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. The experimental results show that our algorithm greatly outperforms the approach of inserting buffers after clock routing.

1 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, 11, 12, 16].

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, which was pointed out in [11]. 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, 14]. There are algorithms that insert buffers after routing is completed [15] or clock net topology

*This work was partially supported by the Texas Advanced Research Program under Grant No. 003658459.

is determined [5]. Few efforts have been made to simultaneously perform clock routing and buffer insertion. 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 [16], a bottom up merging scheme which ensures zero skew under Elmore delay model [10] 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[1] assuming tree topology is given. This method is called Deferred Merging Embedding (DME) algorithm. To determine tree topology, in [7, 9], a clustering bottom-up algorithm which bases on nearest neighbor selection is used. In terms of wirelength, it achieves the best result compared with other previous works.

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. In addition, 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 running time of the shortest path algorithm.

The remainder of this paper is organized as follows. In Section 2, we introduce some basic concepts, and briefly describe related works. Our algorithm is presented in Section 3. We describe in details our buffer insertion approach, clustering strategy, the concept of merging blocks, buffer placement, and top-down embedding. Experimental results are given in Section 4.

2 Reviews of Basic Concepts

2.1 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 (Figure 1). Each edge of the tree represents a routing segment. Elmore delay model is used

[1] Note that although we use the term routing, in fact most clock routing algorithms only determine the positions of all non-leaf vertices of the clock tree.

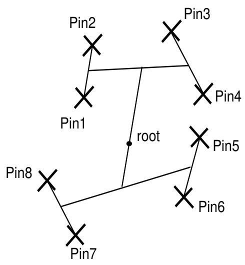

Figure 1: A simple clock tree.

in computing the signal propagation delay of the clock tree. Hence the clock tree is modeled as an RC tree in which each leaf or internal node is associated with a capacitance, and each edge is associated with a resistance. Let e_v denote the edge between 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

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

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 $l_i (1 \leq i \leq N)$ be a leaf. Clock skew can be defined as

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

Buffers are often inserted to separate a clock tree. They serve as *capacitance separators*, and therefore

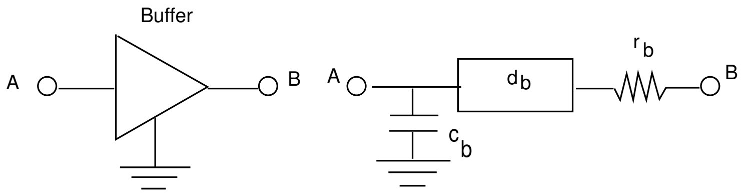

Figure 2: A clock buffer and its model.

reduce phase delay. The model of a buffer is shown in Figure 2. There are three parameters associated with a buffer. They are input capacitance c_b, output resistance r_b, and internal delay d_b. Figure 3 illustrates how buffers reduce delay. Before buffer insertion, the capacitance of node v_3 is C_3, which equals C_{T_3}, the total capacitance of the subtree T_3 rooted by v_3. After buffer insertion, C_3 becomes c_b. Capacitance c_b is usually much less than C_{T_3}, therefore upstream delay is reduced. However, the inserted buffer does incur a delay of $r_b C_{T_3} + d_b$ and so there is a trade-off. In this paper, we make the following assumptions.

1. There is only one type of buffer.
2. Buffers are inserted at all nodes of the same level. For example, in Figure 3, all nodes on level 2 are buffered.
3. The variation among delays through each buffers of the same level should be minimized.
4. The variation among delays through the subtrees rooted by the buffered nodes of the same level should be minimized.

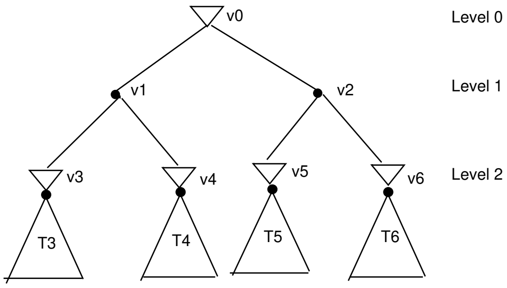

Figure 3: A buffered clock tree.

The above assumptions are reasonable because they can simplify design process, reduce skew induced by buffers, and reduce skew process variation sensitivity [14, 15].

2.2 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 4 and 5. The distributed RC line model we use is the π model. 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 material. The same equations hold for the case in Figure 5, 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

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

According to Assumption 2, it does not occur that only one subtree is buffered. In [16], during each

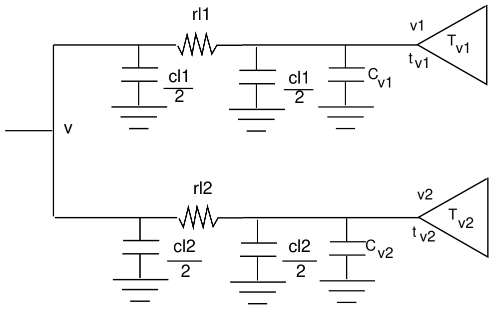

Figure 4: zero-skew merging (no buffers).

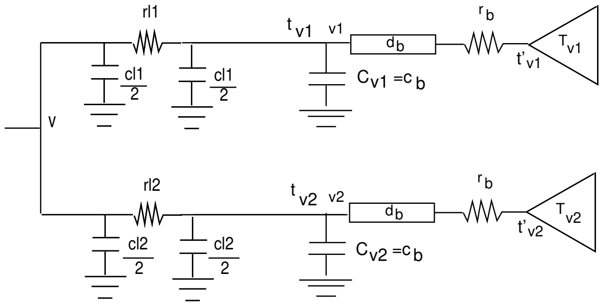

Figure 5: zero-skew merging (with buffers).

merging distances l_1 and l_2 are determined such that zero skew is maintained at the merged subtree. Let l be the Manhattan distance between v_1 and v_2 (In this paper, distance is measured in terms of Manhattan distance). Normally $l_1 + l_2 = l$. If not, then it is implied that the delay and capacitance differences between T_{v_1} and T_{v_2} is large. Hence it is required that either $l_1 > l$ and $l_2 = 0$ or $l_1 = 0$ and $l_2 > l$. In other words, wire elongation is adopted to balance delay.

The DME algorithm[4] makes use of the fact that on each merging, the solution is a diagonal segment instead of a single point. Therefore, the bottom-up phase needs only to determine the merging segment $ms(v)$ for each v and defer embedding until the top-down phase. In the top-down phase, all tree nodes are embedded on their associated merging segments. 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. Since Manhattan arc can be a point (when it corresponds to a clock pin), TRR can be a rhombus. 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, 9] uses a bottom-up clustering strategy to generate the tree topology, while using 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. For 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 append $ms(v)$ to S. This method needs $N-1$ operations of finding the nearest pair to determine the merging segment of the root. Upon completion of these operations the top-down phase of the DME algorithm proceeds. Since Delaunay triangulation is used in [7], each nearest pair selection takes $O(N \log N)$ time. Consequently the overall algorithm takes $O(N^2 \log N)$ time. In [9], the clustering process is improved and runs in linear time, though this is at the cost of some performance degradation.

3 Our Algorithm

Our algorithm uses clustering approach to determine tree topology and applies the DME algorithm to embed non-leaf vertices of a clock tree. It consists of two phase - bottom-up clustering and top-down embedding. During the first phase, we try to insert buffers at appropriate steps of the phase. In the following sections we describe our algorithm in details with respect to its components - buffer insertion, clustering strategy, merging blocks, buffer placement and top-down embedding. The first four components belong to the bottom-up phase, and the last one belongs to the top-down phase.

3.1 Buffer Insertion

During the bottom-up clustering phase, each merging operation generates a new set S of subtrees T_i's, whose size is one less than the size of its immediate predecessor. The value of S is initialized to the set of clock pins. For some set S, we consider the possibility of inserting buffers at the roots of all the subtrees in S (Assumption 2). In order to satisfy Assumptions 3 and 4, we use wirelength elongation. 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 at the new root r_i' (where buffers are positioned) is equal to a constant B. For example, in Figure 6, l_1, l_2, l_3 and l_4 are the extra wires to make $d_1 = d_2 = d_3 = d_4 = B$. For each set $S = \{T_1, \ldots, T_m\}$, we use the following procedure to compute $p(S) = \sum_{i=1}^{m} l_i$, the penalty of inserting buffers at the roots of subtrees in S. The procedure refers to notations α and β, which are the per unit resistance and per unit capacitance of routing wire, respectively.

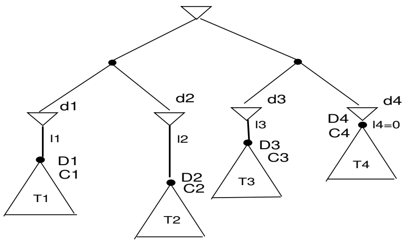

Figure 6: Buffer insertion and wire elongation.

Algorithm 1: Buffer-Insertion-Penalty(S)
Input:A set S of trees $T_i (i = 1, \ldots, m)$ with root r_i,
root delay D_i, and load capacity C_i.
Output: $p(S)$.
Begin
$B \leftarrow \max_{i=1,\ldots,m}(d_b + r_b C_i + D_i)$;
$p \leftarrow 0$;
for each $T_i \in S$ **do begin**
$l_i \leftarrow$ the positive root of equation
$\frac{\alpha\beta}{2}x^2 + (\alpha C_i + r_b\beta)x + d_b + r_b C_i + D_i = B$;
$p \leftarrow p + l_i$;
end;
return p;
End.

Note that in the procedure, the uniform delay B after buffer insertion is first computed. It is not hard to see that B is associated with the tree that does not need wire elongation (e.g. T_4 in Figure 6). Once the value of B is determined, the procedure proceeds to compute the elongated wirelength l_i for T_i. The left side of the equation is the delay after inserting wire of length x between buffer and the original root of tree T_i, which should be equal to B. The penalty $p(S)$ is the sum of the length of the extra wires. If buffers are decided to be inserted at the roots of subtrees in S, then each l_i value computed in Algorithm 1 is used to elongate wires.

Usually a clock tree is a near-balanced tree. Therefore, a candidate set S for inserting buffers has approximately 2^k trees for some integer k, and all buffers inserted are level k buffers. We select S, from the sets with size between $2^k - H$ and $2^k + H$, so that the corresponding penalty $P(S)$ is the minimized. Here, H is a small positive constant. We use 4 as the value of H in our experiment.

3.2 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 in 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

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

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 2: 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, and constant H
Output:A set T of tuples (T_i, d_i, w_i)
Begin
  if $|S| = 1$ **then begin**
    Get $(d(v), w(v))$ associated with the single element v of S;
    if (Not-Redundant$(d(v), w(v))$) **then begin**
    enqueue($Bank$, $(v, d(v), w(v))$);
    return "TRUE";
  end;
  Determine k such that $|2^{k+1} - |S||$ is minimum;
  $L \leftarrow \emptyset$;
  while $|S| > \max(2, 2^k - H)$ **do begin**
    Find v using the shortest delay cost function such that
    $d(v) = \min_{v_i, v_j \in S} \{d(v_k) | mb(v_k) = \text{zmerge}(mb(v_i), mb(v_j))\}$;
    $S \leftarrow S - \{v_i, v_j\}$;
    $S \leftarrow S \cup \{v\}$;
    if $2^k + H \geq |S| \geq 2^k - H$ **then begin**
      $b(S) \leftarrow$ Buffer-Insertion-Penalty(S);
      enqueue(L, $b(S)$, S);
    end;
  end;
  call Bottom-Up-Clustering(S);
  Determine S' such that $b(S') = \min_{SS \in L} \{b(SS)\}$;
  Do buffer insertion and placement on S';

if no buffers are overlapped **then**
call Bottom-Up-Clustering(S');
25. **End**.

Bottom-Up-Clustering 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 merging process stops when set size is reduced by about a half. The merging process is followed by buffer insertion and buffer placement. The details of buffer placement will be described later.

Theorem 1 *The number of merging operations (zmerge) in the Bottom-Up-Clustering Algorithm (Algorithm 2) is $O(N \log N)$, where N is the number of clock pins.*

Proof Without loss of generality, we assume $N = 2^k$ for some integer k. Let $T(n)$ be the number of merge for a set of size n. We have,

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ n + 2T(n/2) & \text{otherwise} \end{cases} \tag{1}$$

The recursive case follows from the strategy we use in selecting set for buffer insertion. To be more exact, it should be $T(n) = n + x + T(n/2) + T(n/2 - x)$ or $T(n) = n - y + T(n/2) + T(n/2 + y)$ for some $x, y \leq H$ (defined in previous section). We assume H is a small integer. In the average case, the recurrence relation in (1) is true. Therefore, $T(n) = O(n \log n)$ [6]. □

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

The proof is similar to that of Theorem 1.

3.3 Merging Blocks

In the previous works such as 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. It is illustrated in Figure 7. Assume that the tree under root v has no buffers. Then the feasible points for v forms a segment

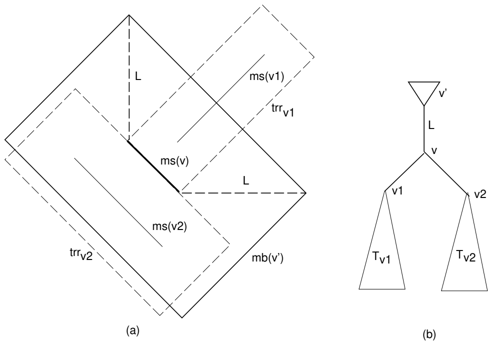

Figure 7: A merging block.

$(ms(v))$ 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 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 $\text{zmerge}(mb(v_1), mb(v_2)) = \text{zero-seg-merge}(b_k, c_l)$, where $(\text{dist}(b_k, c_l) = \min_{1 \le i,j \le 4}\{\text{dist}(b_i, c_j))\}$. (See Figure 8.)

Case 2: $mb(v_1)$ and $mb(v_2)$ overlap, and neither of v_1 and v_2 is a buffered root. Assume $d(v_1) \ge d(v_2)$. 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 9.) Note that Case 2 includes the situation that one merging block totally encloses the other.

3.4 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

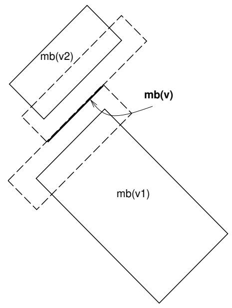

Figure 8: Case 1.

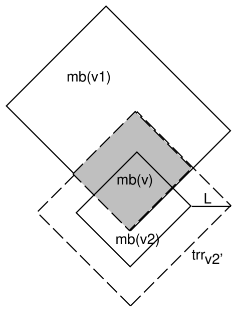

Figure 9: Case 2.

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. 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 rectangle by $W'/2$, since buffers 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 10 for an example.)*

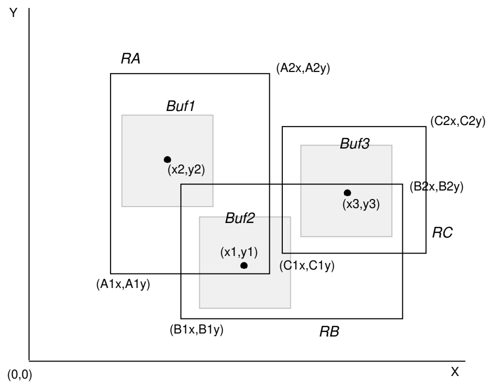

Figure 10: An example of Problem 1.

An undirected graph $G = (V, E)$ is used to model the set of rectangles. Each vertex corresponds to a rectangle, and there exist an edge between v_i and v_j if the two corresponding rectangles overlap. We call G *overlap graph*. In our application, the number of connected sub-graphs of a overlap graph is not large. Therefore, we can use the following approach to obtain exact solution.

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*.

$$x_1 - x_2 \leq -W' \tag{2}$$

$$x_2 - x_1 \leq -W' \tag{3}$$

$$y_1 - y_2 \leq -W' \tag{4}$$

$$y_2 - y_1 \leq -W' \tag{5}$$

For the example in Figure 10, there are two pairs of overlapped rectangles. The above four inequalities are again used to 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.

$$
\begin{aligned}
x_1 - x_3 &\leq -W' && (6)\\
x_3 - x_1 &\leq -W' && (7)\\
y_1 - y_3 &\leq -W' && (8)\\
y_3 - y_1 &\leq -W'. && (9)
\end{aligned}
$$

Square Buf_1 is contained by R_A, square Buf_2 is contained by R_B, and square Buf_3 is contained by R_C. We introduce a pair (x_0, y_0) of variables corresponding to the origin $(0,0)$. Let $w = W'/2$. We have the following inequalities to specify the above information.

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

The first two inequalities specify that $x_0 = y_0$. The next four specify that Buf_1 is contained in R_A, the following four specify that Buf_2 is contained in R_B, and the last four specify that Buf_3 is contained in R_C. We call the inequalities (10)-(23) the *confinement* set. Note that all of the inequalities have the form $x_j - x_i \leq a_{ij}$. For this specific example, the problem is now transformed to a type of linear program as follow.

Problem 2 *Given constraints (10) - (23), any one from (2) - (5), and any one from (6) - (9), determine a set of values for x_1, y_1, x_2, y_2, x_3 and y_3 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 [13]. 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 11-(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 sixteen times. In general, in the worst case we need to run Bellman-Ford algorithm 4^k time, which leads to total time complexity $O(4^k(2m+2)(4m+k+2)) = (O(4^k m^2))$ (Assume $k = O(m)$). As we observe, m is not large. In addition, we can check beforehand to see if some of the inequalities in separation sets are impossible for a given rectangle pair. For example, in Figure 10, inequalities (2) and (5) are not possible.

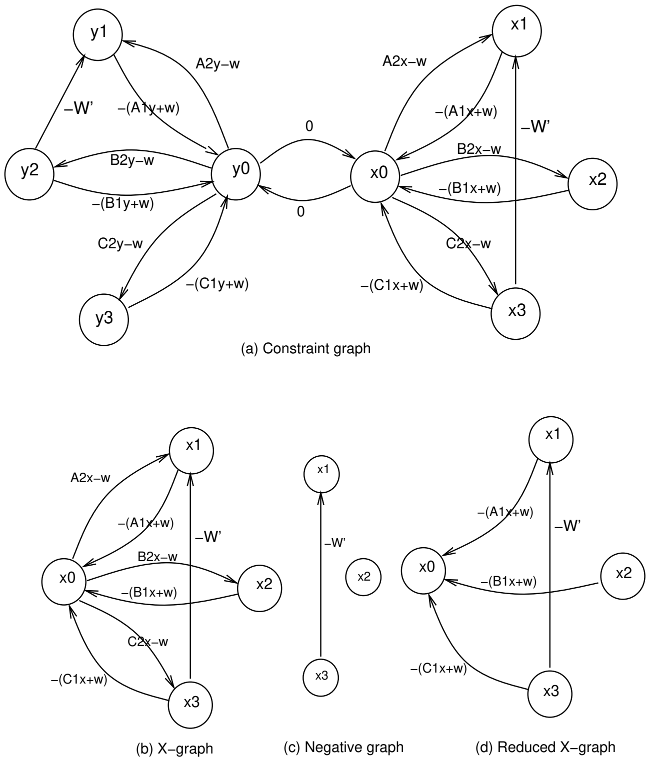

Figure 11: Examples of constraint graph, etc.

Theorem 3 below shows that the algorithm can be made more efficient. The theorem is preceded by a sequence of lemmas.

Lemma 1 *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 to all vertices shortest path problem on the two sub-graphs with source x_0 and y_0 respectively.*

Proof. From the characteristic of the inequality set, there are no edges from x_i to y_j or vice versa. Removing edges $x_0 \to y_0$ and $y_0 \to x_0$ disconnect the constraint graph. The shortest paths from x_0 to each x_i do not pass through any y_i, since any path from x_0 to x_i passing through some y_i's is not a simple path and therefore is not a shortest path. Similarly, The shortest paths from x_0 to each y_i do not pass through any x_i except x_0, but it must pass through y_0. Since the weight of the edge $x_0 \rightarrow y_0$ is 0, the weight of the shortest path from x_0 to any y_i is the same as the weight of the shortest path from y_0 to y_i. □

Now we only need to consider the two sub-graphs independently. We call them X-graph and Y-graph (Figure 11-(b)).

For simplicity, the following discussion is only in terms of X-graph. Nevertheless, it applies to Y-graph as well. This includes all of the following lemmas and Theorem 3. 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 $E(G_{nx}) = \{x_i \rightarrow x_j | i \neq j \text{ and } i, j \neq 0\}$. In other words, $E(G_{nx})$ corresponds to the set of edges from separation sets. We call G_{nx} *negative graph* (Figure 11-(c)), since all of its edges have negative weights.

Lemma 2 *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 2). Therefore, instead of running the Bellman-Ford algorithm to detect negative cycles, we can first use 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 associate an attribute $L(x_i)$ with each vertex in the negative graph. The meaning of $L(x_i)$ is the number of edges of the path consisting of the most edges from x_i to any other vertex. $L(x_i)$'s can also be obtained by depth-first search.

Lemma 3 *If the X-graph G_x has no negative cycle, and G_{nx} is the the corresponding negative graph, then the shortest paths from x_0 to every x_i have at most $D+1$ edges, where $D = \max_{x_i \in V(G_{nx})}\{L(x_i)\}$.*

Proof. By assumption, the negative graph has no negative cycles. The shortest path from x_0 to any x_i must have all of its edges in $E(G_{nx})$ except the first edge, which is an outgoing edge of x_0. Therefore, the shortest path has at most $D + 1$ edges. □

We remove all outgoing edges of x_0 from the X-graph and it becomes *reduced X-graph* (Figure 11-(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 3: Modified-Bellman-Ford (MBF)(G'_x, D)
Input: A reduced X-graph (Y-graph), and D (defined in Lemma 3);
Output: The shortst path length $d(v)$ for each $v \in V(G'_x)$;
or "FALSE" if negative cycle detected

```
Begin
    d(x_0) ← 0 and for every v ≠ x_0, d(v) ← W(x_0 → v).
    for i ← 1 to D
        do for each edge u -e-> v ∈ E(G'_x) do begin
            if d(v) > d(u) + w(e) then
                d(v) ← d(u) + w(e)
            if v = x_0 and d(v) < 0
                return "FALSE"
        end
    for each edge u -e-> v ∈ E(G'_x)
        do if d(v) > d(u) + w(e)
            then return "FALSE"
    return "TRUE"
End.
```

Compared to original algorithm, we modify the initialization step such that $d(x_0) \leftarrow 0$ as usual, but $d(x_i)(i \neq 0)$ is initialized to $W(x_0 \rightarrow x_i)$ instead of ∞. We also change the number of iterations of the outer loop to D instead of $|V|$ -1, where $|V|$ is the number of vertices.

Theorem 3 *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 the corresponding negative graph is cycle free.*

Proof. In the original Bellman-Ford algorithm (BF algorithm for short), at the completion of the k-th ($1 \leq k \leq |V(G)| - 1$) iteration of the first nested loop (we denote it as *Main Loop*) $d(x_i)$'s are the sums of weights of all path from x_0 to x_i consisting of k edge. The way we initialize $d(x_i)$ is equivalent to running the original BF algorithm on the X-graph for the first iteration of the Main Loop.

Next we show that we can discard all outgoing edges of x_0 from the X-graph without affecting the solution. If the test (**if** $d(v) > d(u) + w(e)$) succeeds and $v = x_0$, it must be true that $d(x_0) < 0$, since $d(x_0)$ is initialized to 0, and by assumption there is no negative cycle in the corresponding negative graph. It implies that the X-graph has a negative cycle going through x_0. Therefore the algorithm can terminate when $d(x_0) < 0$, so all outgoing edges of x_0 do not need to be tested.

If there are no negative cycles in the X-graph, then by lemma 3, we only have to run the Main Loop of the original BF algorithm $D + 1$ times. Equivalently, we only have to run the Main Loop of the Modified BF algorithm D times. □

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 4: Buffer-Placement (M)
Input: A list of merging blocks M. Each merging block overlaps with
    at least one other merging block in M.
Output: "False" if not placeable. Otherwise
    a list of buffer center coordinates (x_i, y_i)'s
Begin
  $Q \leftarrow \emptyset$;
  Construct Overlap Graph G_o;
  for each connected sub-graph of G_o **do begin**
    Rotate the corresponding merging blocks by 45°;
    Make all inequalities;
    Construct G_{cx} and G_{cy};
    for each combination of inequalities from the separation sets **do begin**
      Construct negative graphs G_{nx} and G_{ny};
      Depth-first search G_{nx} and G_{ny};
      if both G_{nx} and G_{ny} have no cycles **then begin**
          $D_x \leftarrow \max_{x_i \in G_{nx}}(L(x_i)$
          $D_y \leftarrow \max_{x_i \in G_{nx}}(L(x_i)$
          Add the edges of G_{nx} to G_{cx} and form G'_x;
          call MBF(G'_x, D_x);
          if MBF return "TRUE" **then begin**
            Add the edges of G_{ny} to G_{cy} and form G'_y;
            call MBF(G'_y, D_y);
            if MBF return "TRUE" **then begin**
              Rotate each $(d(x_i), d(y_i))$ by $-45°$ and
                  put into list Q;
              return Q;
            end;
          end;
      end;
    end;
  end;
End.

The use of depth-first search also saves running time. Since a negative graphs and its immediate predecessor differ only by one edge, we can detect cycles in constant time. For example, changing the direction of a forward edge must form a cycle, and adding a back edge must also form a cycle.

To summarize the discussion of the bottom-up phase, we state the time complexity of of Algorithm 2 in Theorem 4.

Theorem 4 *The time complexity of Algorithm 2 is $O(N^3 \log N + 4^k m^2 N^2)$, where k and m are the upper bounds of the number of edges and number of vertices of an overlap graph.*

Proof According to Theorem 1, there are totally $O(N \log N)$ merge operations. Each merge operation is associated with a finding minimum delay operation, which takes $O(N^2)$ time. Therefore, all merge operations take $O(N^3 \log N)$ time. According to Theorem 2, there are $O(N \log N)$ buffer insertion operations. Each takes constant time. It is clear that the time of merging operations dominates the time of buffer insertion. As is discussed in Section 3.4, the Buffer-Placement Algorithm runs in time $O(4^k m^2)$. Each recursive call of Bottom-Up-Clustering calls Buffer-Placement $O(N)$ times, so in total Buffer-Placement is called $O(N^2)$ times. The overall algorithm has time complexity $O(N^3 \log N + 4^k m^2 N^2)$. □

3.5 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 user selects the desired tree based on its (delay, wirelength) value, the top-down phase proceeds. We state it as follows.

Algorithm 5: 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 begin
      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 begin**
        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 (v_1);
        call Top-Down (v_2);
      end;
      else begin
        Choose any $pl(v) \in mb(v) \cap trr_p$;
        Find the original un-buffered 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 $pl(v) \in mb(v_0) \cap trr_v$;
        Let v_1 and v_2 be the two children of v_0;
        call Top-Down (v_1);
        call Top-Down (v_2);
      end;
    end;
End.

When a unbuffered node is visited, the behavior of Algorithm 3 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 merge segments. When a buffered node is visited, a wire with length $l(v_0)$ (can be 0) is inserted between v and v_0. Therefore we need to embed two nodes v and v_0, and buffer is placed at v. The time complexity of Algorithm 5 is $O(N)$ (N is the number of clock pins).

4 Experimental Results

Our algorithm was implemented in C++ on the Sun SPARC-5 workstation. We ran experiments on benchmark data r1-r5 [16]. The parameters of clock buffers are those used in [14]. 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 $\times$ 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 12-16. 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, our algorithm still performs better when more buffers are inserted.

In order to show that the delay and wirelength values of the clock trees constructed by our algorithm are actually good, in the figures we also show the positions of the results generated by one of the previous best algorithm [2] (labeled "Early"). Although doing buffer insertion causes wirelength to increase, our algorithm still manages to get very good wirelength values.

Acknowledgement

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. The authors are also grateful to M. Edahrio and A. Tsao of UCLA for valuable discussions.

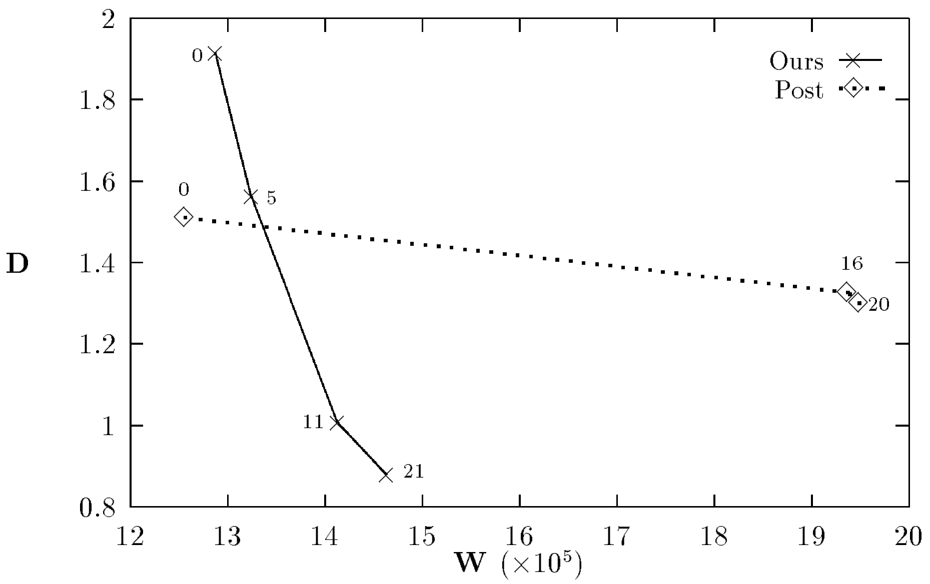

Figure 12: r1 (267 pins)

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] M. Edahiro, "An efficient zero-skew routing algorithm", *Proc. ACM/IEEE Design Automation Conference*, pp.375-380, 1994.

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

[11] 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.

[12] 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.

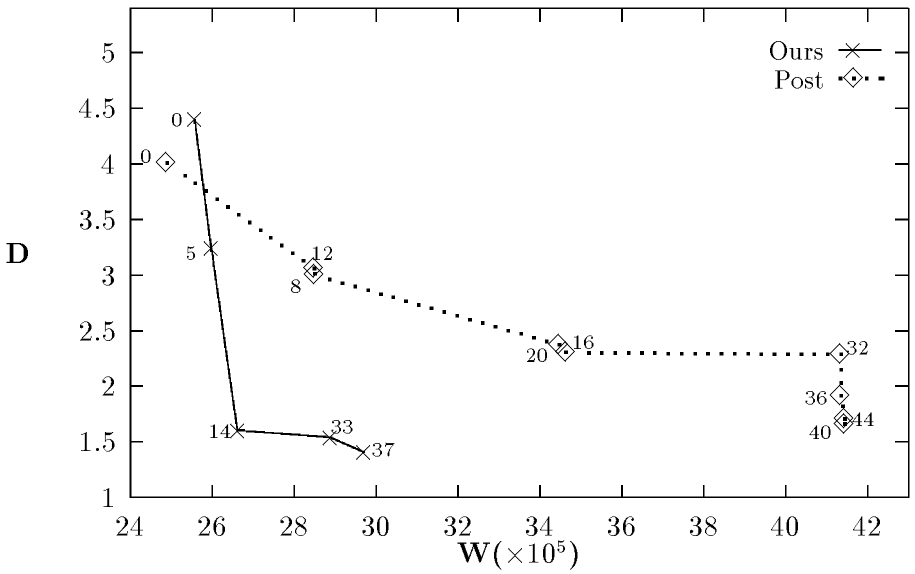

Figure 13: r2 (598 pins)

[13] E.L. Lawler, "Combinatorial Optimization: Networks and Matroids," Holt, Rinehart & Winston, New York, 1976.

[14] 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.

[15] 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.

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

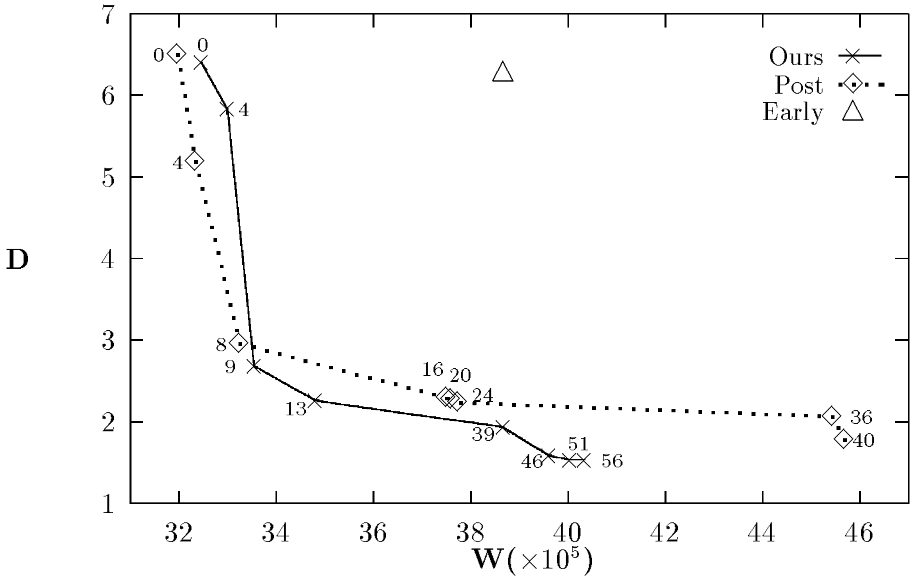

Figure 14: r3 (862 pins)

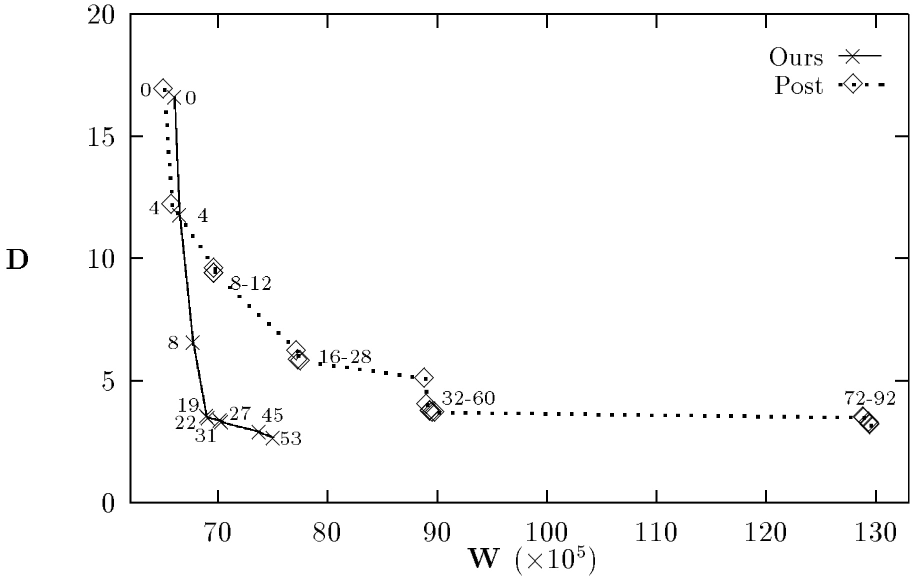

Figure 15: r4 (1903 pins)

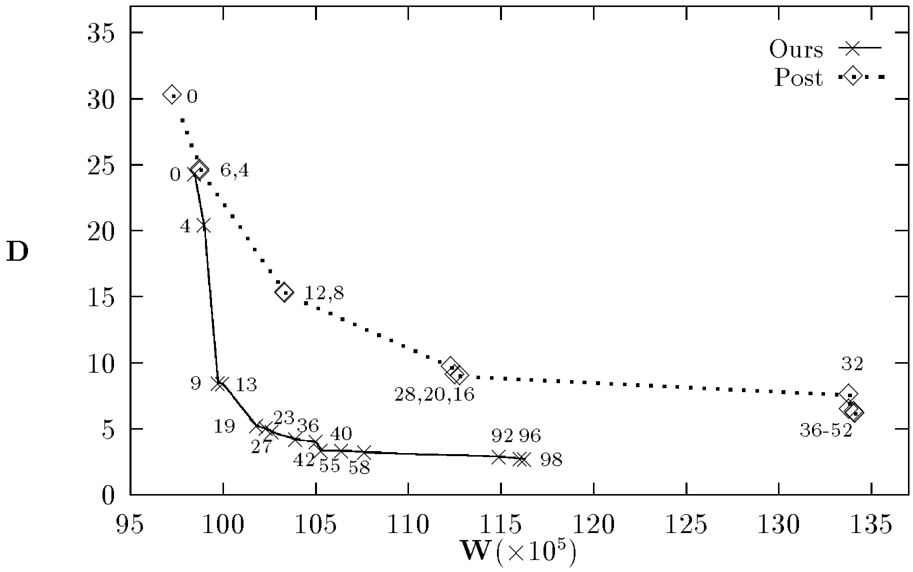

Figure 16: r5 (3101 pins)