An Algorithm for Zero-Skew Clock Tree Routing with Buffer Insertion

Y. P. Chen and D. F. Wong
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

We study the problem of multi-stage zero skew clock tree construction for minimizing clock phase delay and wirelength. In existing approaches clock buffers are inserted only after clock tree is constructed. The novelty of this paper lies in simultaneously perform clock tree routing and buffer insertion. We propose a clustering-based algorithm which uses shortest delay as the cost function. We show that the feasible positions for clock tree nodes and buffers can be generalized from diagonal segments (merging segments) to rectangles (merging blocks). Buffers are large components and must be placed pairwise disjointly. We also show that the problem of finding legal positions for buffers such that no buffers overlap can be formulated as a shortest path problem on graphs, and can be solved by the Bellman-Ford algorithm. By making use of the special properties of the graphs, we further speedup the Bellman-Ford algorithm. The experimental results show that our algorithm greatly outperforms the approach of inserting buffers after clock routing.

I. Introduction

Many VLSI circuits are synchronous circuits which are synchronized by clock signals. The clock frequency determines the performance of a circuit. Clock skew, defined as the maximum arrival time difference from the clock source to sinks, can degrade the clock frequency significantly. Therefore, minimizing clock skew is the most important task of clock tree routing [1, 4, 7, 10, 11, 14].

In addition to clock skew, signal delay and wirelength are also important object functions to be minimized, as in general net routing problem. Clock signal delay is also called phase delay. It appears as inter-chip skew and affects system performance. Long wire length consumes large routing area and affects the routability of other nets.

Long phase delay can be reduced by inserting clock buffers at appropriate levels of clock tree [1, 12]. There are algorithms that insert buffers after routing is completed [13, 5]. To simultaneously perform clock routing and buffer insertion is a challenging work. In this paper we propose an algorithm that performs the two tasks simultaneously. The algorithm generates a set of zero-skew clock trees, each of which are non-redundant in terms of phase delay and wirelength. We show that our algorithm provides a sound tradeoff between phase delay and wire length.

In [14], a bottom up merging scheme which ensures zero skew under Elmore delay model [9] is proposed. Later three research groups [2, 3, 8] independently propose a two-phase method (bottom-up merging phase and top-down embedding phase) for clock tree routing assuming tree topology is given. This method is known as Deferred Merging Embedding (DME) algorithm. To determine tree topology, a clustering bottom-up algorithm[7] which is based on nearest neighbor selection achieves very good results in wirelength minimization.

We adopt the two-phase DME algorithm and generalize the concept of merging segments to *merging blocks*. Loosely speaking, a merging block is a set of feasible embedding points for non-leaf vertices or center positions of clock buffers. In addition, our algorithm uses the smallest maximum delay among all possible merging as the criterion for selecting next pair to be merged in the bottom-up phase. This criterion has the same performance of nearest neighbor selection in terms of minimizing delay and wirelength. Furthermore, it cause subtrees to grow evenly. Since buffers are usually large components in a circuit. We propose a graph theoretic method to place buffers in their merging blocks (which may overlap) so that buffers do not overlap one another. We transform the problem into a special linear program which can be modeled as a *constraint graph* and solved by the shortest path algorithm. Taking advantage of the properties of the constraint graph, we further improve the time efficiency of the shortest path algorithm.

The remainder of this paper is organized as follows. In Section 2, we review some basic concepts and related works. In Section 3, we describe in details the core of our algorithm including clustering strategy, merging block concept, buffer insertion and placement. Finally experimental results are given in Section 4.

II. Reviews of Basic Concepts

A. Buffered Clock Tree

We assume that a clock tree is a binary tree in which the root is the clock source and the leaves are clock pins. Elmore delay model is used in computing the signal propagation delay of the clock tree. Let e_v denote the edge between node an internal node v and its parent. Let r_v be the resistance of edge e_v and C_v be the total capacitance of subtree T_v. The delay from a node u of the clock tree to a descendant node v of u, denoted by $d(u, v)$ is

$$d(u,v) = \sum_{u' \in \text{path}(u,v)} r_{u'} C_{u'},$$

where *path(u,v)* is the set of nodes along the unique path from u to v excluding u. Let r be the root, N be the number of clock pins, and $l_i (1 \le i \le N)$ be a leaf. Clock skew can be defined as

$$S = \max_{1 \le i,j \le N} |d(r,l_i) - d(r,l_j)|.$$

Buffers are often inserted to reduce phase delay. The model of a buffer is shown in Figure 1. In this model a buffer has three parameters: input capacitance, c_b, output resistance r_b, and internal delay d_b. Buffers of different sizes have different parameters. Figure 2 shows a buffered clock tree. Before buffer insertion, the capacitance under node v_3 is C_3, which equals

ED&TC '96
0-89791-821/96 $5.00 © 1996 IEEE

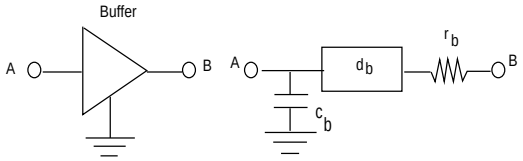

Figure 1: A clock buffer and its model.

C_{T_3}, the total capacitance of the subtree T_3 rooted by v_3. After buffer insertion, C_3 reduces to c_b. Capacitance c_b is usually much less than C_{T_3}, therefore upstream delay is reduced. The delay through the buffer is $r_b C_{T_3} + d_b$. Therefore, we should carefully select buffer size and insertion points in a tree. In this paper, we make the following assumptions. First, there is only one type of buffer. Second, buffers are leveled. For example, in Figure 2, the four buffers in the second row are level-2 buffers. Third, the variation among delays through each buffer of the same level should be minimized. Forth, the variation among delays through the subtrees rooted by the buffered nodes of the same level should be minimized. These assumptions are reasonable because they can simplify design process, reduce skew induced by buffers, and reduce skew process variation sensitivity [12, 13].

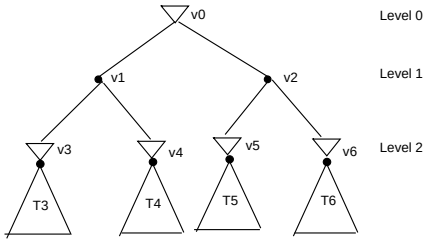

Figure 2: A buffered clock tree.

B. Zero Skew Subtree Merging

If the tree topology is given, the position of the root v of the new subtree formed after merging can be determined from the positions of the two children v_1 and v_2 of v. This is illustrated in Figure 3 and 4. Let t_v denote the delay from v to any leaf of subtree T_v. Then

$$\begin{aligned} t_v &= rl_1(\frac{cl_1}{2} + C_{v_1}) + t_{v_1} \\ &= rl_2(\frac{cl_2}{2} + C_{v_2}) + t_{v_2}, \\ C_v &= C_{v_1} + C_{v_2} + c(l_1 + l_2), \end{aligned}$$

where $l_1(l_2)$ is the wire length from v to $v_1(v_2)$, and r and c are per unit resistance and per unit capacity of the routing wire. The same equations hold for the case in Figure 4, where node v_1 and v_2 are buffered. The relation between t_{v_i} and $t'_{v_i}(i = 1, 2)$, the delay value before buffer insertion, is

$$t_{v_i} = d_b + r_b \times C_{v_i} + t'_{v_i}.$$

In [14], during each merging distances l_1 and l_2 are determined such that the delays from the new root v to all leaves are all the same. In other words, zero skew is maintained. Extra wires sometimes are needed to ensure zero skew.

The DME algorithm makes use of the fact that for each merging, the candidate positions for the new node form a diagonal segment (merging segment). Therefore, the bottom-up phase determines the merging segment $ms(v)$ for each v

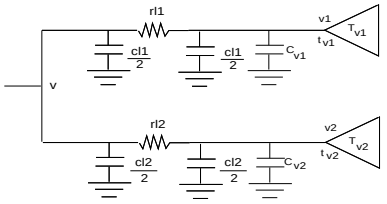

Figure 3: zero-skew merging (no buffers).

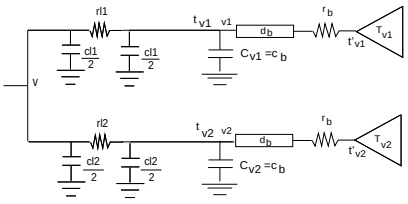

Figure 4: zero-skew merging (with buffers inserted).

and later the top-down phase determines the exact position of v on $ms(v)$ (embedding). A *Manhattan arc* is a line segment, possibly of zero length, with slope of +1 or -1. The collection of points within a fixed distance of a Manhattan arc is called a *tilted rectangular region*, or TRR. The boundary of a TRR consists of Manhattan arcs. The Manhattan arc at the center of a TRR is called its *core*. The *radius* of a TRR is the distance between its core and its boundary. It was proven [4] that all merging segments are Manhattan arcs, and $ms(v) = trr_{v_1} \cap trr_{v_2}$, where $trr_{v_1}(trr_{v_2})$ is the TRR with core $ms(v_1)(ms(v_2))$ and radius $l_1(l_2)$. The top-down phase works as follows. It determines the root position arbitrarily on the root merging segment. Let $pl(v)$ denote the embedded position of node v, and p be the parent of v. For each node v, it embeds v on a point in $ms(v) \cap trr_p$, where trr_p is the square TRR with core $pl(p)$ and radius e_v. Since both the bottom-up and top-down phases run in linear time, the DME algorithm has linear time-complexity.

The algorithm in [7] uses a bottom-up clustering strategy to generate the tree topology, while applying the DME algorithm to determine the positions of non-leaf vertices. Giving N clock pins, initially there are N merging segments which form a set S. During each iteration it finds a nearest pair $ms(v_i)$ and $ms(v_j)$ among all merging segments, and performs a zero-skew merging on $ms(v_i)$ and $ms(v_j)$ to form $ms(v)$. Then it deletes $ms(v_i)$ and $ms(v_j)$ from S and put $ms(v)$ into S. It takes $N-1$ merging steps to fix the tree topology. Upon completion of these operations the top-down phase of the DME algorithm proceeds.

III. Our Algorithm

Our algorithm uses clustering approach to determine tree topology and applies the DME algorithm to embed the internal nodes of a clock tree. We integrate bottom-up clustering and buffer insertion/placement. We start with the description of buffer insertion.

A. Buffer Insertion

The clustering process maintains a set S of subtrees, which is updated after each merging step. Whenever the size of S is reduced to 2^k for some integer k, we consider the possibility of inserting buffers at the roots of all the subtrees in S. For each $T_i \in S$, we insert a buffer at a distance l_i from the root r_i of T_i so that the delay from each leaf in the subtree to the new

root r_i' (where buffers are positioned) is equal to a constant B. For example, in Figure 5, l_1, l_2, l_3 and l_4 are the extra wires to make $d_1 = d_2 = d_3 = d_4 = B$. The value B is the longest delay among all subtrees assuming buffers are inserted without any wire elongation. For example, in Figure 5, B is the delay of T_4. Thus T_4 does not need wire elongation. This approach tries to balance the delays through the buffers and the delays of the subtrees altogether.

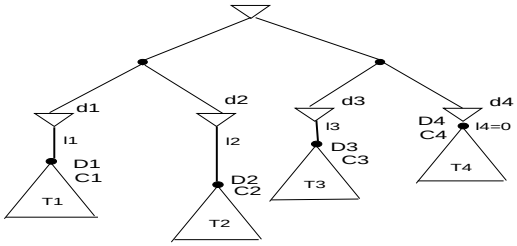

Figure 5: Buffer insertion and wire elongation.

B. Clustering Strategy

In [7], each merging operation merges two nearest neighbors in the Manhattan distance. We also adopt a similar greedy approach. Our algorithm merges two trees in a way that the merged tree has the smallest root-to-leaf delay. By using this *shortest delay cost function* we can ensure that trees grow more evenly than those using the nearest neighbor selection method.

Formally, when using the shortest delay cost function, we find a new root v with minimum $d(v)$ such that

$$d(v) = \min_{v_i, v_j \in S} \{d(v_k) | mb(v_k) = \text{zmerge}(mb(v_i), mb(v_j)\}, \quad (1)$$

where *zmerge* stands for zero-skew merge operation, and $mb(v)$ is the *merging block* of the tree rooted by v. We will define merging block in the next section. This operations has time complexity $O(|S|^2)$.

We present the bottom-up phase of our algorithm as follows.

Algorithm 1: `Bottom-Up-Clustering (S)`
Input: `A set of clock pins or subtrees` $S = \{v_i | i = 1, \ldots N\}$; `Buffer parameters` c_b, r_b `and` d_b.
Output: `A set` T `of tuples` (T_i, d_i, w_i).
Begin
  if $|S| = 1$ **then**
    `Get` $(d(v), w(v))$ $(S = \{v\})$;
    if `(Not-Redundant(`$d(v), w(v)$`))` **then**
      `enqueue(`$Bank$, $(v, d(v), w(v))$`);`
    return;
  `Find` k `such that` $2^k < |S| \leq 2^{k+1}$;
  while $|S| > 2^k$ **do**
    `Find` v `using the cost function (1);`
    $S \leftarrow S - \{v_i, v_j\}$;
    $S \leftarrow S \cup \{v\}$;
  call `Bottom-Up-Clustering(S);`
  `Copy` S `to` S';
  `Do buffer insertion and placement on` S';
  if `no buffers are overlapped` **then**
    call `Bottom-Up-Clustering(S');`
End.

Algorithm 1 is a recursive algorithm. For the basis case, the input set consists of only one tree, which is the whole clock tree. There is a pair of delay and wirelength values (d, w) associated with the root. If (d, w) is *non-redundant*, we keep (d, w) and the associated tree topology in a storage $Bank$, otherwise we discard it. A pair (d, w) is non-redundant if there does not exist another pair (d', w') in $Bank$ such that $d \geq d'$ and $w \geq w'$.

When input size is larger than one, we use the shortest delay cost function to successively merge pairs of subtrees. the operations of buffer insertion and buffer placement are applied after a certain number of merging steps as specified in the algorithm. The operation of buffer placement will be described in details in Section D. We have the following two lemmas.

Lemma 1 *The number of merging operations (zmerge) in the Bottom-Up-Clustering Algorithm (Algorithm 1) is* $O(N \log N)$, *where* N *is the number of clock pins.*

Lemma 2 *The number of the operations of inserting buffer on a tree in the Bottom-Up-Clustering algorithm is* $O(N \log N)$.

We can see that each successful buffer insertion and placement generates a new independent merging process, which is generated by the recursive call at the last line of Algorithm 1.

C. Merging Blocks

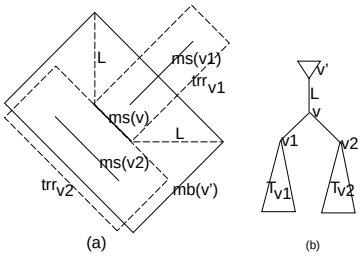

Figure 6: A merging block.

In DME algorithm, the set of feasible points for positioning a node v is a diagonal segment $ms(v)$. In our algorithm, the set of feasible points can be generalized to a rectangle area which we call *merging block*. It originates from buffer insertion as illustrated in Figure 6. Assume that the tree under root v has no buffers. Then the feasible points for v forms a segment $(ms(v))$ as usual. Now that a buffer is inserted, and an extension wire of length L is applied, the buffer may be positioned anywhere within the TRR with core $ms(v)$ and radius L. The TRR is called the merging block of buffered root v' denoted by $mb(v')$. We consider diagonal segments and points as special cases of merging blocks. Let *zero-seg-merge*(s_1, s_2) denote the merge operation in the original DME algorithm. We now describe the merge operation (zmerge) of merging blocks. Let $mb(v_1)$ and $mb(v_2)$ be the merging blocks of trees rooted by v_1 and v_2. Let $dist(l_1, l_2)$ be the distance between segments l_1 and l_2.

Case 1: $mb(v_1)$ and $mb(v_2)$ do not overlap. Let b_1, b_2, b_3 and b_4 be the four boundaries of $mb(v_1)$, and c_1, c_2, c_3 and c_4 be the four boundaries of $mb(v_2)$. Note that it is possible that $b_i = b_j$ (or $c_i = c_j$) for some i and j, which is the case that the merging block is degenerated. We have zmerge$(mb(v_1), mb(v_2))$ = zero-seg-merge(b_k, c_l), where $(\text{dist}(b_k, c_l) = \min_{1 \leq i,j \leq 4}\{\text{dist}(b_i, c_j))\}$. (See Figure 7(a).)

Case 2: $mb(v_1)$ and $mb(v_2)$ overlap, and neither of v_1 and v_2 is a buffered root. Assume $d(v_1) \geq d(v_2)$. The parent v has delay $d(v) = d(v_1)$, and v can reside on $mb(v_1)$. Let l be the length of wire that connects v to v_2. Enlarge $mb(v_2)$ by l and obtain a TRR $trr_{v_2'}$. Then $mb(v)$ =

$mb(v_1) \cap trr_{v_2'}$. (See Figure 7(b).) Note that Case 2 includes the situation that one merging block totally encloses the other.

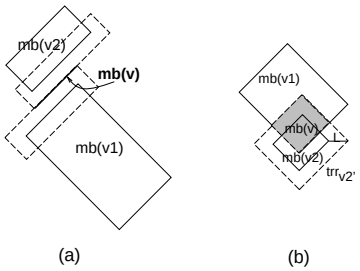

Figure 7: Two cases of determining new merging blocks

D. A Graph-Theoretic Approach to Buffer Placement

Clock buffers are large components in a circuit. It is essential that the buffer positions determined by our algorithm do not cause buffers to overlap. The merging block associated with a buffered root is the possible placement area for that buffer. In order to reserve enough space for buffers at lower levels of the clock tree, whose merging blocks are determined later than buffers at the higher level, we need to shrink the merging blocks as soon as their parents are determined. Therefore, in Case 1 of Section 3.3, we shrink $mb(v_1)$ to a single segment b_k, and shrink $mb(v_2)$ to c_l.

The higher level merging blocks associated with buffers are considered as obstacles to low level merging blocks. During the bottom-up phase, we keep a set B of obstacles. For each buffered node $v \in S$, the newly generated merging block $mb(v)$ should be shrunk to $mb(v) - B$ immediately. This shrinking process can be done efficiently by constructing a grid which divides the chip area into a set of small squares. Each square is associated with a list of elements in B which overlap with the square. So we only have to test the squares which overlap with $mb(v)$ to perform the shrinking process. If by doing so $mb(v)$ become $\emptyset$, then it is impossible to insert buffers at the roots of the subtrees in S. We abort S and later try another set with smaller number of subtrees.

The above strategy avoid the overlap between the buffers of the current level and the buffers of the higher (previous) levels. To avoid overlap among the buffers of the current level, we fix an exact position for a buffer if its corresponding merging block overlap with other merging blocks. It can be considered as shrinking the merging block to a point. Although this may result in increased delay or wirelength of clock tree, buffers are guaranteed legal positions. If we rotate all merging blocks by $45°$, and shift them to the first quadrant, all merging blocks become rectangles with rect-linear boundaries. Let W be the width of a buffer. We assume a buffer has a square shape. After rotation, each buffer becomes a rhombus. We find the smallest square that encloses the rhombus, which has dimension $W' \times W'$, where $W' = \sqrt{2}W$. We enlarge each side of a rectangle by W', since buffer centers can be positioned at the boundary of merging blocks. Now we can formally define the problem as follows.

Problem 1 *We are given M rectangles with rect-linear boundaries. Each is large enough to contain a square with dimension $W' \times W'$, and each overlaps with at least one other rectangle. Find a placement of all the squares such that each is contained by a rectangle, and no two squares overlap. (See Figure 8 for an example.)*

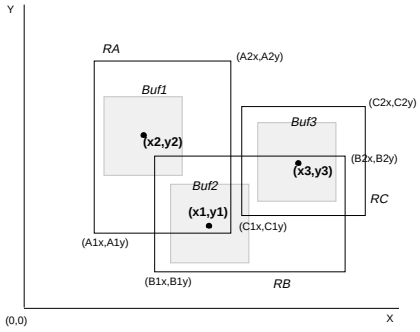

Figure 8: An example of Problem 1.

Let B_1 and B_2 be two squares with center coordinates (x_1, y_1) and (x_2, y_2). They do not overlap if and only if $|x_1 - x_2| \geq W' \vee |y_1 - y_2| \geq W'$, which is equivalent to the disjunction of the following four inequalities. This set is called *separation set*.

$$x_1 - x_2 \leq -W' \quad (2)$$
$$x_2 - x_1 \leq -W' \quad (3)$$
$$y_1 - y_2 \leq -W' \quad (4)$$
$$y_2 - y_1 \leq -W' \quad (5)$$

For example, in Figure 8, there are two pairs of overlapped rectangles. The above four inequalities can specify that Buf_1 and Buf_2 do not overlap. We need another four inequalities to specify that Buf_2 and Buf_3 do not overlap.

$$x_1 - x_3 \leq -W' \quad (6)$$
$$x_3 - x_1 \leq -W' \quad (7)$$
$$y_1 - y_3 \leq -W' \quad (8)$$
$$y_3 - y_1 \leq -W'. \quad (9)$$

The square Buf_1 is contained by R_A (specified by inequalities (12)-(15)), while Buf_2 is contained by R_B (specified by inequalities (16)-(19)), and Buf_3 is contained by R_C (specified by inequalities (20)-(23)). We introduce a pair (x_0, y_0) of variables corresponding to the origin $(0, 0)$. Inequalities (10) and (11) are exactly $x_0 = y_0$. Note that $w = W'/2$. We call the inequalities (10)-(23) the *confinement* set.

$$x_0 - y_0 \leq 0 \quad (10)$$
$$y_0 - x_0 \leq 0 \quad (11)$$
$$x_0 - x_1 \leq -(A_{1x} + w) \quad (12)$$
$$x_1 - x_0 \leq (A_{2x} - w) \quad (13)$$
$$y_0 - y_1 \leq -(A_{1y} + w) \quad (14)$$
$$y_1 - y_0 \leq (A_{2y} - w) \quad (15)$$
$$x_0 - x_2 \leq -(B_{1x} + w) \quad (16)$$
$$x_2 - x_0 \leq (B_{2x} - w) \quad (17)$$
$$y_0 - y_3 \leq -(B_{1y} + w) \quad (18)$$
$$y_3 - y_0 \leq (B_{2y} - w) \quad (19)$$
$$x_0 - x_3 \leq -(C_{1x} + w) \quad (20)$$
$$x_3 - x_0 \leq (C_{2x} - w) \quad (21)$$
$$y_0 - y_3 \leq -(C_{1y} + w) \quad (22)$$
$$y_3 - y_0 \leq (C_{2y} - w) \quad (23)$$

Note that all of the inequalities from (2) to (23) have the form $x_j - x_i \leq a_{ij}$. Problem 1 is now transformed to a type of linear program as follow.

Problem 2 *Given a confinement set B (e.g. constraints (10) - (23)), and a group of k separation sets $D_1, D_2, \ldots, D_k$ (e.g. constraints (2) - (5), and constraints (6) - (9)), determine a set of values for all of the variables (e.g. x_1, y_1, x_2, y_2, x_3 and y_3) that satisfy B and at least one of the constraints in each D_i or determine that no such values exists.*

There is a natural correspondence between Problem 2 and the single-source shortest-paths problem on a graph [6]. We construct a directed graph $G = (V, E)$ called *constraint graph*. We create a vertex for each variable. Let $v(x)$ denote the vertex that corresponds to variable x. For each constraint $x_j - x_i \leq a_{ij}$, we create an edge from vertex $v(x_i)$ to $v(x_j)$ with weight a_{ij}. We designate x_0 as the source. For example, if we choose (4) from the first separation set, and choose (6) from the second separation set, together with all inequalities from the confinement set the constraint graph is shown in Figure 9(a). If G contains negative cycle, then no solution exists. Otherwise, the sums of weights of the shortest paths from x_0 to each node is a solution to Problem 2. It is obvious that the solution includes $x_0 = y_0 = 0$. For the shortest path problem, we can use Bellman-Ford algorithm [6], which runs in time $O(|V||E|)$. The size of vertex set $|V| = 2m+2$ and the size of edge set $|E| = 4m+k+2$, where m is the number of rectangles (squares), and k is the number of pairs of overlapped rectangles. For problem 2, in the worst case, we need to run Bellman-Ford algorithm 4^k times, which leads to total time complexity $O(4^k(2m+2)(4m+k+2))$. To improve the efficiency, we check beforehand to see if some of the inequalities in separation sets are impossible for a given rectangle pair. For example, in Figure 8, inequalities (2) and (5) are not possible.

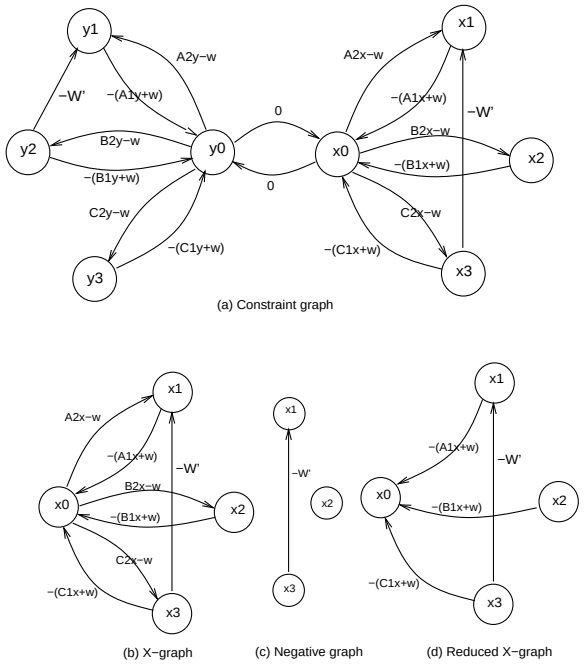

Figure 9: Examples of constraint graph, etc.

We explore the characteristics of the constraint graphs and develop the following techniques to further reduce the running time by reducing the problem size for each run of the shortest path algorithm, and by performing some preprocessing to avoid unnecessary runs of the shortest path algorithm.

Lemma 3 *Problem 1 can be solved by constructing the constraint graph and removing the edges $x_0 \to y_0$ and $y_0 \to x_0$ and solving the single source shortest path problems on the two sub-graphs with source x_0 and y_0 respectively.*

Now we only need to consider the two sub-graphs independently. We call them X-graph and Y-graph (Figure 9-(b)). For simplicity, the following discussion is only in terms of X-graph. Nevertheless, it applies to Y-graph as well. Let G_x denote a X-graph. We construct a sub-graph G_{nx} of G_x, whose vertex set is $V(G_{nx}) = V(G_x) - x_0$ and edge set is $E(G_{nx}) = \{x_i \to x_j | i \neq j \text{ and } i, j \neq 0\}$. In other words, $E(G_{nx})$ corresponds to the set of edges from separation sets. We call G_{nx} *negative graph* (Figure 9-(c)), since all of its edges have negative weights. The following lemma is obvious.

Lemma 4 *All cycles in the negative graph are negative cycles.*

A depth-first search on the negative graph creates a *depth-first forest* and classifies all edges into the category of *tree edges*, *forward edges*, *cross edges* and *back edges*[6]. The existence of back edges implies that there are cycles and they must be negative cycles (Lemma 4). Therefore, instead of running the Bellman-Ford algorithm to detect negative cycles, we can first perform depth-first search on the negative graph to detect back edges. If back edges exist, then we know that there is no solution for the current combination of constraints.

We remove all outgoing edges of x_0 from the X-graph and it becomes a *reduced X-graph* (Figure 9-(d)). We present the *Modified Bellman-Ford* algorithm which applies to reduced X-graphs. As in the original Bellman-Ford algorithm, $d(v)$ is the shortest-path estimate of vertex v.

Algorithm 2: `Modified-Bellman-Ford (MBF)`(G'_x)
Input: `A reduced` X-`graph (`Y-`graph);`
Output: `The shortest path length` $d(v)$ `for each` $v \in V(G'_x)$`;`
`or ''FALSE'' if negative cycle detected`
Begin
$d(x_0) \leftarrow 0$ `and for every` $v \neq x_0$, $d(v) \leftarrow W(x_0 \to v)$.
for $i \leftarrow 1$ `to` $|V(G'_x)| - 1$ `(* Main Loop *)`
do for `each edge` $u \xrightarrow{e} v \in E(G'_x)$ **do**
if $d(v) > d(u) + w(e)$ **then**
$d(v) \leftarrow d(u) + w(e)$
if $d(x_0) < 0$
return `''FALSE''` `(*End of Main Loop.*)`
for `each edge` $u \xrightarrow{e} v \in E(G'_x)$
do if $d(v) > d(u) + w(e)$
then return `''FALSE''`
return `''TRUE''`
End.

Compared to the original algorithm, we modify the initialization step such that $d(x_0)$ is 0 as usual, but $d(x_i)(i \neq 0)$ is initialized to, instead of ∞, $W(x_0 \to x_i)$ for the edge $x_0 \to x_i$ in the X-graph. Also we add a test: $d(x_0) < 0$ for early termination.

Theorem 1 *The shortest paths from x_0 to every vertex in the X-graph can be obtained by running the Modified Bellman-Ford algorithm on the reduced X-graph. Assume that the corresponding negative graph is cycle free.*

In summary, the algorithm to find a placement of all buffers in their enclosing merging blocks is presented as follows. Let G_{cx} and G_{cy} be the sub-graphs of X-graph and Y-graph respectively with the outgoing edges of x_0 and y_0 removed and the edges corresponding to the separation sets removed. Let G'_x and G'_y be the reduced X-graph and reduced Y-graph respectively.

Algorithm 3: Buffer-Placement (M)
Input: A list of merging blocks M.
Output: ''False'' if not placeable. Otherwise a list of buffer center coordinates (x_i, y_i)'s
Begin
  $Q \leftarrow \emptyset$;
  Rotate all merging blocks by 45°;
  Prepare all inequalities;
  Construct G_{cx} and G_{cy};
  for each selection of inequalities from the separation sets **do**
    Construct the negative graphs G_{nx} and G_{ny};
    Do depth-first search on G_{nx} and G_{ny};
    if both G_{nx} and G_{ny} have no cycles **then**
      Add the edges of G_{nx} to G_{cx} and form G'_x;
      call MBF(G'_x);
      if MBF return ''TRUE'' **then**
        Add the edges of G_{ny} to G_{cy} and form G'_y;
        call MBF(G'_y);
        if MBF return ''TRUE'' **then**
          Rotate each $(d(x_i), d(y_i))$ by -45° and put into list Q;
          return Q;
End.

We have finished the discussion of the bottom-up phase, with buffer placement being the most sophisticated step. We now describe the top-down phase.

E. Top-Down Embedding

At the end of the bottom-up clustering phase, a list of trees with non-redundant delay and wirelength values are created. After a user selects the desired tree based on its delay and wirelength value, the top-down phase proceeds. We state it as follows.

Algorithm 4: Top-Down-Embedding (v)
Begin
  if v has no child **then**
    return;
  if v is the root **then**
    Choose any $pl(v) \in mb(v)$;
  else
    Let p be the parent node of v;
    Construct trr_p as follows:
    $core(trr_p) \leftarrow pl(p)$;
    $radius(trr_p) \leftarrow |e_v|$;
    if v is not buffered **then**
      Choose any $pl(v) \in mb(v) \cap trr_p$;
      Let v_1 and v_2 be the two children of v;
      call Top-Down-Embedding (v_1);
      call Top-Down-Embedding (v_2);
    else
      Choose any $pl(v) \in mb(v) \cap trr_p$;
      Find the original unbuffered node v_0 and its merging block $mb(v_0)$;
      Construct trr_v such that:
      $core(trr_v) \leftarrow pl(v)$;
      $radius\ (trr_v) \leftarrow l(v_0)$;
      Choose any $pl(v_0) \in mb(v_0) \cap trr_v$;
      Let v_1 and v_2 be the two children of v_0;
      call Top-Down-Embedding (v_1);
      call Top-Down-Embedding (v_2);
End.

When a unbuffered node v visited, the operation is the same as the top-down phase of the DME algorithm, except that it chooses embedding points $pl(v)$'s from merging blocks instead of merging segments. When v is a buffered node, a wire with length l (can be 0) is inserted between v and v_0, the original unbuffered node. Therefore, if $l > 0$, we need to embed two nodes v and v_0, and buffer center is placed at v. The top-down phase can be seen as embedding an incomplete binary tree, some of whose internal nodes (corresponding to buffer centers) only have one child. The time complexity of Algorithm 4 is $O(N)$ (N is the number of clock pins).

IV. Experimental Results

Our algorithm was implemented in C++ on the Sun SPARC-5 workstation. We ran experiments on benchmark data r1-r5 [14]. The parameters of clock buffers are those used in [12]. The output resistance is 122 Ω, input capacitance is 400fF, and the buffer internal delay is 75 ps. We assume the dimension of a clock buffer is 400 microns × 400 microns. We also assume the clock diver at the root has 10 Ω output resistance. For comparison, we used the clock tree topology of r1-r5 obtained by the algorithm of [7], and exhaustively tried all possible combinations of levels in order to insert buffers. In other words, the counterpart performs buffer insertion after tree topology is determined, while our algorithm simultaneously determines tree topology and positions of buffers.

The results are shown in Figure 10-14. In each figure the vertical axis is phase delay (D), and the horizontal axis is wirelength (W). The curve labeled "Ours" represents the results produced by our algorithm, and the curve labeled "Post" (for post-processing) represents the results from the counterpart. The number associated with each point is the number of buffers used. They show that our algorithm produces significantly better results than the counterpart in the sense that most points of the counterpart are redundant to the points of our algorithm. Especially, when more buffers are inserted, the difference becomes larger. The only exception is in the case of r3, where our algorithm and the counterpart obtain comparable results. The reason is that the counterpart by chance generates a clock tree whose subtrees at the same level are balanced in both delay and capacity. However, for r3, our algorithm still performs better when more buffers are inserted.

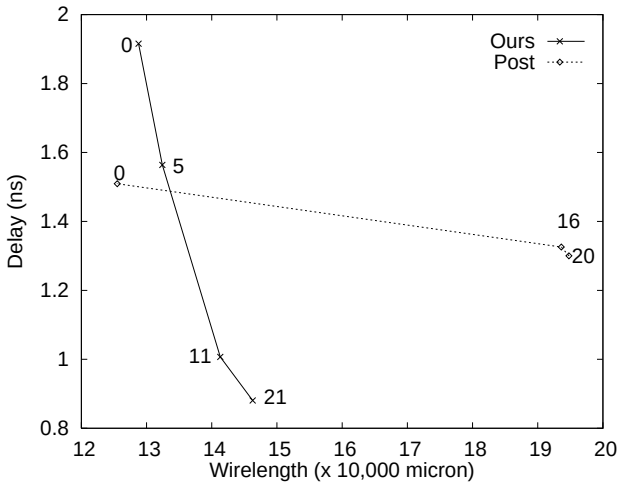

Figure 10: r1 (267 pins)

Acknowledgment

The authors would like to thank R. S. Tsay of ArcSys Inc. for providing benchmark data, and M. Edahiro of C&C Research Lab., NEC Corp. for providing program and benchmark data.

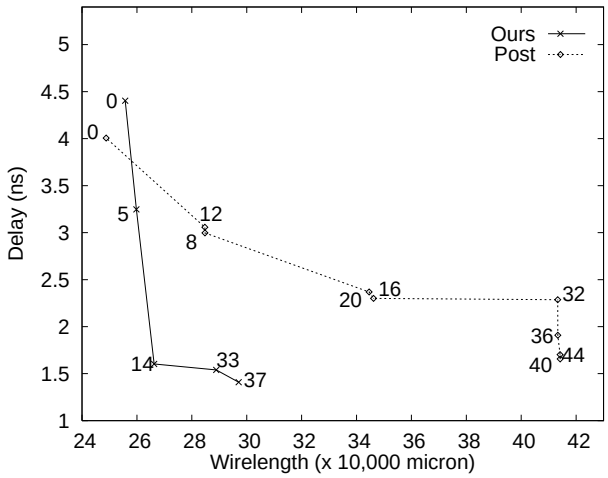

Figure 11: r2 (598 pins)

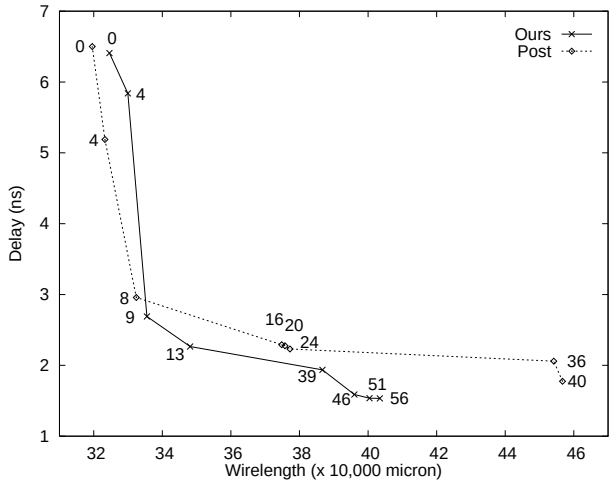

Figure 12: r3 (862 pins)

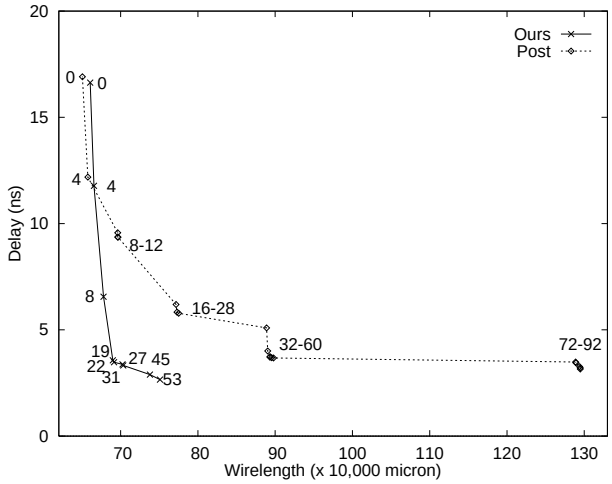

Figure 13: r4 (1903 pins)

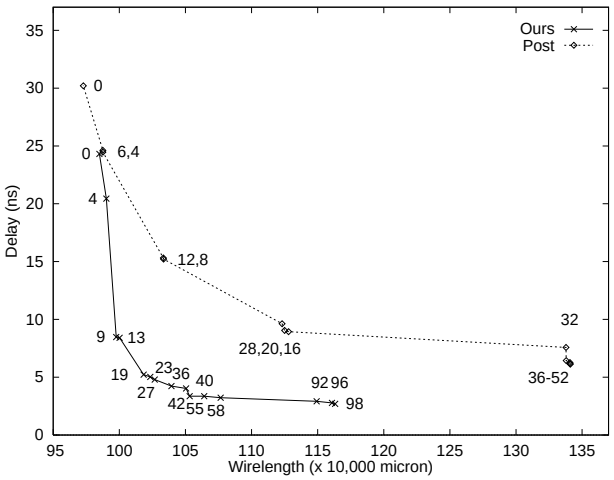

Figure 14: r5 (3101 pins)

References

[1] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley Publishing Co., 1990.

[2] K.D. Boose and A.B. Kahng, "Zero-skew clock routing trees with minimum wirelength," *Proc. IEEE Intl. Conf. on ASIC*, pp.1.1.1-1.1.5, 1992.

[3] T. Chao, Y. Hsu and J. Ho, "Zero skew clock net routing," *Proc. ACM/IEEE Design Automation Conference*, pp.518-523, 1992.

[4] T. Chao, Y. Hsu, J. Ho, K.D. Boose and A.B. Kahng, " Zero skew clock routing with minimum wirelength," *IEEE Trans. on Circuits and Systems* 39(11), pp.799-814, 1992.

[5] J. D. Cho and M. Sarrafzadeh, "A buffer distribution algorithm for high-speed clock routing," *Proc. ACM/IEEE Design Automation Conference*, pp.537-543, 1992.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, The MIT Press, 1991.

[7] M. Edahiro, "Clustering-based optimization algorithm in zero-skew routings," *Proc. ACM/IEEE Design Automation Conference*, pp.612-616, 1993.

[8] M. Edahiro, "Minimum skew and minimum path length routing in VLSI layout design ," *NEC Research and Development* 32(4), pp.569-575, 1991.

[9] W.C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *J. Applied Phys.*, 19(1):55-63, 1948.

[10] M. A. B. Jackson, A. Srinivasan and E. S. Kuh, "Clock routing for high-performance ICs," *Proc. ACM/IEEE Design Automation Conference*, pp.573-579, 1990.

[11] A Kahng, J. Cong and G. Robins, "High-performance clock routing based on recursive geometric matching," *Proc. ACM/IEEE Design Automation Conference*, pp.322-327, 1991.

[12] S. Pullela, N. Menezes, J. Omar and L. T. Pillage, "Skew and delay optimization for reliable buffered clock trees," *Proc. IEEE Intl. Conference on Computer-Aided Design*, pp.556-562, 1993.

[13] G. Tellez and M. Sarrafzadeh, "Clock period constrained minimal buffer insertion in clock trees," *Proc. IEEE Intl. Conference on Computer-Aided Design*, pp.219-223, 1994.

[14] R. Tsay, "Exact zero skew," *Proc. IEEE Intl. Conference on Computer-Aided Design*, pp.336-339, 1991.