

Routing Tree Construction Under Fixed Buffer Locations

Jason Cong and Xin Yuan
 Department of Computer Science
 University of California, Los Angeles, CA 90095
 Email: {cong,yuanxin}@cs.ucla.edu

Abstract

Modern high performance design requires using a large number of buffers. In practice, buffers are organized into buffer blocks and planned in the early stages of design process [1]. Thus, the locations of buffer blocks are usually fixed prior to routing tree construction. In this paper we present the first algorithm for simultaneous routing tree construction and buffer insertion for multiple-pin nets under fixed buffer locations. Given a source and n sinks of a net, the required arrival time associated with each sink, and m buffers with fixed locations, our algorithm can construct a routing tree for this net with possible insertion of buffers at given locations such that the required arrival time at the source is maximized. Experimental results show that our algorithm is efficient to handle fixed buffer location constraints and can also be used for routing tree construction without buffer insertion. Moreover, it can handle obstacles and congestion which will benefit its adaption in a global router. Compared to the well-known BA-tree algorithm [2] followed by a post-processing step for handling fixed buffer location constraints, our algorithm outperforms it by up to 46% in terms of delay while using comparative wirelength.

1 Introduction

Rapid scaling of IC technology leads to much smaller and faster devices, but more resistive interconnects with a large coupling capacitance. This makes interconnect delay a dominant factor in determining the overall performance [3]. Many interconnect performance optimization techniques have been studied extensively, such as topology construction, buffer insertion, driver sizing, wire sizing and spacing (see [4] for a tutorial). Among them, buffer insertion is the most effective way to improve interconnect performance. It has been shown that without buffer insertion, the interconnect delay for a wire of length l increases at the rate of $O(l^2)$ without wiresizing, or $O(l\sqrt{l})$ with optimal wiresizing, but it only increases linearly under proper buffer insertion [5]. As the intrinsic delay of a buffer decreases and the chip dimension increases, a large number of buffers are needed to be inserted for a high-performance design (e.g., close to 800,000 for 70nm technology, as estimated in [3]). These buffers need to be planned as early as possible to ensure timing closure and design convergence because a large number of buffer insertions may greatly change the floorplan, complicate the power/ground routing and make it difficult to incorporate hard IP cores. Currently, in industry, buffer planning is mainly done manually during floorplan stage due

Copyright ©2000 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM Inc., fax +1 (212) 869-0481, or (permissions@acm.org).

to lack of good tools and methodologies. Recently Cong *et al.* proposed the BBP algorithm [1] which can automatically generate buffer blocks for interconnect optimization during floorplanning. As an example, we show a floorplan with buffer block planning in Figure 1 (for simplicity we only show the pins of one net). Once the buffer planning is done in either way, the locations of buffers are fixed. It imposes constraints for routing tree construction when buffer insertion is needed; that is, buffer insertion can only occur at the given places. We call the problem *routing tree construction under fixed buffer locations* (RFB problem).

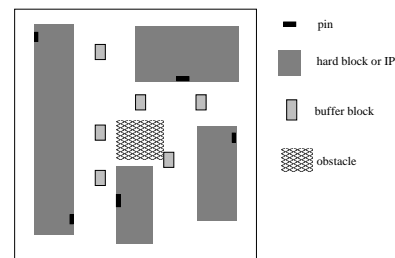


Figure 1: An example of floorplan with buffer block planning

An important early work on buffer insertion was done by van Ginneken who presented a polynomial time algorithm [6] using dynamic programming for delay optimal buffer insertion on a given tree topology. After that, several algorithms on simultaneous routing and buffer insertion have been proposed, such as the algorithm in [7] which combined the P-Tree algorithm with buffer insertion, and the BA-tree algorithm [2] which combined the A-tree algorithm [8] with buffer insertion in a bottom-up fashion. Though these algorithms can efficiently construct a buffered routing tree, they can not handle any constraint on buffer locations, i.e., they assume that a buffer can be inserted at any place, which may be impractical due to the pre-placed buffer blocks and the restriction imposed by the design hierarchy mentioned above. Moreover, they can not handle obstacles and congestion in the routing region. The GA-tree algorithm [9] is the only performance-driven routing tree construction algorithm we know that can handle obstacles and routing congestion. But it does not consider buffer insertion.

Recently Zhou *et al.* proposed an algorithm [10] which deals with restriction on buffer locations during simultaneous routing and buffer insertion. The restriction is modeled by the presence of a set of blocks that allow wires to pass but are obstacles for buffer insertion. Their algorithm works for 2-pin nets only. It can find a path from source to sink such that the Elmore delay is minimized using feasible buffer insertion. Since this algorithm can only handle 2-pin nets, its application is limited.

In this paper, we present an algorithm named RMP (*Recursively Merging and Pruning*), to address the RFB prob-

lem. It uses the dynamic programming approach to construct the routing tree with buffer insertion under fixed buffer locations in a bottom-up fashion. Our algorithm can construct a performance-driven topology with possible insertion of buffers at given locations. It can also handle obstacles and routing congestion.

The remainder of this paper is organized as follows. Section 2 presents some preliminaries and the problem formulation. Our algorithm is described in Section 3. The experimental results are shown in Section 4 followed by conclusions and future work in Section 5.

2 Preliminaries and Problem Formulation

The task of our algorithm is to construct a routing tree for a net under given fixed buffer locations such that the required arrival time at the source is maximized. The routing tree is constructed on a *routing graph* derived from the floorplan solution. Given a floorplan with buffer block planning, before a net is routed, its routing graph is set up based on the floorplan and current routing status. A buffer block that has unused buffers (buffers not being used by other nets), and is in the bounding box of a net is regarded as an available buffer of this net. In the routing graph, source, sinks and available buffers of this net are connected by edges. These edges are not allowed to cross obstacles, and some edges may be removed due to high routing congestion over it. Therefore the routing tree constructed in the routing graph can avoid obstacles and high routing congestion areas. After a net is routed, the number of unused buffers in buffer blocks and the overall routing congestion information will be updated.

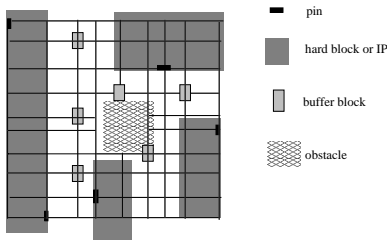


Figure 2: A routing graph derived from a floorplan with buffer block planning

Figure 2 shows a simple 2-dimension Hanan grid-based routing graph induced on pins and buffer blocks generated from the floorplan shown in Figure 1. Of course the routing graph may not be restricted to the Hanan grid on a single layer. In practice, we can use a multi-layer routing graph derived from either the channel intersection graph or an underlying routing grid for a global router. In this paper, we consider the routing tree construction under fixed buffer locations for a single net only. However, our algorithm can be used in a global router repeatedly to route multiple nets.

We classify the nodes in the routing graph $G = (V, E)$ into four types:

source node : node occupied by the source of the net

sink node : node occupied by a sink of the net

buffer node : node occupied by an available buffer

vacant node : node other than the above three types

The *routing tree construction under fixed buffer locations* (RFB) problem is stated as follows.

Given: a routing graph $G = (V, E)$,

- 1) a source node $s_0 \in V$ and n sink nodes $s_1, s_2, \dots, s_n \in V$ of a net S ,
- 2) a required arrival time associated with each $s_i, (1 \leq i \leq n)$
- 3) a set of buffer nodes $b_1, b_2, \dots, b_m \in V$.

Find: a buffered routing tree that spans S with necessary buffer insertions at given buffer nodes.

Objective: maximize the required arrival time of the source s_0 .

As in most previous works on interconnect layout optimization, e.g., [2], we adopt the widely used Elmore delay model for interconnect and switch-level RC model for buffers.

3 RMP Algorithm

3.1 Algorithm Description

Our algorithm is based on the dynamic programming approach combined with a bottom-up tree construction. The algorithm generates a set of subtrees from the sinks and gradually expands and merges them until a complete routing tree with the best performance is produced.

There are a few important differences between our algorithm and other bottom-up dynamic programming-based tree construction algorithms: i) the sets of subtrees may not be disjoint, ii) multiple subtrees may be generated at a node, and iii) our algorithm works on a routing graph.

During the subtree generation process, expansion and merging are realized by generating labels for the nodes. The algorithm may generate a set of labels for each node in the routing graph $G = (V, E)$. A 4-tuple label L of node w is in the form of (cap, rat, RE, buf) and represents a subtree $T_L(w)$ rooted at node w specified by the following parameters.

- *cap*: load capacitance at node w in subtree $T_L(w)$
- *rat*: required arrival time at node w in subtree $T_L(w)$
- *RE*: reachable sink set of subtree $T_L(w)$. It is the set of sinks contained by subtree $T_L(w)$.
- *buf*: whether or not a buffer is inserted at node w in subtree $T_L(w)$. *buf*=1 if a buffer is inserted; otherwise 0. If *buf*=1, then add C_{bi} as an auxiliary field. C_{bi} is the downstream load capacitance of subtree $T_L(w)$ without buffer insertion at node w .

Initially each node's label set contains one 4-tuple label:

- sink node s_i : label $(C_{Li}, rat_i, \{s_i\}, 0)$. rat_i is the required arrival time of the sink s_i . C_{Li} is the load capacitance of sink s_i .
- buffer node b_i : label $(C_{buf_i}, +\infty, \phi, 1)$. C_{buf_i} is the input capacitance of buffer b_i . Since no subtree is generated initially, C_{bi} is set to 0.
- source node and vacant node w : label $(0, +\infty, \phi, 0)$.

We also define a working queue Q which includes all the labels to be expanded. Initially the working queue contains the label of every sink, i.e.,

$$Q = \{(C_{Li}, rat_i, \{s_i\}, 0), \forall s_i \in sink_set\}$$

In our algorithm, each time, a label with the maximal rat is fetched from Q for expansion. In order to handle multiple-sink nets, we allow subtrees to be merged to a new subtree and keep the original separate subtrees at the same time. When we expand a label $X = (C_1, rat_1, RE_1, buf_1)$ of node w to node u via edge $(w, u) \in E$, we try to merge label X with each label $Y = (C_2, rat_2, RE_2, buf_2)$ in node u 's label set. But in order to avoid duplicate connection, we only allow a merge between two labels that have no common sink nodes in their reachable sets. If a merge is allowed, a new label $S = (C_3, rat_3, RE_3, buf_3)$ for node u is generated where

$$\begin{aligned} C_3 &= C_1 + C_2 + C_{e_{wu}} \\ rat_3 &= \text{MIN}(rat_1 - D_{e_{wu}}, rat_2) \\ RE_3 &= RE_1 \cup RE_2 \\ buf_3 &= 0 \end{aligned}$$

In addition, if $buf_2=1$, i.e., node u is a buffer node, an extra label $S_b = (C_4, rat_4, RE_4, buf_4)$ plus C'_{bl} will be generated for node u where C_{bl} is the auxiliary field of label Y ,

$$\begin{aligned} C'_{bl} &= C_{bl} + C_1 + C_{e_{wu}} \\ C_4 &= C_2 \\ q_w' &= rat_1 - D_{e_{wu}} - R_b C'_{bl} - D_b \\ q_u' &= rat_2 - R_b(C_1 + C_{e_{wu}}) \\ rat_4 &= \text{MIN}(q_w', q_u') \\ RE_4 &= RE_1 \cup RE_2 \\ buf_4 &= 1 \end{aligned}$$

The reason for keeping two labels is to leave open the possibility for finding the best solution, as using all available buffers may not offer the best performance. If node u is the source and $RE_1 \cup RE_2 = \text{sink_set}$, then the rat_3 of label S has to be updated, i.e., $rat_3 = rat_3 - R_d C_3$.

After a new label is generated, it is added to node u 's label set and the working queue Q . Then pruning is performed for u 's label set and the working queue Q . (This is described in Section 3.2.) This expansion proceeds until the algorithm fetches a label from Q that corresponds to a complete routing tree. That is, the label belongs to the source, and RE of that label contains all the sinks. This routing tree has the best solution because it has the maximal rat among all the routing trees generated and stored in Q .

During the process, for each node's label set, there may be more than one labels with the same RE and/or several labels with different RE s. These labels correspond to several subtrees which are rooted at the same node and span the same/different set of sinks. Unlike the A-tree and BA-tree algorithms which only keep one subtree at a merging point for pursuing shortest wirelength, our algorithm allows multiple subtrees to be kept. It provides more flexibility to find a performance-driven topology under various situations, such as asymmetric load capacitances and different required arrival times at sinks. In addition, buffer insertion can be considered when a label is expanded to a buffer node. In this way, performance-driven topology construction and buffer insertion under fixed buffer location constraints can be done simultaneously in a bottom-up fashion, leading to a buffered routing tree with the best performance.

3.2 Pruning

Pruning is based on a redundant relationship defined on two labels of the same node. Given two 4-tuple labels of node w : $label_1 = (C_1, rat_1, RE_1, buf_1)$, $label_2 = (C_2, rat_2, RE_2, buf_2)$, if $RE_1 = RE_2, C_1 < C_2, rat_1 > rat_2$, then $label_2$ is said to

be redundant with respect to $label_1$. Obviously, $label_2$ can be substituted by $label_1$ without increasing the delay.

After a new label is put back to a node's label set and the working queue, pruning is done based on the redundant definition, i.e., for each label set of a node and the working queue, only irredundant labels are kept. In this way we can avoid expanding inferior labels and enhance efficiency.

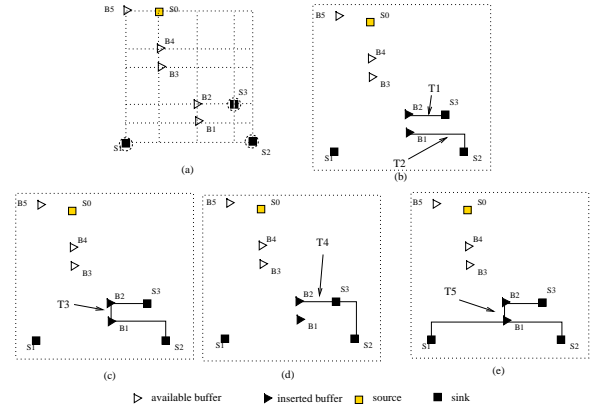


Figure 3: Subtree growth in the tree construction for a 4-pin net

Figure 3 shows how the subtrees grow during the tree construction for a 4-pin net with five available buffers. At the beginning, each sink is a subtree (Figure 3a). With the process of label expansion and generation going on, more subtrees are produced at each node. For example, there are two subtrees, T_1 and T_2 shown in Figure 3b. After they are merged at node B_2 , a new subtree T_3 (Figure 3c) rooted at node B_2 is generated. Moreover, multiple subtrees may be generated at a node and they may span the same/different set of sinks and may not be disjoint, like subtree T_3, T_4 and T_5 (Figure 3c, d, e) rooted at node B_2 . T_3 and T_4 span the same set of sinks, while T_3 and T_5 do not, and these subtrees are not disjoint. Unlike the A-tree algorithm, in order to leave open more possibility for finding the best performance-driven topology, we still keep the original separate subtrees T_1 and T_2 after merging them. Such subtree growth combined with pruning proceeds until a complete routing tree with the maximal rat is fetched from the working queue.

The pseudo-code of the RMP algorithm is shown in Figure 4.

3.3 Heuristic Rule for Expansion

As shown in the pseudo-code, when a node is expanded, it may go to any direction. In order to enhance efficiency, the edges toward the source are expanded first (similar to the A^* routing algorithm). The backward edges are stored as a backup in case the existence of obstacles prevents the shortest connection.

4 Experimental Results

We implemented the RMP algorithm in C++ language and tested it on a Sun Ultra 1 workstation with 256M memory running at 167 Mhz.

The key parameters for experimenting with the RFB problem are based on NTRS'97 0.18 μ m technology and are listed in Table 1. All the given buffers are of the same type with C_b , R_d and D_b .

Algorithm RMP

```

Begin-RMP
initiate label set for each node in
the routing graph  $G = (V, E)$ .
initiate working queue  $Q$ .
while  $Q$  is not empty do
{ fetch  $X = (C_1, rat_1, RE_1, buf_1)$  of node  $w$ 
with the maximal  $rat$  among all labels from  $Q$ ;
if  $w$  is source node and  $RE = sink\_set$  then
return the solution;
else for each edge  $(w, u) \in E$  do
{ for each label  $Y = (C_2, rat_2, RE_2, buf_2)$ 
in the label set of node  $u$  do
{ if  $RE_1 \cap RE_2 = \phi$  then
{  $RE_3 = RE_1 \cup RE_2$ ;
if  $buf_2 = 1$  then
{  $C'_{bl} = C_{bl} + C_1 + C_{e_{wu}}$ ;
 $C_{bl}$  is the auxiliary field of label  $Y$ ;
 $q_u = rat_2 - R_b(C_1 + C_{e_{wu}})$ ;
 $q_w = rat_1 - D_{e_{wu}} - R_b C'_{bl} - D_b$ ;
 $S_b \leftarrow (C_2, MIN(q_w, q_u), RE_3, 1)$ ;
add  $S_b$  to the label set of node  $u$  and  $Q$ 
and prune the redundant labels. }
 $C_3 = C_1 + C_{e_{wu}} + C_2$ ;
 $q_w = rat_1 - D_{e_{wu}}$ ;
 $S \leftarrow (C_3, MIN(rat_2, q_w), RE_3, 0)$ ;
if  $u$  is source and  $RE_3 = sink\_set$  then
{  $D_{s0} = R_d C_3$ ;
 $S \leftarrow (C_3, MIN(rat_2, q_w) - D_{s0}, RE_3, 0)$ ; }
add  $S$  to the label set of node  $u$  and  $Q$ 
and prune the redundant labels. } } } }
End-RMP

```

Figure 4: Pseudo-code of the RMP algorithm

r_0	unit length wire resistance ($\Omega/\mu m$)	0.076
c_0	unit length wire capacitance ($fF/\mu m$)	0.118
C_l	load capacitance of sink (fF)	23.4
R_d	driver resistance of source (Ω)	180
D_b	intrinsic delay for buffer (ps)	36.4
C_b	input capacitance of buffer (fF)	23.4
R_b	output resistance of buffer (Ω)	180

Table 1: Key parameters

4.1 Capabilities of the RMP Algorithm

The RMP algorithm has two major capabilities: routing tree construction with buffer insertion under fixed buffer location constraints and routing tree construction without buffer insertion. In addition, because the tree construction by RMP is based on the routing graph, RMP can handle obstacles and congestion, which will benefit its adaption in a global router. Figure 5 shows the routing tree of a 5-pin net without any buffer insertion and Figure 6 shows the routing tree for the same net with buffer insertion under fixed buffer location constraints. Both are generated by the RMP algorithm. All the required arrival times of sinks are set to 0. In these figures a square stands for a pin and a triangle stands for a buffer. The dark squares refer to sinks and the single light one refers to the source of the net. The number attached with a sink is the source-sink delay in ps . The triangles in a dark color refer to the buffers RMP uses for insertion, while those in a light color refer to the buffers provided, but not used by RMP (it means that inserting those buffers will not improve performance). In deep submicron technologies, with smaller driver resistance and relatively larger wire unit resistance, performance-driven routing topologies tend to be star-like as shown in Figure 6. It can be seen that RMP algorithm can choose the best buffer insertion at a given location for overall performance purposes. One may

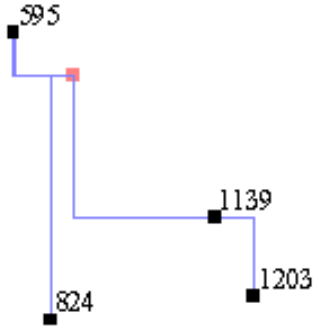


Figure 5: A routing tree without buffer insertion generated by the RMP algorithm

note that in order to use some buffers, RMP algorithm has to make some detours.

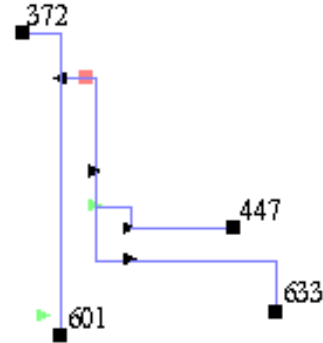


Figure 6: A routing tree with buffer insertion under fixed buffer location constraints generated by the RMP algorithm

4.2 Comparison between the RMP and A-tree Algorithms

In order to better understand the quality of the RMP algorithm, we tested it for routing tree construction only (without buffer insertion) and compared the result with that of the well-known A-tree algorithm in terms of required arrival time at the source. Because the A-tree algorithm uses a geometric abstraction and a linear delay model and does not address some issues in performance-driven routing such as asymmetric load capacitances and different required arrival times at sinks, it may not produce topologies with the best performance in the presence of asymmetric load capacitances and different required arrival times at sinks. We tested both algorithms under three cases:

- Case 1: uniform load capacitances and uniform required arrival times at sinks. All load capacitances of the sinks are set to $C_{l_{uniform}}$. The required arrival times at all the sinks are set to 0.
- Case 2: uniform load capacitances and non-uniform required arrival times at sinks. All load capacitances

of the sinks are set to $C_{l_{uniform}}$. The required arrival time at one sink is set to 0 and the other sinks' required arrival times are set to a large positive number; i.e., we define a critical source-sink pair.

- Case 3: non-uniform load capacitances and non-uniform required arrival times at sinks. The load capacitances of the sinks are not equal and are between a capacitance range $C_{l_{arrange}}$. The required arrival time at one sink is set to 0 and the other sinks' required arrival times are set to a large positive number; i.e., we define a critical source-sink pair.

Under each case, we randomly generate 100 nets for different kinds of nets which fall into a 6mm by 6mm bounding box (for nets spanning a larger distance, buffer insertion is usually required [5]). Average required arrival time of the source and wirelength are shown in Table 2. rat denotes the average required arrival time of the source, while wl denotes the average wirelength. All data are normalized with rat and wl in A-tree set to 1.

Tech: $r_0=0.076\Omega/\mu m$, $c_0=0.118fF/\mu m$, $R_d=270\Omega$ $C_{l_{uniform}}=23.4fF$, $C_{l_{arrange}}=1.17 \sim 70.2fF$						
#pin	case 1		case 2		case 3	
	rat	wl	rat	wl	rat	wl
5	0.99	1.02	0.91	1.06	0.89	1.09
6	0.99	1.03	0.88	1.11	0.86	1.08
7	0.98	1.06	0.85	1.11	0.84	1.10
8	0.97	1.06	0.83	1.07	0.87	1.08
9	0.97	1.05	0.82	1.08	0.82	1.09
10	0.97	1.06	0.83	1.09	0.82	1.09

Table 2: Experimental results of the RMP vs. A-tree algorithms

Experimental results show that in Case 1 the required arrival times at the sources of the topologies generated by the RMP and A-tree algorithms are very close for most of the nets, while the A-tree algorithm has smaller total wirelength. That is as expected since the A-tree is very good at total wirelength while maintaining a shortest path from the source to every sink. This shows that as a heuristic algorithm, the A-tree algorithm can most often find topology with the best performance in the case of uniform load capacitances and uniform required arrival times at sinks. However, for Case 2 and Case 3, RMP can outperform A-tree by up to 18% in terms of delay reduction with a 6-11% wirelength increase.

4.3 Comparison Between the RMP and Modified BA-tree Algorithms

In order to understand RMP algorithm's ability to handle fixed buffer locations, we implemented a modified BA-tree algorithm called MBA-tree algorithm to handle fixed location constraints and compared it to RMP. The MBA-tree algorithm imitates how human designers interact with a routing tree construction tool with unconstrained buffer insertion in a semi-automatic way. In the BA-tree algorithm, Steiner routing tree construction and buffer insertion are achieved simultaneously by combining A-tree construction and the dynamic programming-based buffer insertion algorithm (these two steps were carried out independently in the past). It is shown in [2] that the BA-tree algorithm can outperform a conventional two-step approach by up to 75% in terms of delay. In the MBA-tree algorithm, we first run the BA-tree algorithm to get a buffered BA-tree with buffers inserted at the ideal locations. Then we put it in the routing

graph of the RFB problem, check each ideal buffer location and try to "round it" to the nearest available buffer location. If there are no available buffers near the ideal buffer location within a certain range (we call it the neighborhood range, denoted as NR), this buffer insertion is abandoned, i.e., no buffer is inserted here. If the NR is very small, the MBA-tree tends to be the same as an A-tree with larger delay but smaller wirelength. But if the NR is too large, it is possible that using the available buffer in the neighborhood range will increase delay rather than decrease it. We define two cases for NR:

- Strict NR: NR only contains the node in the routing graph which is the closest to the ideal buffer and its adjacent nodes (as illustrated in Figure 7a).
- Relaxed NR: NR contains the first t nodes in the routing graph which are close to the ideal buffer (as illustrated in Figure 7a), where t is a parameter to be set by the algorithm or designers.

Through the experiment, we set t to 30 to get the best possible performance. These two cases bring about two different MAB-tree topologies as shown in Figure 7b and Figure 7c.

We extract 15 nets for each kind of net from the buffer planning results generated by BBP algorithm [1]. All the required arrival times of the sinks are set to 0. The average rat of the source and wirelength of the routing tree generated by the RMP algorithm and the MBA-tree algorithm under strict NR case and relaxed NR case are shown in Table 3. All data are normalized with rat and wl of RMP set to 1.

#pin	RMP		MBA-tree (strict NR)		MBA-tree (relaxed NR)	
	rat	wl	rat	wl	rat	wl
4	1.0	1.0	1.62	0.90	1.29	1.08
5	1.0	1.0	1.99	0.90	1.46	1.13
6	1.0	1.0	1.84	0.85	1.29	0.97

Table 3: Comparison between the RMP and MBA-tree algorithms

From the results, we can observe that RMP can outperform MBA-tree by up to 46% in terms of delay with comparative wirelength.

4.4 Runtime

The most time-consuming part of our algorithm is the expansion, which depends on the number of labels generated at each node. For a net with n sinks, theoretically there could be up to $2^n - 1$ different kinds of combination for the reachable set. But in practice, only a few nodes have so many reachable sets, while most nodes only have a small number of reachable sets during the expansion process. If we follow the pseudo-code strictly when implementing the algorithm, we should keep all the irredundant labels with the same reachable set at each node, which may result in long runtime. To speed up the algorithm, we only keep one label with the smallest cap for each reachable set. Table 4 shows the runtime enhancement and quality of such speedup version of the RMP algorithm, where rat_{SP} and rat_{NSP} refer to the rat of the speedup version and the non-speedup version respectively, T_{SP} and T_{NSP} refer to the runtime of the speedup version and the non-speedup version respectively. From Table 4 we can see that the runtime of the speedup version can decrease to 29% with only less than 3% quality loss.

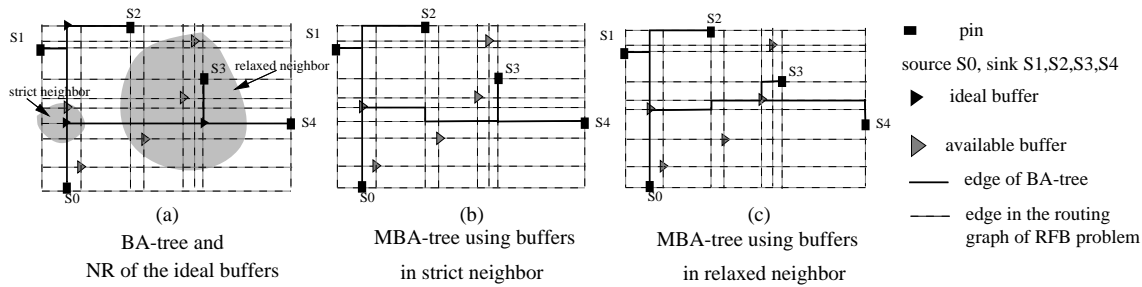


Figure 7: MBA-tree derived from BA-tree using buffers in strict/relaxed NR

without buffer insertion			with buffer insertion (#buffers ≤ 30)		
#pin	$\frac{ratsp}{rat_{NSP}}$	$\frac{Tsp}{T_{NSP}}$	#pin	$\frac{ratsp}{rat_{NSP}}$	$\frac{Tsp}{T_{NSP}}$
6	1.02	0.58	4	1.01	0.51
7	1.03	0.46	5	1.01	0.37
8	1.03	0.38	6	1.03	0.29

Table 4: Runtime and quality comparison between two versions of RMP (with speedup vs. without speedup)

Table 5 shows the average number of reachable sets per node generated for different number of sinks and buffers. In practice, n is usually 3 to 6, while the number of buffers can reach 30. The number of nodes in the routing graph is determined not only by the number of pins and buffers but also by their distribution. Besides the number of pins and buffers, the distribution may greatly affect the runtime. From Table 5, we see that the RMP algorithm scales reasonably well in practice.

#sink	without buffer insertion			with buffer insertion			
	#node per node	#RE	runtime (s)	#buffers	#node per node	#RE	runtime (s)
3	16	1	0.01	30	456	1	1
4	16	4	0.02	30	510	2	3
5	30	2	0.03	30	554	1	2
6	28	8	0.17	30	356	5	5

Table 5: The average number of REs per node in the routing graph and the runtime of the RMP algorithm

5 Conclusions and Future Work

In this paper, we presented an algorithm named RMP to solve the routing tree construction under fixed buffer locations problem. Our algorithm can deal with multiple-pin nets as well as fixed buffer location constraints. It can also handle obstacles and routing congestion. Experimental results show that the RMP algorithm is efficient and produces better results compared to using existing approaches to handle fixed buffer location constraints.

We plan to further enhance the algorithm for runtime improvement and extend the algorithm so that it can be used by a global router after buffer block planning. Also, we plan to use this algorithm to evaluate different buffer planning schemes.

Acknowledgments

The authors would like to thank Dr. M. K. Mohan from Intel and Prof. C.-K. Koh from Purdue University for their help-

ful discussions. This research is partially sponsored by Semiconductor Research Corporation under Contract 98-DJ-605 and a grant from Intel Corporation.

References

- [1] J. Cong, T. Kong, and D. Z. Pan, "Buffer block planning for interconnect-driven floorplanning," in *Proc. Int. Conf. on Computer-Aided Design*, pp. 358-363, 1999.
- [2] T. Okamoto and J. Cong, "Interconnect layout optimization by simultaneous Steiner tree construction and buffer insertion," in *Proc. ACM/SIGDA Physical Design Workshop*, pp. 1-6, 1996.
- [3] J. Cong, "Challenges and opportunities for design innovations in nanometer technologies," in *SRC Design Science Concept Papers*, http://www.src.org/prg_mgmt/frontier.dgw, 1997.
- [4] J. Cong, L. He, C.-K. Koh, and P. H. Madden, "Performance optimization of VLSI interconnect layout," *Integration, the VLSI Journal*, vol. 21, pp. 1-94, 1996.
- [5] J. Cong and D. Z. Pan, "Interconnect delay estimation models for synthesis and design planning," in *Proc. Asia and South Pacific Design Automation Conf.*, pp. 97-100, 1999.
- [6] L. P. P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *Proc. IEEE Int. Symp. on Circuits and Systems*, pp. 865-868, 1990.
- [7] J. Lillis, C. K. Cheng, and T. T. Y. Lin, "Simultaneous routing and buffer insertion for high performance interconnect," in *Proc. the Sixth Great Lakes Symp. on VLSI*, pp. 148-153, 1996.
- [8] J. Cong, K. S. Leung, and D. Zhou, "Performance-driven interconnect design based on distributed RC delay model," in *Proc. Design Automation Conf.*, pp. 606-611, 1993.
- [9] J. Cong, A. Kahng, and K. Leung, "Efficient algorithm for the minimum shortest path steiner arborescence problem with application to VLSI physical design," *IEEE Trans. on Computer-Aided Design*, vol. 17, pp. 24-38, 1998.
- [10] H. Zhou, D. F. Wong, I. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restriction on buffer location," in *Proc. Design Automation Conf.*, pp. 96-99, 1999.