A Flexible Data Structure for Efficient Buffer Insertion[*]

Ruiming Chen and Hai Zhou
Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208
{rch659, haizhou}@ece.northwestern.edu

ABSTRACT

With continuous down-scaling of minimum feature sizes and increasing of chip areas, buffering has become a necessary technique to control the interconnect delays in VLSI chips. Recently, Shi and Li proposed an efficient $O(n \log n)$ time algorithm to speed up buffering. Based on balanced binary search trees, their algorithm showed superb performance with the most unbalanced sizes of merging solution lists. We propose in this paper a more flexible data structure for the same buffering operations. With parameters to adjust, our algorithm works better than Shi and Li under all cases: unbalanced, balanced, and mix sizes. Our data structure is also simpler than theirs.

1. INTRODUCTION

With aggressive scaling down of feature sizes in VLSI fabrication, the interconnect delay becomes more and more dominant. Buffer insertion has been widely used to reduce interconnect delay [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Recently, projections of historical scaling trends by Saxena et al. [11] have predicted synthesis blocks with 70% of their cell count dedicated to interconnect buffers within a few process generations. Thus, a *fast* buffer insertion method is very necessary.

Van Ginneken [6] proposed a dynamic programming method to complete buffer insertion in distributed RC-tree networks for minimal Elmore delay, and his method runs in $O(n^2)$ time and space, where n is the number of legal buffer positions. Then many works based on his work emerged [1, 3, 4, 5, 2, 7, 8]. So the speed-up of van Ginneken's algorithm can benefit many other algorithms. Shi and Li [8] presented an $O(n \log n)$ algorithm for the optimal buffer insertion problem. Their work is based on the work proposed by Shi [12], which improves Stockmeyer's algorithm [13] for area minimization in slicing floorplan. The main idea of both works is that balanced binary search tree is used to represent solution candidates, and it avoids updating every candidate during the merging of two candidate lists. Since the most costly step in buffer insertion is the merging of two candidate lists, Shi's algorithm can get good speedup. However, as is shown in [12], the merging of two candidate lists based on balanced binary search tree can only speed-up the merging of two candidate lists of much different lengths (*unbalanced situation*) while for the merging of two candidate lists of similar lengths (*balanced situation*), the performance of their methods is worse than the method based on linked list.

[*]This work was supported by NSF under CCR-0238484.

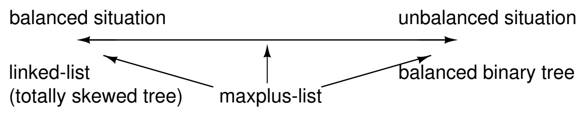

Figure 1: The flexibility of maxplus list

Figure 1 illustrates the best data structure for maintaining solutions in each of the two extreme cases: the balanced situation requires the linked list which can be viewed as a totally skewed tree; the unbalanced situation requires the balanced binary tree. However, most cases in reality are between these extremes, where neither data structure is the best. As we can see, the most balanced situation requires the most skewed data structure while the most unbalanced situation requires the most balanced data structure. Therefore, we need a data structure that is between a linked list and a balanced binary tree for the cases in the middle. We discovered that the skip list [14] is such a data structure as it migrates smoothly between a linked list and a balanced tree. In this paper, we propose a flexible data structure called *maxplus list* based on the skip list and related algorithms for operations in dynamic programming for buffer insertion. As is shown in Figure 1, we can migrate maxplus lists to suitable positions based on how balanced the routing tree is: a maxplus list will become a linked-list in balanced situations; it will behave like a balanced binary tree in unbalanced situations. So the performance of our algorithm is always very good. The experimental results show that it is even faster than the algorithm based on balanced binary search tree in unbalanced situations, and especially, it is much faster in balanced situations. Besides, maxplus list data structure is much easier to understand and implement than balanced binary search tree.

The rest of this paper is organized as follows. In Section 2, the general problem of merging two candidate lists is formulated, and the skip list data structure is introduced. In Section 3, maxplus list data structure and an efficient algorithm to merge two maxplus lists are shown. In Section 4, the method to find the optimal positions for buffer insertion is shown. The experimental results are reported in Section 5. The paper is concluded in Section 6.

2. PRELIMINARY

2.1 Maxplus problem

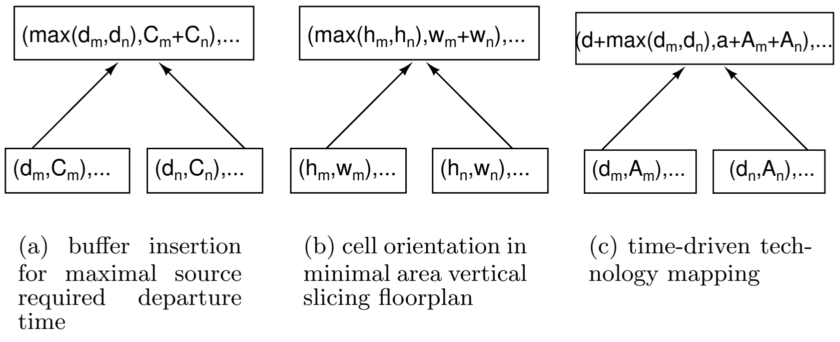
(max(d_m,d_n),C_m+C_n),...
(d_m,C_m),...
(d_n,C_n),...
(max(h_m,h_n),w_m+w_n),...
(h_m,w_m),...
(h_n,w_n),...
(d+max(d_m,d_n),a+A_m+A_n),...
(d_m,A_m),...
(d_n,A_n),...
(a) buffer insertion for maximal source required departure time
(b) cell orientation in minimal area vertical slicing floorplan
(c) time-driven technology mapping

Figure 2: The similar merge operations in three different problems

Given a routing tree as a distributed RC network, a dynamic programming approach to buffer insertion will build non-inferior solutions bottom-up from the sinks to the root. Each solution (d_v, C_v) at a node v represents a buffering of the subtree at v having d_v as the maximal delay to the sinks and C_v as the loading capacitance. When a tree at u is composed of a wire (u, v) and a subtree at v, its solution (d_u, C_u) can be computed as follows.

$$\begin{aligned} d_u &= d_v + r(u,v)(C_v + c(u,v)/2) \\ C_u &= C_v + c(u,v) \end{aligned}$$

where $r(u, v)$ and $c(u, v)$ are the resistance and capacitance of (u, v), respectively. When a buffer is inserted at the node v, the new solution can be computed similarly.

$$\begin{aligned} d'_v &= d_v + d_b + r_b C_v \\ C'_v &= c_b \end{aligned}$$

where d_b, r_b and c_b are the delay, resistance and input capacitance of the buffer, respectively. The most interesting case happens when two branches are combined at a node. As is shown in Figure 2(a), assuming that (d_m, C_m) is a solution in one branch and (d_n, C_n) in the other, the combined solution is given as follows.

$$\begin{aligned} d &= \max(d_m, d_n) \\ C &= C_m + C_n \end{aligned}$$

The optimal structure of dynamic programming requires that there is no solutions (d_1, C_1) and (d_2, C_2) such that $d_1 \le d_2$ and $C_1 \le C_2$ for the same subtree.

Given a slicing tree representing a floorplan, the problem of area minimization is to select the size of each module such that the chip area is minimized [13]. The dynamic programming approach [13] builds up the solutions bottom up. As is shown in Figure 2(b), given the solutions (h_m, w_m), (h_n, w_n) of the two subtrees and a parent node with vertical cut, a possible solution at the parent node can be constructed as $(\max(h_m, h_n), w_m + w_n)$.

Given a generic gate-level netlist, the technology mapping needs to map the circuit into a netlist composed of cells from a library. A popular heuristic [15] is to decompose the circuit into trees and apply a dynamic programming on each tree. When the objective is to minimize the area under a delay constraint [16], the generation of solutions at a node from those of its subtrees is shown in Figure 2(c), where (d_m, A_m) and (d_n, A_n) are the solutions at the fan-ins of a possible mapping, and d and a are the delay and area of the mapped cell. It can be decomposed into two steps: first compute $(\max(d_m, d_n), A_m + A_n)$ and then add d and a to its elements.

A common operation in the above dynamic programming approaches can be defined as follows.

Problem 1 (Maxplus problem). *Given two ordered lists* $A = \{(A_1.m, A_1.p), \ldots, (A_a.m, A_a.p)\}$ *and* $B = \{(B_1.m, B_1.p), \ldots, (B_b.m, B_b.p)\}$*, that is,* $A_i.m > A_j.m \wedge A_i.p < A_j.p$ *and* $B_i.m > B_j.m \wedge B_i.p < B_j.p$ *for any* $i < j$*, compute another ordered list* $C = \{(C_1.m, C_1.p), \ldots, (C_c.m, C_c.p)\}$ *such that it is a* maxplus merge *of A and B, that is, for any* $0 < k \le c$ *there are* $0 < i \le a$ *and* $0 < j \le b$ *such that*

$$C_k.m = \max(A_i.m, B_j.m) \wedge C_k.p = A_i.p + B_j.p$$

and for any $0 < i \le a$ *and* $0 < j \le b$ *there is* $0 < k \le c$ *such that*

$$\max(A_i.m, B_j.m) \ge C_k.m \wedge A_i.p + B_j.p \ge C_k.p.$$

2.2 Stockmeyer's algorithm

A straight-forward approach to the maxplus problem is to first compute $(\max(A_i.m, B_j.m), A_i.p + B_j.p)$ for every $0 < i \le a$ and $0 < j \le b$ and then delete all inferior solutions. However, it takes at least $\Omega(ab)$ time. Stockmeyer [13] proposed a $O(a + b)$ time algorithm where inferior solutions can be directly avoided. The idea is as follows. First, since $A_1.p + B_1.p$ is the smallest, the solution $(\max(A_1.m, B_1.m), A_1.p + B_1.p)$ must be assigned to C_1. Then if $A_1.m = C_1.m$, any solution $(\max(A_1.m, b_i.m), A_1.p + B_i.p) = (C_1.m, A_1.p + B_i.p)$ for any $1 < i \le b$ is inferior to C_1, thus should not be generated. Since all combinations with A_1 have been considered (even though not generated), we can proceed with A_2. This process can be iterated and the pseudo-code is given in Figure 3.

2.3 Skip list

The advantage of a balanced binary search tree over a linked list is its capability to quickly find an item ranked around the middle. Skip list [14] is an alternative data structure to a balanced binary search tree. It can be viewed as a combination of multiple linked lists, each on a different level. The list on the lowest level includes all the items while that on a higher level has fewer items. An example skip list is illustrated in Figure 4. An item on k linked lists, that is, with k forward pointers, is called a *level k* item. As we can

```
Algorithm STOCKMEYER$(A, B, C)$
    $C \leftarrow \Phi$
    while $A \neq \Phi \wedge B \neq \Phi$
        do if $A_1.m \geq B_1.m$
            then $P \leftarrow A_1$
                 $Q \leftarrow B_1$
            else $P \leftarrow B_1$
                 $Q \leftarrow A_1$
           $P.p \leftarrow P.p + Q.p$
           Append $P$ to $C$
           if $P.m = Q.m$
              then delete $Q$
    return $C$
```

Figure 3: Stockmeyer's Algorithm

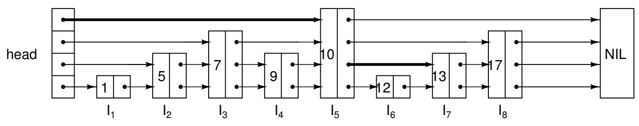

Figure 4: Skip list

see, if the level-k items are evenly distributed among the level-$(k-1)$ items, a skip list can achieve the function of a balanced binary search tree, that is, finding any item in $O(\log n)$ time.

It is impractical to modify item levels during operations to maintain the balance among levels. An effective way is to randomly choose an item level during insertion and keep it fixed thereafter. Therefore, a skip list has two parameters: the maximal permitted level *MaxLevel* and the probability p that a level-k item is also a level-$(k+1)$ item. Different values of *MaxLevel* and p may lead to different costs for operations. In [14], it is suggested that $p = 0.25$ and $MaxLevel = \log_{\frac{1}{p}}(N)$, where N is the upper bound on the number of items in the skip list. Each skip list has a *head* that has forward pointers at levels one through *MaxLevel*. The expected running time of search, insertion and deletion of one item in a skip list with n items is $O(\log n)$.

3. MAXPLUS LIST

Even though Stockmeyer's algorithm takes linear time to combine two lists, when the merge tree is skewed, it may take n^2 time to combine all the lists even though the total number of items is n. For example, in such a case, a list A of size n and a list B of size one may be combined in one step, which may take $O(n)$ time in Stockmeyer's algorithm. However, if we can quickly find $0 < i \leq n$ such that $A_i.m \geq B_1.m$ and $A_{i+1}.m < B_1.m$, the new list will have the first i items in A with their p properties incremented by $B_1.p$. This is the idea explored by Shi [12]. He proposed to use a balanced binary search tree to represent each list so that the searching can be done in $O(\log n)$ time. To avoid modifying the p properties individually, the modification was annotated on an item for the subtree rooted at it. Shi's algorithm is faster when the merge tree is skewed since it takes $O(n \log n)$ time comparing with Stockmeyer's $O(n^2)$ time. However, Shi's algorithm is complicated and also much slower than Stockmeyer's when the merge tree is balanced.

In stead of a balanced binary tree, we proposed the maxplus list based on the skip list to hold the solutions for buffer insertion etc. Since a maxplus list is close to a linked list, its merge operation is just a simple extension of Stockmeyer's algorithm. As is shown in Figure 3, during each iteration of Stockmeyer's algorithm, the current item with the maximal m property in one list is finished and the new item is equal to the finished item with its p property incremented by the p property of the other current item. The idea of the maxplus list is to finish a sub-list of more than one items at one iteration. Assume that $A_i.m > B_j.m$, we want to find a $i \leq k \leq a$ such that $A_k.m \leq B_j.m$ but $A_{k+1}.m < B_j.m$. These items $A_i, \ldots, A_k$ are finished and can be put into the new list after their p properties is incremented by $B_j.p$. Of course, the speed-up over Stockmeyer's algorithm comes from the fact that this sub-list is identified and updated not item-by-item. The pointers in the skip list will be used to "skip" items when searching for the sub-list, and an adjust field will be associated with each forward pointer to record the incremental amount on the skipped items.

3.1 Data structure

Maxplus list is a skip list with each item defined by the following C code.

```
struct maxplus_item{
    int level; /*the level*/
    float m, p; /*two properties*/
    float *adjust;
    struct maxplus_item **forward; /*forward pointers*/
}
```

The size of *adjust* array is equal to the level of this item, and *adjust*$[i]$ means that p properties of all the items jumped over by *forward*$[i]$ should add *adjust*$[i]$.

3.2 Merge operation

```
Algorithm ML-MERGE$(A, B, C)$
    $C \leftarrow \Phi$
    while $A \neq \Phi \wedge B \neq \Phi$
        do if $A_1.m \geq B_1.m$
            then $list \leftarrow A$
                 $item \leftarrow B_1$
            else $list \leftarrow B$
                 $item \leftarrow A_1$
           $cut \leftarrow$ ML-SEARCHSUBLIST$(list, item)$
           ML-CLEARADJUST$(cut)$
           Append the sub-list before $cut$ to $C$
           if $cut[1].m = item.m$
              then Clear the $adjust$ array of $item$
                   Delete $item$ from the maxplus list
    return $C$
```

Figure 5: Merge of two maxplus lists

We define a *cut* after item I of a maxplus list L, denoted by cut_I, as an array of size *MaxLevel* with the ith item being the last item with its level larger or equal to i before item I (including I). For example, in Figure 4, the *cut* after I_7 is:

$cut[1] = I_7$, $cut[2] = I_7$, $cut[3] = I_5$, $cut[4] = I_5$. We can see that the levels of items in a cut form a stair.

ML-Merge, the algorithm to solve Maxplus problem, is shown in Figure 5. It is very similar to Stockmeyer's algorithm. As we have said before, the basic idea is to find the sub-list with $A_i.m \geq B_j.m$ very efficiently, and the sub-list with $B_j.m \geq A_i.m$ very efficiently, then we can move ahead much more on both lists in Stockmeyer's algorithm.

Suppose in ML-Merge initially maxplus list A and B are both sorted in the decreasing order of m and increasing order of p, and have no redundant items. The pseudocode of procedure ML-SearchSublist(*list*, *item*) is shown in Figure 6. It starts from the head of *list* to search for the longest sub-list R satisfying

$$\forall I \in R \; I.m \geq item.m\,.$$

During the search, the p property of each visited item is increased by *item.p*, and *adjust*$[i]$ of each visited item is increased by *item.p* if the corresponding forward pointer is used to jump over. It returns a cut after the last item in R. The subprocedure ML-Propagate increases the p properties and *adjust*[*level*] of items on the rising stairs before the item with the highest level of the cut by *item.p*.

```
ML-SearchSublist(list, item)
    cut ← list.head
    item1 ← NIL
    for i ← MaxLevel downto 1
            do while $cut[i].forward[i].m \geq item.m$
                    do if item1 ≠ NIL
                            then $cut[i].adjust[i] \leftarrow$
                                    $cut[i].adjust[i] + item.p$
                            else ML-Propagate
                                    (list.head.forward[1],
                                    $cut[i].forward[i]$, item.p)
                        $cut[i].forward[i].p \leftarrow$
                          $cut[i].forward[i].p + item.p$
                        item1 ← $cut[i]$ ← $cut[i].forward[i]$
                $cut[i]$ ← item1;
    return cut;
```

Figure 6: Procedure ML-SearchSublist

In order to simplify the move procedure, the following condition is required to be satisfied before appending the sub-list.

> Condition*1*: *adjust*$[i]$ of *cut*$[i]$ is always 0 for $i = 1, \ldots MaxLevel$.

Procedure ML-ClearAdjust(*cut*) completes this task. As is shown in Figure 7, we want to reset the *adjust* array of the cut (N_6, N_5, N_4, N_1). We walk down along the left stair from N_1. Firstly, in order to satisfy Condition1, the value of $N_1.\,adjust[7]$ is required to be 0, so we add its value to $N_1.\,adjust[6]$ and $N_4.\,adjust[6]$ and $N_4.p$. Then we can set the value of $N_1.\,adjust[7]$ to be 0. Now we have gone to the level 6 stair, and we need to set the value of $N_4.\,adjust[6]$ to be zero, so similarly, we add its value to $N_4.\,adjust[5]$. Keep walking till we arrive at the downstairs, then the p property of any item along the right stair is the actual value. Now Condition1 is satisfied. The update of *adjust* arrays in ML-Propagate, and the clearance of *adjust* arrays in ML-Merge can be similarly accomplished by walking along stairs.

Another important operation in Maxplus problem is the evaluation of p properties of items. It has two step, firstly we search for item I and simultaneously find the cut after I, then

$$I.p = I.p + \sum_{I.\,level < i \leq MaxLevel} cut[i].adjust[i].$$

3.3 Determination of *MaxLevel*

Our experiments show that the different values of *MaxLevel* can lead to much different running time. For example, when *MaxLevel* is equal to 1, the algorithm runs fastest in balanced situations, while it becomes much worse in unbalanced situations. As is shown in Figure 1, the method based on linked-list is much faster in balanced situations, while the method based on binary search tree is faster in unbalanced situations. An important property of maxplus list is that it is a very flexible data structure, that is, when $MaxLevel = 1$, it becomes a linked-list, while when *MaxLevel* increases, it becomes more like a binary search tree. In order to get the best speedup, we cannot keep the same value of *MaxLevel* in all situations. Instead, here we presented a simple strategy to determine the value of *MaxLevel*. According to results of the statistical experiments in [14], here we fix the value of p at 0.25.

In the problems of buffer insertion, floorplan or technology mapping related with Maxplus problem, the input is always a tree. We define *basic element* as the buffer positions in buffer insertion, realizations of basic blocks in floorplan and mappings in technology mapping. During the read of input files, we can record the maximal and minimal depth of leaves in the tree: D_{max} and D_{min}. Then if $(D_{min} \geq D_{max}/2)$, we set $MaxLevel = 1$, otherwise we set $MaxLevel = \lfloor \log_{\frac{1}{p}}(n/8) \rfloor$, where n is the number of basic elements.

4. BUFFER INSERTION

For buffer insertion in a distributed RC-tree network, besides the merging of candidate list, it is required to attach wires and buffers, and the methods to do these tasks introduced in [8] can be easily implemented using our data structure. Another important task is to find the optimal buffer positions after the optimal T_{source} is calculated.

In order to record the composition of each item, we modify our data structure to include a pointer array *comp* of size *level* in each item. During the bottom-up calculation of non-redundant candidates, we maintain a *configuration graph* to record the composition of each item. The pointers in *comp* array point to the vertices representing compositions in configuration graph. Similar to the *adjust* array, *comp*$[i]$ means that all the items between this item and the next item with level larger or equal to i are composed partly by the composition that it points.

Now we can update the *comp* array similar to the *adjust* array in the merge operations, and at the same time construct the configuration graph. For example, if the items from item m to item n in a maxplus list are merged with an item k, then when we update the value of *adjust*$[i]$ of an item in ML-Merge, we simultaneously create a new vertex v in the configuration graph, and add edges from v to

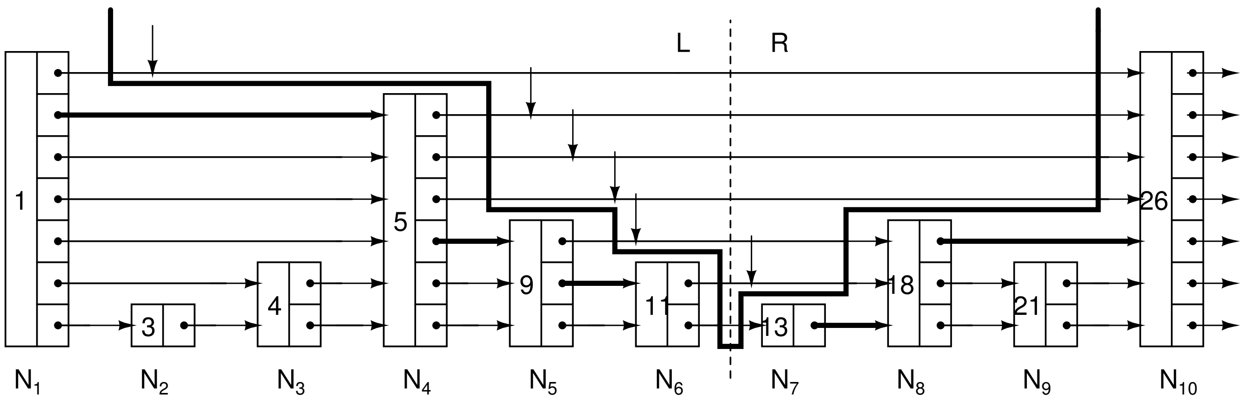

Figure 7: An example to show the flow of ML-Lowering

comp[*i*] and all the vertices pointed by *comp* array of item k, then let *comp*[*i*] point to v. Similarly, we also modified ML-Clearadjust to keep an additional condition before appending the sub-list in each iteration of ML-Merge:

> Condition*2*: *comp*[*i*] of *cut*[*i*] is always NIL for $i = 2, ...MaxLevel$.

When we want to attach a buffer at the current location, we create a new vertex v in the configuration graph, and add edges from v to the vertex representing the buffer location and the vertex pointed by *comp*[1], then make *comp*[1] point to v.

We can see that the configuration graph is a directed acyclic graph that is very similar to the well-known Binary Decision Diagram(BDD) [17]. Traversing from any vertex v in the configuration graph, the buffers visited represent part of a buffering solution.

After the bottom-up calculation of non-redundant candidates, we firstly evaluate the m and p properties of each item and select an optimal item. At the same time, we merge all the pointers in *comp* array for each item. So after evaluation, each item has a single pointer pointing to its composition. We traversal the configuration graph starting from the vertex that the composition pointer of the optimal item points to, then we get all the inserted buffers in this optimal solution.

5. EXPERIMENTAL RESULTS

Table 1: Comparison results for unbalanced trees

Name	Leaves	Alg 1	Alg 2	Our Alg.		
	#	time	time	time	T_1/T_3	T_2/T_3
		(T_1 s)	(T_2 s)	(T_3 s)		
U100	100	0.083	0.033	0.033	2.52	1.00
U200	200	0.233	0.083	0.067	3.48	1.24
U300	300	0.567	0.150	0.100	5.67	1.50
U400	400	1.033	0.200	0.133	7.77	1.50
U500	500	1.530	0.250	0.180	8.50	1.39
U600	600	2.080	0.333	0.217	9.59	1.53
U700	700	2.900	0.485	0.267	10.86	1.82
U800	800	3.533	0.433	0.317	11.15	1.37
U900	900	4.500	0.500	0.333	13.51	1.50
U1000	1,000	5.633	0.767	0.383	14.71	2.00

Since merging of solution lists is the most costly step in buffer insertion, we are focusing on testing the performance of our maxplus list based merging method. We designed many test cases with each case corresponding to a tree, and every leaf in a tree has four basic options. We implement a bottom-up algorithm based on our merge algorithm in C to calculate the non-redundant candidate list at the root of each tree. We implement Stockmeyer's algorithm, and download the code of Shi's merge algorithm from Shi's webpage [18] for comparison. The running time is the total time for executing each algorithm 100 times, and doesnot include the time to read input files and to print final results. All the experiments were run on a Linux PC with 2.4G Hz Xeon CPU and 2.0GB memory.

For unbalanced trees, the comparison of our algorithm, Stockmeyer's algorithm and Shi's algorithm is shown in Table 1. Column 3 and Column 4 are the running time of Stockmeyer's algorithm and Shi's algorithm respectively. We use $MaxLevel = 4$, and $p = 0.25$ in the maxplus list. The results indicate that our algorithm is much faster than Stockmeyer's algorithm, and with the increasing size of cases, it can get more and more speed-up. Most importantly, our algorithm is about 1.5 times faster than Shi's algorithm on average.

For balanced trees, the comparison of our algorithm, Stockmeyer's algorithm and Shi's algorithm is shown in Table 2. The 5th column is the running time of our method with $MaxLevel = 1$, $p = 0$, and the 6th column is the running time of our method with $MaxLevel = 4$, $p = 0.25$. The results shows that our algorithm with $MaxLevel = 1$ is even faster than Stockmeyer's algorithm in some cases. It is because when $MaxLevel = 1$, the skip list becomes an ordinary linked-list, but our method moves a series of items in each iteration while Stockmeyer's algorithm moves one by one. When $MaxLevel = 4$, our algorithm is slower than Stockmeyer's algorithm but more than 2.5 times faster than Shi's algorithm.

For mixed trees that contain both balanced subtrees and unbalanced subtrees, we use our strategy mentioned before to determine the value of $MaxLevel$. The comparison of our algorithm, Stockmeyer's algorithm and Shi's algorithm is shown in Table 3. We can see that our algorithm is always more than 2 times faster than Shi's algorithm. Especially for Mdata6 case, our strategy to determine $MaxLevel$ improved the speed much.

Table 2: Comparison results for balanced trees

Name	Leaves#	Stockmeyer's Alg.	Shi's Alg.	Our Alg.			
		time(T_1 sec.)	time(T_2 sec.)	(1,0) time(sec)	(4,0.25) time(T_3 sec.)	T_1/T_3	T_2/T_3
B128	128	0.033	0.083	0.033	0.033	1.00	2.52
B256	256	0.100	0.317	0.133	0.100	1.00	3.17
B512	512	0.167	0.883	0.183	0.217	0.77	4.07
B1024	1,024	0.350	1.433	0.267	0.466	0.75	3.08
B2048	2,048	0.717	2.950	0.583	0.983	0.73	3.01
BLARGE	32,768	12.730	50.390	10.700	18.550	0.69	2.72

Table 3: Comparison results for mixed trees

Name	Leaves#	Stockmeyer's Alg.	Shi's Alg.	Our Alg.			
		time(T_1 sec.)	time(T_2 sec.)	*MaxLevel*	time(T_3 sec.)	T_1/T_3	T_2/T_3
Mdata1	82	0.017	0.067	2	0.033	0.52	2.03
Mdata2	296	0.483	0.300	3	0.117	4.13	2.56
Mdata3	1236	0.633	1.750	4	0.616	1.03	2.84
Mdata4	2196	11.633	2.767	5	1.333	8.73	2.08
Mdata5	8046	3.317	11.383	5	4.550	0.73	2.50
Mdata6	21892	9.200	37.233	1	9.533	0.97	3.91

6. CONCLUSION

We presented a flexible data structure, maxplus list, to represent solution list, and designed an efficient buffer insertion algorithm based on maxplus list. With parameters to adjust automatically, our algorithm works better than Shi and Li [8] under all cases: unbalanced, balanced, and mix sizes. Our data structure is also simpler than theirs.

7. REFERENCES

[1] C. Alpert and A. Devgan. Wire segmenting for improved buffer insertion. In *Proc. of the Design Automation Conf.*, pages 588–593, 1997.

[2] M. Kang, W. W.-M. Dai, T. Dillinger, and D. LaPotin. Delay bounded buffered tree construction for timing driven floorplanning. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 707–712, 1997.

[3] 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*, 31:437–447, 1996.

[4] J. Lillis et al. New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing. In *Proc. of the Design Automation Conf.*, pages 395–400, 1996.

[5] T. Okamoto and J. Cong. Buffered Steiner tree construction with wire sizing for interconnect layout optimization. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 44–49, 1996.

[6] L. P. P. P. van Ginneken. Buffer placement in distributed RC-tree networks for minimal Elmore delay. In *Proc. Intl. Symposium on Circuits and Systems*, pages 865–868, 1990.

[7] Hai Zhou, D. F. Wong, I-Min Liu, and Adnan Aziz. Simultaneous routing and buffer insertion with restrictions on buffer locations. Proc. of the Design Automation Conf., 1999.

[8] Weiping Shi and Zhuo Li. An O(nlogn) time algorithm for optimal buffer insertion. In *Proc. of the Design Automation Conf.*, pages 580–585, 2003.

[9] V. Khandelwal, A. Davoodi, A. Nanavati, and A. Srivastava. A probabilistic approach to buffer insertion. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 560–567, 2003.

[10] C. J. Alpert, M. Hrkic, and S. T. Quay. A fast algorithm for identifying good buffer insertion candidate locations. In *International Symposium on Physical Design*, pages 47–52, 2004.

[11] P. Saxena, N. Menezes, P. Cocchini, and Desmond A. Kirkpatrick. The scaling challenge: Can correct-by-construction design help? In *International Symposium on Physical Design*, pages 51–58, 2003.

[12] Weiping Shi. A fast algorithm for area minimization of slicing floorplans. *tcad*, 15:550–557, 1996.

[13] L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Information and Control*, 59:91–101, 1983.

[14] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), 1990.

[15] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *Proc. of the Design Automation Conf.*, pages 617–623, June 1987.

[16] K. Chaudhary and M. Pedram. A near optimal altorithm for technology mapping minimizing area under delay constraints. In *Proc. of the Design Automation Conf.*, pages 492–498, July 1992.

[17] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[18] Weiping Shi. `http://ece.tamu.edu/~wshi`.