[12] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive protocols," in *Proc. DAC*, pp. 361–367.

[13] Y. I. Ismail and E. G. Friedman, "Effects of inductance on the propagation delay and repeater insertion in VLSI circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 2, pp. 195–206, Apr. 2000.

[14] W. Dai, L. Wu, and S. Zhang, *GSRC T2 Bookshelf at UCSanta Cruz*. 2003. [Online]. Available: www.cse.ucsc.edu/research/surf/GSRC/progress.html

[15] M. Singh and M. Theobald, "Generalized latency insensitive systems for single-clock and multi-clock architectures," in *Proc. DATE*, Paris, France, 2004, p. 21 008.

[16] S. Adya, H. H. Chan, and I. Markov Parquet, *Fixed-Outline Floorplanner,* (2006). [Online]. Available: http://vlsicad.eecs.umich.edu/BK/parquet/

[17] S. N. Adya and I. L. Markov, "Fixed-outline floorplanning: Enabling hierarchical design," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 6, pp. 1120–1135, Dec. 2003.

[18] M. Ikeda *et al.*, "A hardware/software concurrent design for real-time SP@ML MPEG2 video-encoder chip set," in *Proc. Eur. Des. and Test Conf.*, Mar. 1996, pp. 320–326.

An Efficient Data Structure for Maxplus Merge in Dynamic Programming

Ruiming Chen and Hai Zhou

***Abstract*—Dynamic programming is a useful technique to handle slicing floorplan, technology mapping, and buffering problems, where many maxplus merge operations of solution lists are needed. Shi proposed an efficient $O(n \log n)$ time algorithm to speed up the merge operation. Based on balanced binary search trees, his algorithm showed superb performance with the most unbalanced sizes of merging solution lists. The authors propose in this paper a more efficient data structure for the merge operations. With parameters to adjust adaptively, their algorithm works better than Shi's under all cases, unbalanced, balanced, and mix sizes. Their data structure is also simpler.**

***Index Terms*—Data structure, dynamic programming, timing optimization.**

I. Introduction

Dynamic programming is an effective technique to handle slicing floorplan [1], technology mapping [2], and buffering [3] problems. For example, van Ginneken [3] 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. An essential operation in van Ginneken's algorithm is to merge two candidate lists into one list where inferior candidates are pruned. Shi [4] proposed an efficient algorithm that improves Stockmeyer's algorithm [1] for the merge operation in slicing floorplan. Based on it, Shi and Li [5] presented an $O(n \log^2 n)$ algorithm for the optimal buffer insertion problem. In these algorithms, a balanced binary search tree is used to represent a list of solution candidates, and it avoids updating every

Manuscript received April 8, 2005; revised August 19, 2005 and November 23, 2005. This paper was recommended by Associate Editor J. Lillis.

The authors are with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: haizhou@ece.northwestern.edu).

Digital Object Identifier 10.1109/TCAD.2006.882479

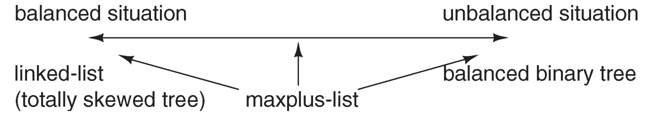

Fig. 1. Flexibility of maxplus-list.

candidate during the merge of two candidate lists. However, as shown in [4], the merge of two candidate lists based on balanced binary search trees can only speed up the merge of two candidate lists of much different lengths (unbalanced situation), but not the merge of two candidate lists of similar lengths (balanced situation).

Fig. 1 illustrates the best data structure for maintaining solutions in each of the two extreme cases: the balanced situation requires a linked list that can be viewed as a totally skewed tree or the unbalanced situation requires a 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 a skip-list [6] is such a data structure as it migrates smoothly between a linked list and a balanced tree. In this paper, we propose an efficient data structure called maxplus-list based on the skip-list and corresponding algorithms for merge operations in the dynamic programming. As shown in Fig. 1, we can migrate the maxplus-lists to suitable positions based on how balanced the routing tree is; a maxplus-list becomes a linked-list in balanced situations or it behaves like a balanced binary tree in unbalanced situations. Therefore, the performance of our algorithm is always very good. The maxplus-merge algorithm based on maxplus-list has the same asymptotic time complexity as the merge algorithm used in [4] and [5]. Our experimental results show that it is even faster than the balanced binary search tree in unbalanced situations, and it is much faster in balanced situations. Besides, the 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 II, the general problem of merging two candidate lists is formulated, and the skip-list data structure is reviewed. In Section III, the maxplus-list data structure and an efficient algorithm to merge two maxplus-lists are shown. In Section IV, the approach for finding the optimal solutions after the bottom-up merge operations is shown. The experimental results are reported in Section V. Finally, the conclusion and future work are given in Section VI.

II. Preliminary

A. Maxplus Problem

The following three different problems have the similar algorithmic structure, the merge of candidate solution lists.

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 [1]. The dynamic programming approach [1] builds the solutions bottom up. Each solution (h_v, w_v) at a node v represents a floorplan at v having h_v as the height and w_v as the width. As shown in Fig. 2(a), given the solutions (h_m, w_m), (h_n, w_n) of the two subtrees and a parent node with vertical cut, a candidate solution at the parent node can be constructed as $(\max(h_m, h_n), w_m + w_n)$. The optimal structure of dynamic programming requires that there are no solutions (h_1, w_1) and (h_2, w_2) such that $h_1 \le h_2$ and $w_1 \le w_2$ for the same subtree.

0278-0070/$20.00 © 2006 IEEE

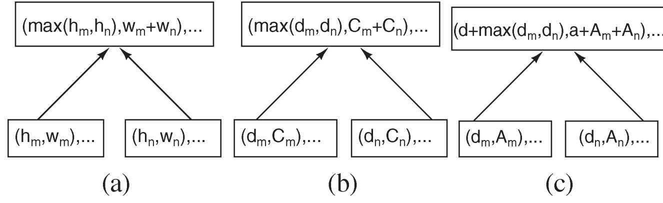

Fig. 2. Similar merge operations in three different problems: (a) cell orientation in minimal area slicing floorplan; (b) buffer insertion for maximal required departure time; and (c) time-driven technology mapping.

It is very interesting to note that such an operation also appears in many other optimization problems, such as the single net buffering problem. Given a routing tree as a distributed RC network, a dynamic programming approach to buffer insertion will build noninferior solutions bottom up from the sinks to the root. The objective is to insert buffers such that the maximal delay D_{source} from the root to sinks is minimized. Each solution (d_v, C_v) at a node v represents a buffering of the subtree rooted 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:

$$d_u = d_v + r(u,v)(C_v + c(u,v)/2)$$
$$C_u = C_v + c(u,v)$$

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

$$d'_v = d_v + d_b + r_b C_v$$
$$C'_v = c_b$$

where d_b, r_b, and c_b are the internal delay, output resistance, and input capacitance of the buffer, respectively. The most interesting case happens when two branches are combined at a node. As shown in Fig. 2(b), 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

$$d = \max(d_m, d_n)$$
$$C = C_m + C_n.$$

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

Also, the technology mapping problem involves the merge of two candidate solution lists. 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 [2] is to decompose the circuit into trees and apply a dynamic programming on each tree. Each solution (d_v, A_v) at a node v represents a technology mapping at v having d_v as the maximal delay to the sinks and A_v as the area. When the objective is to minimize the area under a delay constraint [7], the generation of solutions at a node from those of its subtrees is shown in Fig. 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 at node v, respectively. The mapping can be decomposed into two steps, first compute $(\max(d_m, d_n), A_m + A_n)$ and then add d and a to the solutions.

```
Algorithm STOCKMEYER(A, B, C)
  C ← ∅
  while A ≠ ∅ ∧ B ≠ ∅
  {
    if A_1.m ≥ B_1.m
    {
      P ← A_1
      Q ← B_1
    }
    else
    {
      P ← B_1
      Q ← A_1
    }
    P.p ← P.p + Q.p
    Append P to C
    Delete P from its original list
    if P.m = Q.m
      delete Q
  }
  return C
```

Fig. 3. Stockmeyer's algorithm.

One 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 $0 < k \leq c$ there are $0 < i \leq a$ and $0 < j \leq b$ such that

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

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

$$\max(A_i.m, B_j.m) \geq C_k.m$$

and

$$A_i.p + B_j.p \geq C_k.p.$$

B. Stockmeyer's Algorithm

A straightforward approach to the maxplus problem is to first compute $(\max(A_i.m, B_j.m), A_i.p + B_j.p)$ for every $0 < i \leq a$ and $0 < j \leq b$ and then delete all inferior solutions. However, it takes at least $\Omega(ab)$ time. Stockmeyer [1] 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 $1 < i \leq b$ 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 pseudocode is given in Fig. 3, where $\emptyset$ represents an empty list.

C. 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.

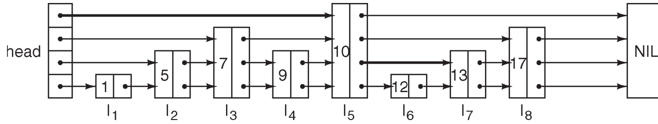

Fig. 4. Skip-list.

Skip-list [6] 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 Fig. 4. An item on k linked lists, that is, with k forward pointers, is called a level-k item. As we can 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 expensive 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_r that an item with level k forward pointers also has level-$(k+1)$ forward pointers. Different values of MaxLevel and P_r may lead to different costs for operations. In [6], it is suggested that $P_r = 0.25$ and MaxLevel $= \log_{1/P_r}(N)$, 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 $O(\log n)$ [6].

III. 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, a list A of size $n/2$ and a list B of size one may be combined in one stage, which may take $O(n/2)$ time in Stockmeyer's algorithm. For a skew tree with n items, the total number of merging stages could be $n/2$, giving a total running time of $O(n^2)$. 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 [4]. He proposed to use a balanced binary search tree to represent each list so that the search can be done in $O(\log n)$ time. To avoid updating the p properties individually, the update was annotated on a node for the rooted subtree. Shi's algorithm is faster when the merge tree is skewed since $O(n \log n)$ 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.

Instead of a balanced binary tree, we proposed a data structure called maxplus-list based on the skip-list for keeping candidate solutions. Since a maxplus list is similar to a linked list, its merge operation is just a simple extension of Stockmeyer's algorithm. As shown in Fig. 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 sublist of more than one item at one iteration. Assume that $A_i.m > B_j.m$, we want to find a $i \leq k \leq a$ such that $A_k.m \geq B_j.m$ but $A_{k+1}.m < B_j.m$. These items $A_i, \ldots, A_k$ are finished and put into the new list after their p properties are incremented by $B_j.p$. The speedup over Stockmeyer's algorithm comes from the fact that this sublist is processed (identified and updated) in a batch mode instead of item by item. The forward pointers in a maxplus-list are used to skip items when searching for the sublist, and an adjust field is associated with each forward pointer to record the incremental amount on the skipped items.

```
Algorithm ML-MERGE(A, B, C)
  C ← ∅
  CUT ← C.head
  while A ≠ ∅ ∧ B ≠ ∅
  {
    if A1.m ≥ B1.m
    {
        list ← A
        item ← B1
    }
    else
    {
        list ← B
        item ← A1
    }
    cut ← ML-SEARCHSUBLIST(list, item)
    sublist ← ML-EXTRACTSUBLIST(cut)
    ML-APPENDSUBLIST(sublist, CUT)
    CUT ← cut
    if cut[1].m = item.m
    {
        Clear the adjust array of item
        Delete item from the maxplus-list
    }
  }
  return C
```

Fig. 5. Merge of two maxplus-lists.

A. Data Structure

A 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 a value of adjust$[i]$.

B. Merge Operation

We define a cut after item I in a maxplus-list, 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 Fig. 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 items in a cut form stairs.

ML-Merge, the algorithm to solve Maxplus problem, is shown in Fig. 5. It is very similar to Stockmeyer's algorithm. As we have mentioned before, the basic idea is to find and update a sublist with $A_i.m \geq B_j.m$ efficiently, or a sublist with $B_u.m \geq A_v.m$ efficiently.

```
ML-SearchSublist(list, item)
    cut ← list.head
    item1 ← NIL
    for i ← MaxLevel downto 1
    {
        while cut[i].forward[i].m ≥ item.m
        {
            cut[i].adjust[i] ← cut[i].adjust[i] + item.p
            cut[i].forward[i].p ← cut[i].forward[i].p +
                item.p
            item1 ← cut[i] ← cut[i].forward[i]
        }
        cut[i] ← item1;
    }
    return cut;
```

Fig. 6. Procedure ML-SEARCHSUBLIST.

```
ML-AppendSublist(sublist, CUT)
    for i ← MaxLevel downto 1
    {
        CUT[i].forward[i] ← sublist.head.forward[i]
        diff ← CUT[i].adjust[i] − sublist.head.adjust[i]
        CUT[i].adjust[i] ← sublist.head.adjust[i]
        item ← CUT[i]
        if i ≥ 2
        {
            while item ≠ CUT[i − 1].forward[i − 1]
            {
                item.adjust[i − 1] ← item.adjust[i − 1] + diff
                if item.forward[i − 1] ≠ NIL
                item.forward[i − 1].p ← item.forward[i − 1].p+
                    diff
                item ← item.forward[i − 1]
            }
        }
    }
```

Fig. 7. Procedure ML-APPENDSUBLIST.

Suppose initially the maxplus-lists 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 Fig. 6. It starts from the head of the list to search for the longest sublist R satisfying

$$\forall I \in R : I.m \geq q\text{item}.m.$$

During this search, the p property of every visited item is increased by item.p, and the adjust[i] of every visited item is also increased by item.p if the corresponding forward pointer is used for jumping. It returns a cut after the last item L in R.

Then, all the items (including the head) before the next item of cut[1] are moved to a new list sublist in procedure ML-EXTRACTSUBLIST. A new head of the original list is generated, and for each level i, adjust[i] in the new head is set to be adjust[i] of cut[i]. Now, we have successfully extracted the sublist. The number of forward pointers visited in ML-EXTRACTSUBLIST is O(MaxLevel).

Now, we can start to append the sublist to the maxplus-list C. The procedure ML-APPENDSUBLIST accomplishes this. The pseudocode of ML-APPENDSUBLIST is shown in Fig. 7. The argument CUT is the cut after the last item in C. From ML-SEARCHSUBLIST and ML-EXTRACTSUBLIST, we can see that the adjust fields of the head of a maxplus-list may not be equal to zero, so the most important step

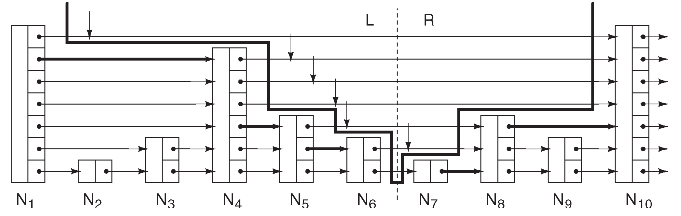

Fig. 8. Flow of ML-APPENDSUBLIST.

in ML-APPENDSUBLIST is to update the adjust fields of the items in cut.

The operations of ML-APPENDSUBLIST are illustrated in Fig. 8. Here, we want to append sublist R to L. The current CUT is $\{N_6, N_6, N_5, N_4, N_4, N_4, N_1\}$. We walk down along the stairs formed by cut. First, we set N_1.forward[7] to be N_{10}. We compute the difference diff between N_1.adjust[7] and R.head.adjust[7], and set N_1.adjust[7] to be R.head.adjust[7]. If diff is not zero, then the actual p properties of the items from N_2 to N_6 is no longer correct. We need to propagate this difference to the next level to correct the p properties. Increasing N_1.adjust[6], N_4.adjust[6], and $N_4.p$ by diff will handle that. Such a procedure successfully appends R to L at level 7. At level 6, we set N_4.forward[6] to be N_{10}, compute the difference between N_4.adjust[6] and R.head.adjust[6], and propagate the difference to level 5. Similarly, we can append R to L from level 5 to 1 using the same difference propagation technique. At the end, for each level i, adjust[i] of CUT[i] is equal to adjust[i] of the head of R. The reason that we choose to update the adjust fields of CUT instead of the adjust fields of the items in the rising stairs of R is that the forward pointers visited during the update of the CUT have already been visited by ML-SEARCHSUBLIST, which can help the time complexity analysis.

After we finish all the merge operations, we need to evaluate the p properties of items in the candidate solution lists at the root of the tree. To evaluate the p property of any item I, first we search for item I and simultaneously find the cut after I, then

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

C. Complexity Analysis

Pugh [8] proposed an algorithm to merge two skip-lists. For two skip-lists with sizes n_1 and n_2, respectively, without loss of generality, we assume that $n_1 \leq n_2$. The merge algorithm in [8] runs in $O(n_1 + n_1 \log n_2/n_1)$ expected time. Pugh [8] also claimed that in almost all cases the skip-list algorithms are substantially simpler and at least as fast as the algorithms based on balanced trees.

The merge procedure in our algorithm is similar to the merge procedure in [8], but we need to update the adjust fields in ML-APPENDSUBLIST. The running time of the skip-list merge algorithm in [8] is proportional to the number of jump operations. The running time of the maxplus merge algorithm in this paper is also proportional to the number of the involved jump operations. Considering this, we have the following theorem concerning the number of jump operations in ML-MERGE.

Theorem 1: The number of jump operations in ML-MERGE is a constant times of the number of jump operations in the skip-list merge algorithm in [8].

Proof: Let Pt(proc) denote the number of jump operations in an execution of the procedure proc. As mentioned before

$$\text{Pt}(\text{ML-ExtractSublist}) \leq 2 \times \text{MaxLevel}$$

and

$$\text{Pt}(\text{ML-AppendSublist}) \leq \text{Pt}(\text{ML-SearchSublist}).$$

So

$$\text{Pt}(\text{ML-MergeSublist}) \leq 2(\text{MaxLevel} + \text{Pt}(\text{ML-SearchSublist}))$$

while the number of jump operations in the corresponding iteration of the skip-list merge algorithm proposed by [8] is at most

$$\text{MaxLevel} + \text{Pt}(\text{ML-SearchSublist}).$$

Therefore, the number of jump operations in ML-MERGE is no more than constant times of the number of jump operations in the skip-list merge algorithm in [8]. △

Further, since the skip-list merge algorithm proposed by Pugh takes $O(n_1 + n_1 \log n_2/n_1)$ expected time [8], we have the following corollary.

Given two maxplus-lists with sizes n_1 and n_2 such that $n_1 \leq n_2$, the expected time complexity of ML-MERGE is $O(n_1 + n_1 \log n_2/n_1)$.

Therefore, the expected time complexity of the merge algorithm based on maxplus-list is the same as the time complexity based on balanced binary tree.

D. Determination of MaxLevel

Our experiments show that different values of MaxLevel may lead to different running times. For example,when MaxLevel is equal to 1, the algorithm runs fastest in balanced situations, while it becomes much worse in unbalanced situations. As shown in Fig. 1, the method based on linked-list is faster in balanced situations, while the method based on binary search trees is faster in unbalanced situations. An important property of maxplus-list is that it is a flexible data structure, that is, when MaxLevel $= 1$, it becomes a linked-list, while when MaxLevel increases, it behaves like a binary search tree. In order to get the best speedup, we use different values of MaxLevel in different situations. Here, we presented a simple strategy to determine the value of MaxLevel. Based on the results of the statistical experiments in [6], the value of P_r is always fixed at 0.25.

In the problems of floorplan, technology mapping, or buffering, the input is always a tree. We define basic elements as the realizations of basic blocks in floorplan, the mappings in technology mapping, and the buffer positions in buffer insertion. During the read of input files, we record the maximal and minimal depths of leaves in the tree, $D_{\max}$ and $D_{\min}$. Then, we can set

$$\text{MaxLevel} = \begin{cases} 1, & \text{if } D_{\min} \geq D_{\max}/2 \\ \left\lfloor \log_{\frac{1}{P_r}} (n/8) \right\rfloor, & \text{otherwise} \end{cases}$$

where n is the number of basic elements.

IV. Solution Extraction

We use the slicing floorplan problem as an example to show how to extract the best solution after the bottom-up dynamic programming procedure.

In order to record the composition of each solution, we modify our data structure to include a pointer array comp of size level in each solution. During the bottom-up calculation of nonredundant 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 a configuration graph. Similar to the adjust array, comp$[i]$ means that all the items between the current item and the first item with level larger or equal to i are composed partly by the pointed composition.

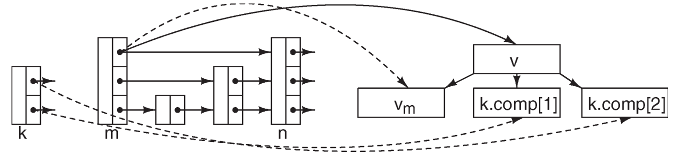

Fig. 9. Configuration graph construction. (Dashed lines represent the original pointers; solid lines represent the pointers after merge.)

We introduce one vertex into the configuration graph to represent each basic block. We can update the comp array similar with the update of the adjust array in the merge operations, and at the same time build the configuration graph. For example, as shown in Fig. 9, if the items from item m to item n in a maxplus-list are merged with an item k, then we create a new vertex v in the configuration graph, and add edges from v to the vertex pointed by the comp[3] of item m and all the vertices pointed by comp array of item k, then let comp[3] of item m point to v. Then, traverse from any vertex v in the configuration graph, and the leaves visited represent part of a solution.

After the bottom-up calculation of nonredundant candidates, we first 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. Therefore, after evaluation, each item has a single pointer pointing to its composition. Traversing the configuration graph starting from the vertex pointed by the optimal item, we get all the basic blocks in this optimal solution.

V. Experimental Results

Since maxplus merge is the only operation involved in slicing floorplanning, we use slicing floorplanning as an example to test the performance of our maxplus-list-based merge operation. 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 in C based on our merge algorithm to calculate the nonredundant candidate list at the root of each tree. We implement Stockmeyer's algorithm and download the code of Shi's merge algorithm from his web page [9] for comparison. The running time is the total time for executing each algorithm 100 times and does not include the time for reading input files and printing final results. All the experiments were run on a Linux PC with 2.4-GHz Xeon CPU and 2.0-GB memory.

For unbalanced trees, the comparison of our algorithm, Stockmeyer's algorithm, and Shi's algorithm is shown in Table I. Columns 3 and 4 are the running time of Stockmeyer's algorithm and Shi's algorithm, respectively. We use MaxLevel $= 4$, and $P_r = 0.25$ in the maxplus-list. The results indicate that our algorithm is much faster than Stockmeyer's algorithm, and with the increasing sizes of cases, it gets faster. 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 II. The fifth column is the running time of our method with MaxLevel $= 4$, $P_r = 0.25$, and the sixth column is the running time of our method with MaxLevel $= 1$, $P_r = 0$. The results shows that our algorithm with MaxLevel $= 1$ is even faster than Stockmeyer's algorithm in some

TABLE I
COMPARISON RESULTS FOR UNBALANCED TREES

Name	# Leaves	Stockmeyer's Alg. time (T_1 s)	Shi's Alg. time (T_2 s)	Our Alg.		
				time (T_3 s)	T_1/T_3	T_2/T_3
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

TABLE II
COMPARISON RESULTS FOR BALANCED TREES

Name	Leaves	Stockmeyer's T_1 (s)	Shi's T_2 (s)	Our Alg.					
				T_3 (s)	T_4 (s)	T_1/T_3	T_2/T_3	T_1/T_4	T_2/T_4
B128	128	0.033	0.083	0.033	0.033	1.00	2.52	1.00	2.52
B256	256	0.100	0.317	0.100	0.133	1.00	3.17	0.75	2.38
B512	512	0.167	0.883	0.217	0.183	0.77	4.07	0.91	4.83
B1024	1,024	0.350	1.433	0.466	0.267	0.75	3.08	1.31	5.37
B2048	2,048	0.717	2.950	0.983	0.583	0.73	3.01	1.23	5.06
BLARGE	32,768	12.730	50.390	18.550	10.700	0.69	2.72	1.19	4.71
Average						0.82	3.10	1.07	4.15

note: T_3 is the running time of our algorithm when $MaxLevel = 4$ and $P_r = 0.25$, and T_4 is the running time of our algorithm when $MaxLevel = 1$ and $P_r = 0$.

TABLE III
COMPARISON RESULTS FOR MIXED TREES

Name	# Leaves	Stockmeyer's Alg. time(T_1 s)	Shi's Alg. time(T_2 s)	Our Alg.			
				$MaxLevel$	time(T_3 s)	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

cases. This 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 item at a time. 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 III. We can see that our algorithm is always more than two times faster than Shi's algorithm. Especially for Mdata6, our strategy for determining MaxLevel improved the efficiency greatly.

VI. CONCLUSION AND FUTURE WORK

The common merge operations of solution lists in the dynamic programming technique for the slicing floorplan, technology mapping, and buffering can be formulated as maxplus merge operations. In this paper, we presented an efficient data structure called maxplus-list to represent a solution list. With parameters to adjust automatically, our maxplus merge algorithm based on maxplus-list works better than Shi [4] under all cases, unbalanced, balanced, and mix sizes. Our data structure is also simpler to implement.

Many other operations (e.g., attach wires and buffers) are involved in the buffering problem, so we plan to use maxplus-list data structure to implement all these operations, and test the performance of maxplus-list in buffering problem in the future.

REFERENCES

[1] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Inf. Control*, vol. 57, no. 2–3, pp. 91–101, 1983.
[2] K. Keutzer, "Technology binding and local optimization by DAG matching," in *Proc. Des. Autom. Conf.*, Jun. 1987, pp. 617–623.
[3] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *Proc. Int. Symp. Circuits Syst.*, 1990, pp. 865–868.
[4] W. Shi, "A fast algorithm for area minimization of slicing floorplans," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 15, no. 12, pp. 550–557, Dec. 1996.
[5] W. Shi and Z. Li, "An O($n \log n$) time algorithm for optimal buffer insertion," in *Proc. Des. Autom. Conf.*, 2003, pp. 580–585.
[6] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, Jun. 1990.
[7] K. Chaudhary and M. Pedram, "A near optimal algorithm for technology mapping minimizing area under delay constraints," in *Proc. Des. Autom. Conf.*, Jul. 1992, pp. 492–498.
[8] W. Pugh, "A Skip List Cookbook," Univ. Maryland, College Park, MD, Tech. Rep. CS-TR-2286.1, 1990.
[9] W. Shi's homepage. [Online]. Available: http://ece.tamu.edu/~wshi