting delay constraints in DSM by minimal repeater insertion," in *Proc. Design Automation and Test in Europe*, 2000, pp. 436–441.

[5] C. Sze, C. Alpert, J. Hu and W. Shi, "Path-based buffer insertion," in *Proc. ACM/IEEE Design Automation Conf.*, 2005, pp. 509–514.

[6] R. Chen and H. Zhou, "Efficient Algorithms for Buffer Insertion in General Circuits Based on Network Flow," in *Proc. Int. Conf. on Computer Aided Design*, 2005, pp. 509–514.

[7] P. Saxena, N. Menezes, P. Cocchini, and D. A. Kirkpatrick, "Repeater scaling and its impact on CAD," *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems* , vol. 23, no. 4, pp. 451–463, April 2004.

[8] C. J. Alpert and A. Devgan. "Wire segmenting for improved buffer insertion," in *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 588–593.

[9] W. Shi and Z. Li, "An $O(n \log n)$ time algorithm for optimal buffer insertion," in *Proc. ACM/IEEE Design Automation Conf.*, 2003, pp. 580–585.

[10] W. Shi, Z. Li and C.J. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," in *Proc. Asia and South Pacific Design Automation Conf.*, 2004, pp. 609–614.

[11] Y. Zhang, Q. Zhou, X. Hong and Y. Cai, "Path-based timing optimization by buffer insertion with accurate delay model", in *Proc. 5th International Conference on ASIC*, Vol.1, pp. 89–92, Oct. 2003.

[12] Y. Jiang, S. Sapatnekar, C. Bamji and J. Kim, "Interleaving buffer insertion and transistor sizing into a single optimization", *IEEE Transactions on VLSI Systems*, vol. 6, no. 4, pp. 625–633, Dec. 1998.

[13] K.S.Lowe and P.G. Gulak, "A joint gate sizing and buffer insertion method for optimizing delay and power in CMOS and BiCMOS combinational logic", *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems* , vol. 17, no. 5, pp. 419–434, May 1998.

[14] S. Lin and M. Marek-Sadowska, "A fast and efficient algorithm for determining fanout tree in large networks", in *Proc. of EDAC*, Feb 1991, pp. 539–544.

[15] H. Zhou, D. F. Wong, I. M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations", *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems* , vol. 19, no. 7, pp. 819–824, July 2000.

[16] C. C. N. Chu and D. F. Wong. "A quadratic programming approach to simultaneous buffer insertion/sizing and wire sizing", *IEEE Trans. on Computer Aided*

Design of Integrated Circuits and Systems, vol. 18, no. 6, pp. 787–798, Sept. 1999.

[17] A. Nalamalpu, S. Srinivasan and W. Burleson, "Boosters for driving long on-chip interconnects: Design issues, interconnect synthesis and comparison with repeaters", *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, vol. 21, no.1, pp. 50–62, Jan. 2002.

[18] W. C. Elmore, "The transient response of damped linear networks", *Journal of Applied Physics* vol. 19, pp. 55–63, Jan 1948.

[19] P. K. Chan and K. Karplus, "Computing signal delay in general RC networks by tree/link partitioning", *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, vol. 9, no. 8, pp. 898–902, Aug. 1990.

[20] R. Gupta, B. Tutuianu and L.T. Pileggi, "The Elmore delay as a bound for RC trees with generalized input signals", *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, vol. 16, no. 1, pp. 95–104, Jan. 1997.

VITA

Mandar Waghmode attended Government College of Engineering, Pune, India and received his Bachelor of Engineering degree in Instrumentation and Control from University of Pune, India. After working in the field of VLSI logic design and simulation for three years in Pune, he enrolled in Texas A&M University, College Station, Texas, USA, in the Master of Science program in Computer Engineering. During the school year 2004-2005, he also served as a Teaching and Research Assistant in the Department of Electrical and Computer Engineering at Texas A&M University.

Mandar Waghmode
Neelkanth, Shivaji Chowk, Indapur, 413106, India