Fast Timing-Model Independent Buffered Clock-Tree Synthesis

Xin-Wei Shih and Yao-Wen Chang, *Senior Member, IEEE*

***Abstract*—In high-performance synchronous chip design, a buffered clock tree with small clock skew is essential for improving clocking speed. Due to the insufficient accuracy of timing models for modern chip design, embedding simulation into a clock-tree synthesis flow becomes inevitable. Consequently, the running time for clock-tree synthesis becomes prohibitively huge as the complexity of chip designs grows rapidly. To construct a buffered clock tree efficiently, we propose an efficient timing-model independent approach to perform skew minimization by structural optimization. To achieve the goal, a novel clock-tree structure, called *symmetrical structure*, is presented. At each level of a symmetrical clock tree, the number of branches, the wirelength, and the inserted buffers are almost the same. It is natural that the clock skew could be minimized if the configurations of all paths from the clock source to sinks are similar. By symmetrically constructing a clock tree, the clock skew can be minimized without referring to simulation information. Experimental results show that our approach can not only efficiently construct a buffered clock tree but also effectively minimize clock skew with marginal wiring overheads. Based on a set of commonly used IBM benchmarks, e.g., a state-of-the-art work without (with) ngspice simulation results in averagely 10.04X (3.44X) clock skew and requires 163X (61906X) running time over our approach.**

***Index Terms*—Clock skew, clock-tree synthesis, physical design, timing.**

I. Introduction

SKEW-MINIMIZED buffered clock-tree synthesis (BCTS) plays an important role in high-performance very large-scale integration (VLSI) designs for synchronous circuits. Due to the insufficient accuracy of existing timing models for modern chip design, embedding simulation process into a clock-tree synthesis flow becomes inevitable [9]. Consequently, the running time for clock-tree synthesis becomes prohibitively huge as the complexity of chip designs grows rapidly. Therefore, it is desirable to develop an efficient mechanism for the synthesis of large-scale clock trees.

A well-designed clock tree improves clocking speed by minimizing clock skew. The clock skew is the difference between the minimum and the maximum arrival times of the clock signal. Earlier clock-tree synthesis works performed skew minimization based on simple timing models [8], [11], [12], [25], such as the linear delay model or the Elmore delay model [10]. Later, the zero-skew concept [25] was extended and dynamic programming was adopted to reduce wirelength in the deferred-merge embedding (DME) algorithm [1], [2], [7]. On the other hand, for modern chip designs, buffers are needed essentially in a clock tree for slew-rate optimization. The slew rate is the maximum changing rate of signals in a circuit. As a result, buffer insertion techniques were applied to optimize both slew rates and signal-phase latency of clock trees [3]–[5], [24].

In most previous works, the accuracy of timing estimation highly depends on timing models. However, although timing models could easily estimate latency-growing trends, they are difficult to analyze skew accurately. Therefore, the works [13]–[16], [21] embedded SPICE simulation in the buffered clock-synthesis flow. Based on the simulation information, these works adjusted the clock network and the skew could be accordingly reduced. However, the speed of involving simulation is quite slow as shown in their experimental results.

A possible way to improve the speed is performing the clock construction by structural optimization. The previously proposed *meshes* structure used in some work, such as [20] and [26], could naturally have small clock skew because of its uniform topology and balanced structure. Compared with the tree structure, nevertheless, the mesh structure requires prohibitively high resource that may not be desired in VLSI designs. Another tree-construction method based on structural optimization, called type matching [6], was proposed to minimize skew by matching the gate type. That is, if gates at the same tree level are of the same type with similar latencies, the clock skew could be optimized. However, the optimization is limited by only performing type matching without considering wiring structures. In our observation, the skew could be further improved by aligning the wiring structure.

In this paper, a novel timing-model independent BCTS method is proposed. We perform the skew minimization based on a new structure that is called symmetrical structure, in which the configurations including buffering and wiring structures of all paths from the clock source to its sinks are

Manuscript received October 18, 2011; revised January 12, 2012; accepted February 24, 2012. Date of current version August 22, 2012. This work was supported in part by the MediaTek Fellowship, SpringSoft, TSMC, and NSC of Taiwan, under Grants NSC 100-2221-E-002-088-MY3, NSC 99-2221-E-002-207-MY3, NSC 99-2221-E-002-210-MY3, and NSC 98-2221-E-002-119-MY3. Preliminary version of this paper was presented at the 2010 ACM/IEEE Design Automation Conference, Anaheim, CA, June 2010 [22]. This paper was recommended by Associate Editor P. Saxena.

X.-W. Shih is with the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan (e-mail: raistlin@eda.ee.ntu.edu.tw).

Y.-W. Chang is with the Department of Electrical Engineering and the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan (e-mail: ywchang@cc.ee.ntu.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2012.2191554

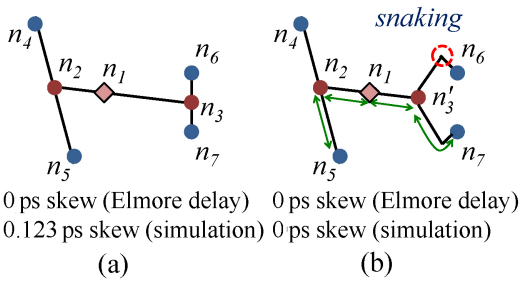

Fig. 1. Example of the difference between symmetrical and unsymmetrical clock trees, where n_1 is the tree root, n_4, n_5, n_6, and n_7 are sinks, and other nodes are tree internal nodes. The node coordinates are listed as follows: $n_1(12, 26)$, $n_2(3, 29)$, $n_3(33, 19)$, $n_4(1, 57)$, $n_5(5, 1)$, $n_6(33, 29)$, $n_7(33, 9)$, and $n_3'(21, 23)$, where the unit is μm. (a) Unsymmetrical clock tree, constructed based on the Elmore delay model, yields zero skew under the timing model, but 0.123 ps skew evaluated by SPICE simulation due to the mismatch of wirelength. (b) Symmetrical clock tree yields zero skew for both the Elmore delay and simulation, since the moving of n_3 to n_3' and the snaking result in identical wirelength at the same tree level.

almost the same. At each level of a clock tree, if the number of branches and wirelength are identical, the latencies from the clock source to all different sinks would be similar. Hence, we can perform timing-model independent skew minimization via fast constructing a symmetrical structure without referring to simulation information. Moreover, this symmetrical structure also simplifies buffer-insertion operations. By aligning buffer distribution on paths from the source to sinks, the delays of buffers at the same level are also similar because of identical buffer-size, driving load, and wiring configuration.

Fig. 1 gives an example to illustrate the difference between symmetrical and unsymmetrical clock trees. Here, n_1 is the tree root, n_4, n_5, n_6, and n_7 are sinks, and other nodes are tapping points (internal nodes). The clock tree in Fig. 1(a) is constructed by using the Elmore delay model and yields zero skew under the timing model. However, the two subtrees of n_1 have different wiring structures, and thus the clock skew derived by simulation is nonzero (0.123 ps with SPICE simulation). In contrast, the symmetrical clock tree in Fig. 1(b), which is constructed by moving n_3 to n_3' and snaking wires of connections (n_3, n_6) and (n_3, n_7), generates zero skew for both the Elmore delay model and simulation since every tree level has their own uniform wirelength.

In this paper, we present a three-stage flow to construct a buffered symmetrical clock tree. First, the branch number for each level of the clock tree is derived. Second, according to the derived branch number, the tree is constructed level by level from leaves to the root by giving each level a certain uniform length. Finally, as the identical number of branches and identical wirelength at each level are established, buffers are inserted symmetrically by tracing along tree edges and aligning buffer distribution.

Our approach for the symmetrical buffered clock-tree construction is timing-model independent, since the skew minimization is achieved by structural optimization. As a result, the tree construction can be performed very efficiently and is particularly suitable for large-scale clock-tree synthesis, since no time-consuming simulation process is required. Furthermore, most of the previous works are based on the binary-tree topology, such as [1], [2], [7], [8], [11]–[16], [21], [25], and [26]; in contrast, our symmetrical clock tree can be of multiple branches, implying a bigger solution space with more desirable clock trees. Experimental results show the superior effectiveness and efficiency of our approach for clock-tree construction and clock-skew minimization. Based on a set of commonly used IBM benchmarks, e.g., a state-of-the-art work without (with) ngspice [18], simulation results in averagely 10.04X (3.44X) clock skew and requires 163X (61906X) running time over our approach.

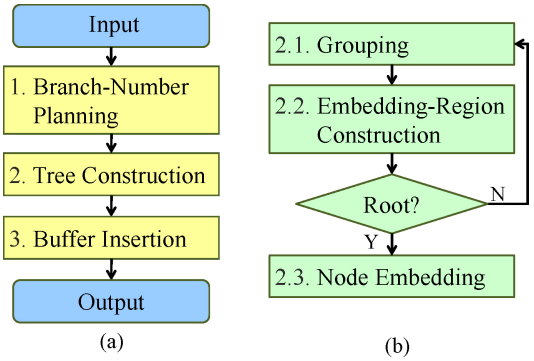

Fig. 2. (a) Overall flow for solving the BCTS problem. (b) Flow of the tree construction.

The remainder of this paper is organized as follows. The problem is formulated in Section II. Section III presents our proposed approach. Section IV extends the problem to consider blockages. Section V reports the experimental results and Section VI concludes our paper.

II. Problem Formulation

We define the BCTS problem as follows.

1) *Problem: BCTS.*
Instance: Given a set of clock sinks, a slew-rate constraint, and a library of buffers.
Question: Construct a buffered clock tree to minimize its skew, subject to no slew-rate violation.

Unlike most previous clock-tree synthesizers that are based on some timing model (e.g., the Elmore delay model [10]), we apply SPICE simulation (e.g., ngspice [18]), which is much more accurate than the Elmore delay model, to measure the resulting skew.

III. Symmetrical Clock-Tree Synthesis

Fig. 2 illustrates our new flow for constructing a buffered symmetrical clock tree, in which: 1) the number of branches; 2) wirelength; and 3) buffers are identical at each tree level.

To obtain the identical number of branches, we first perform branch-number planning to assign specific branch numbers to each tree level. This planning stage restricts the subtrees at the same level to be connected with an identical number of branches. Then, the identical wirelength requirement is established in the second stage, tree construction. The tree construction groups subtrees level by level from leaves to root. At

each tree level, if some subtrees need longer wires to connect together, others must be lengthened by snaking to get a uniform wirelength. Therefore, the resulting clock tree could have identical length of signal wires at the same level. However, the snaking may consume additional resource usage, so we route the snaked wires toward the positions of their parent nodes to utilize the additional resource effectively, and accordingly the wirelength of higher tree level can be shortened.

The detailed flow of tree construction is illustrated in Fig. 2(b). First, the grouping is performed to group subtrees into desired clusters. As different grouping configurations would lead to different uniform wirelength for connection, the resulting clusters should have minimized uniform wirelength to avoid too much additional resource. Then we apply a common connection length to each cluster and perform the embedding-region construction to locate the potential embedding positions to which snaked wires can reach. We repeat these two stages till the embedding region of the tree root is built. Finally, the node embedding finds the exact physical locations for tree nodes and routes wires in a top–down manner. This embedding strategy, which is useful for snaked-wire utilization and wirelength minimization, is inspired by the DME algorithm [1], [2], [7].

As long as the symmetrical tree topology is created and embedded, the buffer insertion can be determined straightforwardly. We trace along the tree edges in a top–down manner and insert the identical type and size of buffers at the positions with an identical distance from the clock source. Consequently, the delays of buffers can be similar, even if we do not introduce timing models or refer to simulation for timing analysis. We detail the three stages in the following subsections.

A. Branch-Number Planning

Before constructing the clock tree for a given BCTS instance, the branch numbers of each tree level are planned first. A branch-number plan (BNP) is a list that indicates the branch numbers from the root to leaves. Since the total branch number of some tree level is equal to the number of preceding level times the corresponding number of branches, the number of leaves (sinks) can be treated as a multiplication sequence of branching. This multiplication sequence exactly forms a *factorization*.

According to the observation, we factorize the number of sinks into prime numbers in the nondecreasing order based on the traditional mathematical procedure to construct a BNP. Equation (1) gives the factorization as follows:

$$n = f_1 \times f_2 \times \ldots \times f_q \tag{1}$$

where n is the number of sinks, f_i is the ith prime number, q is the total number of prime numbers, and $f_i \leq f_{i+1}, \forall i < q$.

Then, the BNP is arranged in the nonincreasing order[1] of the factorization list, shown as follows:

$$B(n) =< b_1, b_2, \ldots, b_q >=< f_q, f_{q-1}, \ldots, f_1 > \tag{2}$$

[1]Any permutation of the factorization list is fine to form a BNP. In our implementation, we choose the nonincreasing order.

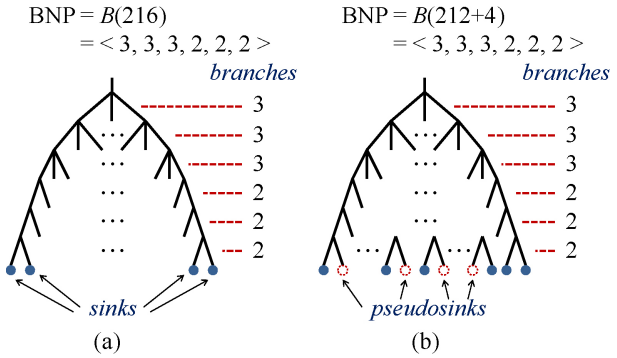

Fig. 3. Examples for the branch-number planning. (a) BNP for 216 sinks. (b) BNP for 212 sinks.

where $B(n)$ is the BNP for n sinks and b_i is the branch number for ith tree level. Note that the total number of tree levels is equal to q, i.e., the number of prime numbers. See Fig. 3(a) for an example; 216 (sinks) are factorized into $2\times2\times2\times3\times3\times3$, so we assign $< 3, 3, 3, 2, 2, 2 >$ to the BNP. It means that branches are all three from the first to third levels and are all two from the fourth to sixth levels.

Nevertheless, the factorization may result in a big branch number, implying a large number of fanout nodes that cannot be driven by a tree node. As shown in Fig. 3(b), for instance, the number 212 (sinks) is initially factorized into $2 \times 2 \times 53$. If the maximum allowable branching is 13, the 53 branching is invalid. To handle this problem, we add pseudosinks to increase the total number of nodes, so that all branch numbers derived by the factorization can be less than or equal to the maximum allowable branching. The pseudosinks should be transformed into dangling wires to maintain the symmetry in the later stages. For skew minimization, we add pseudosinks only at the lowest level, i.e., the leaves.

Fig. 4 compares the results between different structures, in which pseudosinks are added at different levels. Here, we assume that the maximum allowable branching is three and each connection of tree edges uses a unit wirelength of 50 nm. Two symmetrical trees for ten sinks are constructed following the same BNP $< 3, 2, 2 >$: Fig. 4(a) adds one pseudosink at a higher level, whereas Fig. 4(b) adds two pseudosinks at the lowest level. By adding pseudosinks at higher levels, the structure in Fig. 4(a) reduces the inserted pseudosink number and the routing resource usage; however, it incurs a larger skew with the ngspice simulation. We use a path-based analysis to explain the results. We consider two paths, the paths to sinks n_1 and n_4, for Fig. 4(a) and (b), which dominate the skew values for both trees. In Fig. 4(a) [Fig. 4(b)], the configurations of the two paths are different at the second (third) branch. Since the total downstream-loading difference of the second branch in Fig. 4(a) is larger than that of the third branch in Fig. 4(b), the tree in Fig. 4(a) incurs a larger skew.

According to the aforementioned analysis, we add pseudosinks at the lowest level. The adding process increases the total sink number one by one until all branchings are valid. As a result, the case in Fig. 3(b) adds four pseudosinks and the resulting 216 sinks can be factorized with practical branch numbers, as shown in Fig. 3(a). Note that the maximum

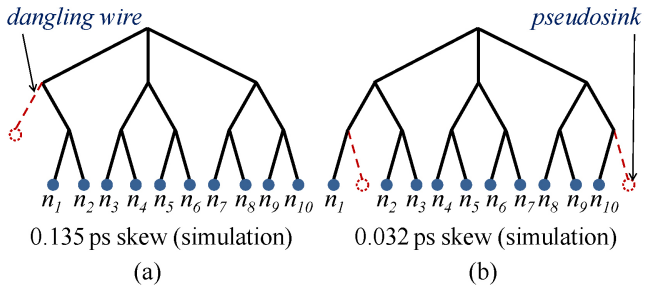

Fig. 4. Example of pseudosink insertion. (a) One pseudosink is inserted at a higher level. (b) Two pseudosinks are inserted at the lowest level, resulting in a smaller skew.

Procedure: *BranchNumberPlanning*$(n, \hat{b}, \vec{b}, p)$
Input: n /*The sink number*/
$\hat{b}$ /*The maximum branching based on the buffer library*/
Output: $\vec{b}$ /*Resulting branch number plan*/
p /*Resulting pseudosink number*/

1 $p \leftarrow 0$;
2 **while** true
3 $\vec{b} \leftarrow B(n+p)$;
4 **if** $b_i \leq \hat{b}, \forall i$
5 **then return** $\vec{b}$ and p;
6 **else** $p \leftarrow p+1$.

Fig. 5. Procedure of branch-number planning.

branching depends on the driving capability of buffers in the given buffer library.

We summarize the branch-number-planning procedure in Fig. 5. Based on the given sink number and maximum branching, the procedure gives the valid BNP and the corresponding number of pseudosinks. At first, line 1 initializes the pseudosink number to 0. Then, lines 2–6 keep factorizing the number of leaves, i.e. $n + p$, and adding pseudosinks until a valid BNP is found.

B. Tree Construction

In this section, the tree construction based on the derived BNP from the preceding stage is detailed. We first introduce the data structure, the tilted rectangular region (TRR) [1], to represent potential embedding positions for tree nodes, in Section III-B1. Then the three main stages of tree construction, grouping, embedding-region construction, and node embedding, are presented in Sections III-B2–III-B4, respectively. Finally, the cases with nonzero pseudosinks are described in Section III-B5.

1) *TRR:* A TRR is introduced to represent potential embedding positions, called embedding region, for a tree node. A TRR is a 45-degree or 135-degree rectangular region that consists of a *core* and a *radius*, as shown in Fig. 6(a). The core is a 45-degree or 135-degree line segment, and the radius denotes the Manhattan distance from the core to the region boundaries.

To perform snaking for wire extension, we define an extending operation, which enlarges the region by pushing the four

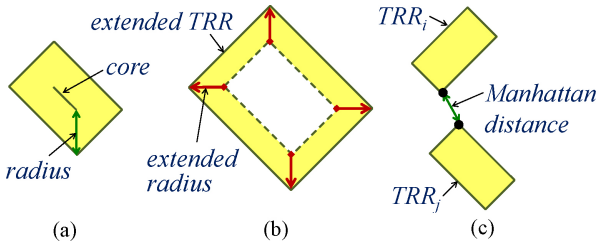

Fig. 6. (a) Configuration of TRR. (b) TRR extension. (c) Distance between two TRRs, TRR_i and TRR_j, is defined as the Manhattan distance between two closest points from the two TRRs.

end points with an extending radius. The extending operation is illustrated in Fig. 6(b), where the arrow lines indicate the pushing. The distance between two TRRs is defined as the Manhattan distance between two closest points from the two TRRs. See Fig. 6(c) for an illustration.

2) *Grouping:* After the BNP is constructed, we group subtrees into desired clusters based on the BNP. The grouping is a key to resource-usage minimization. As aforementioned, a uniform wirelength for the connection at a certain tree level is realized by lengthening the connections of closer subtrees. Obviously, the maximum distance among subtrees within the same cluster is the bound of the uniform wirelength. If this maximum distance could be shorter, the uniform wirelength is also reduced accordingly. We refer to the diameter of a cluster as the maximum distance among subtrees in the cluster. Hence, the objective of grouping is to minimize the maximum diameter within clusters. Note that we use a TRR to represent the potential embedding positions of a tree node, so the distance is determined based on the distance of TRRs.

There are two typical schemes, clustering [8] and partitioning [11], for grouping subtrees into desired clusters. Their frameworks are quite different. Clustering iteratively extracts a cluster at a time until no subtree remains, while partitioning recursively divides subtrees into sets until the set size is the same as the target cluster size. In general, to minimize the maximum cluster diameter, clustering get some advantage when subtree distribution is locally concentrated; in contrast, partitioning has the advantage when the subtrees are evenly distributed in the chip.

However, the sink or subtree distributions might not be the same in different chips. Therefore, we propose a hybrid approach, called hybrid grouping, to integrate the two schemes to consider different distributions. The hybrid approach first estimates the subtree distributions and then adopts either the clustering scheme or the partitioning scheme to group subtrees. In this section, the proposed clustering and partitioning techniques, respectively, are presented in Sections III-B2a and III-B2b, while the proposed hybrid solution is presented in Section III-B2c.

a) *Clustering:* For given branch number, the clustering extracts a cluster at a time until no subtree is remained. To design the clustering approach, we observe that extracting clusters from the periphery of subtree distribution can lead to shorter maximum cluster diameter. See Fig. 7 for an example; the hollow regions and the solid regions, respectively, represent the distributions of extracted subtrees and remained subtrees at

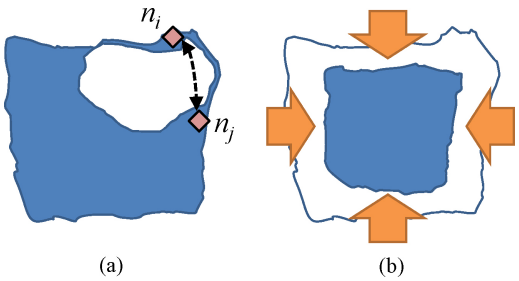

Fig. 7. (a) Scenario that subtrees are not extracted from the periphery. n_i and n_j represent subtrees. (b) Clustering performs the extraction from the periphery.

a certain iteration. In Fig. 7(a), subtrees are not extracted from the periphery. Since the cluster size shall be exactly equal to the branch number for symmetry, subtree n_i at periphery may have no choice but to be clustered with subtree n_j by crossing hollow regions, i.e., the distributions of extracted subtrees. Obviously, once a cluster crosses other clusters, the resulting cluster diameter would be longer than the scenario that avoids crossing. Therefore, our clustering perform the extraction from the periphery, as illustrated in Fig. 7(b).

Our clustering consists of three steps: 1) picking one of subtrees from bottommost, rightmost, topmost, and leftmost as a pivot; 2) growing a cluster from the pivot; and 3) extracting the cluster. To minimize the diameter, the growing augments the cluster by a subtree at a time, where the augmented cluster has minimum diameter within the clusters containing the original cluster. The three steps are repeated iteratively for bottommost, rightmost, topmost, and leftmost pivots by turns until no cluster is remained.

Fig. 8 gives a clustering example for the lowest level of a clock tree. In the figure, the circles indicate extracted clusters, the numbers beside the circles indicate the extraction order, and arrows point to the pivots. There are 18 sinks, as shown in Fig. 8(a), and the target branch is two. Fig. 8(b) illustrates the iterations one to four. At the iteration one, the bottommost subtree is picked as a pivot; then a desired cluster is grown and, finally, the grown cluster is excluded for the following iterations. Based on the same steps, the clusters for rightmost, topmost, and leftmost pivots, respectively, are also extracted at iterations two, three, and four. Fig. 8(c) and (d) demonstrates the following five iterations. The resulting clusters are shown in Fig. 8(e).

The clustering procedure is summarized in Fig. 9. Since each cluster size shall be exactly equal to the branch number b, the final extracted cluster number is $|S|/b$. Thus, the extraction is performed $|S|/b$ times, as lines 1 and 2. Lines 3–12 determine the pivot subtree for growing and put the pivot into the subtree set of extraction X. Then, X is augmented $(b-1)$ times at lines 13–19. To minimize cluster diameter, lines 15–17 pick the subtree that induces the minimum diameter for augmentation, where function $D(Y)$ returns the diameter of subtree set Y. After the size of X reaches b, line 20 takes X into the clustering solution.

To efficiently perform clustering, we propose an incremental-update method for diameter calculation, i.e., function $D(X \cup \{s\})$ in line 16 of Fig. 9. Without loss of

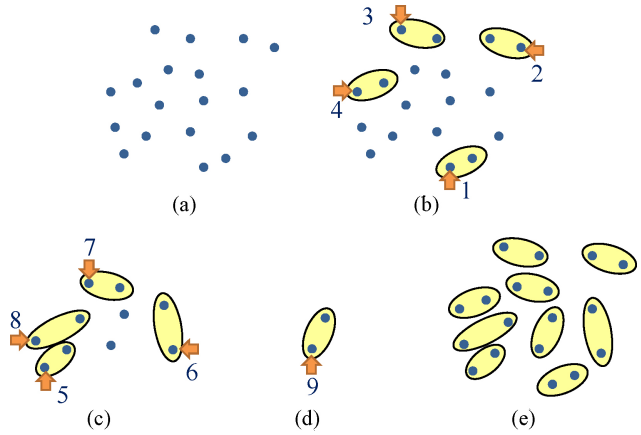

Fig. 8. Example of the clustering. (a) Given 18 sinks and the branch number two. (b) Iterations 1–4 and corresponding four clusters. (c) Iterations 5–8 and corresponding four clusters. (d) Iteration 9 and corresponding cluster. (e) Resulting nine clusters.

generality, a partial subtree set of extraction X^i can be represented as $\{x_1, x_2, \ldots, x_i\}$, where x_i is the ith picked subtree. Accordingly, for the iteration with index j in lines 13–18 in Fig. 9, X equals X^j. As shown in (3), $D(X \cup \{s\})$ can be formulated as $D(X^j \cup \{s\})$. Equation (3) also derives a recursion in two terms for the incremental update as follows:

$$\begin{aligned} D(X \cup \{s\}) &= D(X^j \cup \{s\}) \\ &= \max\{d_{x_1,s}, d_{x_2,s}, \ldots, d_{x_j,s}, D(X^j)\} \\ &= \max\{D(X^{j-1} \cup \{s\}), d_{x_j,s}, D(X^j)\} \\ &= \max\{D(X^{j-1} \cup \{s\}), d_{x_j,s}\} \end{aligned} \tag{3}$$

where $d_{x_i,s}$ is the distance between the subtree x_i in X^j and a candidate subtree s for x_{j+1}. To calculate $D(X^j \cup \{s\})$, one can compute the distances from s to all subtrees in X^j one by one, and then take the maximum value between the distances and $D(X^j)$, namely, $\max\{d_{x_1,s}, d_{x_2,s}, \ldots, d_{x_j,s}, D(X^j)\}$. However, since the diameters with size j have been calculated in the previous iteration, we reuse $D(X^{j-1} \cup \{s\})$. Moreover, as s is not picked as the jth subtree in X^j, we have $D(X^{j-1} \cup \{s\}) \geq D(X^j)$, which simplifies the calculation into only two terms by eliminating $D(X^j)$. Therefore, based on (3), we can calculate the diameters by performing the incremental update in constant time. Note that we define $X^0 = \emptyset$ and $D(\{s\}) = 0$ to complete the usage of (3) for the procedure in Fig. 9.

b) *Partitioning:* For a given BNP [see (2)], we recursively divide the subtrees in a top–down manner. Specifically, for the ith tree level, first we divide subtrees into b_1 clusters of an identical size. Then each of the b_1 clusters is further divided into b_2 smaller clusters, resulting in $b_1 \times b_2$ clusters. The same procedure is repeated recursively until $b_1 \times b_2 \times \ldots \times b_{i-1}$ clusters are derived. Note that we try to minimize the cluster diameter in every recursion. In this way, the partitioning can typically obtain a desired cluster diameter since the global subtree distribution is considered throughout the whole process.

```
Procedure: Clustering(S, b, C)
Input:     S /*The set of subtrees*/
           b /*The branch number*/
Output:    C /*Resulting cluster set*/
1   m ← |S|/b
2   for i ← 1 to m
3       do X ← ∅
4          if i mod 4 = 1
5             then X ← X ∪ {bottommost subtree in S}
6          else if i mod 4 = 2
7                  then X ← X ∪ {rightmost subtree in S}
8          else if i mod 4 = 3
9                  then X ← X ∪ {topmost subtree in S}
10         else if i mod 4 = 0
11                 then X ← X ∪ {leftmost subtree in S}
12         S ← S \ X
13         for j ← 1 to (b − 1)
14            do t ← first subtree in S
15               for each subtree s ∈ S
16                  do if D(X ∪ {s}) ≤ D(X ∪ {t})
17                        then t ← s
18               S ← S \ {t}
19               X ← X ∪ {t}
20         C ← C ∪ {make cluster for X}
21  return
```

Fig. 9. Procedure of clustering.

For the dividing, we borrow the idea of cake cutting, i.e., slicing a cake into pieces from the center of the cake. The benefit of the cake-cutting method is that the partitioning of multiple instances can be performed intuitively. To realize this idea, we work on the polar coordinate system. Mathematically, the polar coordinate system is defined on a fixed point, called origin, and a fixed direction, called polar axis. Based on the system, a point is defined by two terms: radius and polar angle. For a given point, the radius is the distance between the origin and the given point and the polar angle is the angular magnitude between the polar axis and the direction from the origin to the given point. Based on the definition, we first set the origin and the polar angle as the geometric center of the cluster and the x-axis, respectively. Then, we sort the polar angles of subtrees. Since subtrees are represented by TRRs, we use the center point of a TRR as the representative for sorting. By restricting the dividing on this sorted order, we apply dynamic programming to calculate their diameters efficiently, and thus find the best solution in this restricted solution space.

As restricted to the sorted order, if the ith and the jth subtrees of the sorted order are divided in the same cluster, then the $(i+1), (i+2), \ldots, (j-1)$th subtrees shall also be contained in the same cluster. Consequently, we have the recursive formulation in (4). In the formulation, $c[i, j]$ is defined as the diameter of the cluster that contains the $i, (i+1), \ldots, j$th subtrees as follows:

$$c[i, j] = \begin{cases} 0, & \text{if } i \geq j \\ \max\{c[i, j-1], c[i+1, j], d_{i,j}\}, & \text{if } i < j \end{cases} \quad (4)$$

where $d_{i,j}$ is the Manhattan distance between the ith and the jth subtrees of the sorted order. The underlying idea of the formulation is to reuse the calculated diameters of the subclusters. For $c[i, j]$, the only subtree pair not considered in the subclusters is the pair of the ith and the jth subtrees. Thus, $c[i, j]$ is dominated by either $c[i, j-1]$ and $c[i+1, j]$ of subclusters, or the distance between the ith and the jth subtrees. Based on the recurrence, we can perform dynamic programming to find the diameters for all clusters in the restricted solution space. Then, by scanning the clusters linearly, we can obtain the minimized maximum cluster diameter.

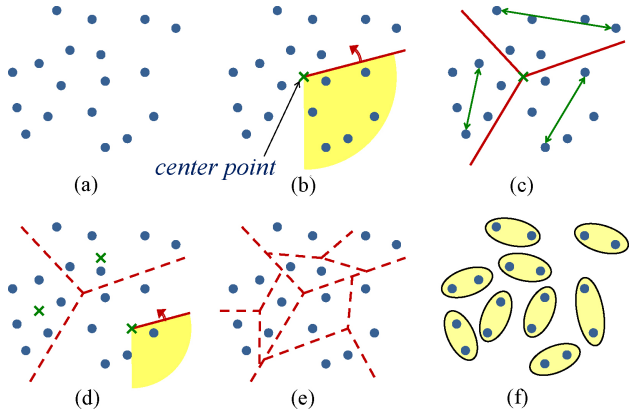

Fig. 10. Example of the partitioning. (a) Given 18 sinks, and the target BNP is $< 3, 3, 2 >$. (b) Geometric center and the sorted order of polar angles relative to the center are determined. (c) Three divisions resulting from the first dividing. (d) Same process is applied to the three divisions recursively. (e) Nine divisions resulting from the second dividing. (f) Resulting clusters for the third level of the clock tree (the top level is the root).

Based on the computed cluster diameters, we then find the cluster set, which results in the minimized maximum cluster diameter, in linear time. We first define the diameter of the ith cluster to be $c[i, i+z-1]$, where z is the target cluster size. Consider that the target number of clusters is b. As we exactly partition subtrees into b clusters, the number of candidate cluster sets is z. For the jth cluster set, the corresponding maximum cluster diameter is $\max\{c[j, j+z-1], c[j+z, j+2\cdot z-1], \ldots, c[j+(b-1)\cdot z, j+b\cdot z-1]\}$. Checking the z sets, we select the one with the minimized maximum cluster diameter. Note that the checking can be performed in linear time, i.e., $O(b \cdot z)$, where $b \cdot z$ is the total number of subtrees.

Fig. 10 gives a partitioning example for the lowest level of a clock tree. There are 18 sinks, as shown in Fig. 10(a), and the target BNP is $< 3, 3, 2 >$. In this example, the lowest level of the clock tree needs nine clusters of the identical size. At the first recursion, we find the geometric center of all sinks and determine the sorted order of polar angles, as illustrated in Fig. 10(b). Note that in this example, representatives of sinks are exactly the positions of themselves, since the TRRs of the sinks have zero length for both the core and the radius. Then, a dynamic programming technique is performed and the maximum cluster diameter among the three divisions are optimized. Fig. 10(c) illustrates the divisions of the three resulting clusters, where the arrow lines denote diameters of the clusters. The same process is applied to the three clusters recursively for the second recursion, as illustrated in Fig. 10(d). Finally, we have the resulting partition and corresponding nine clusters, as shown in Fig. 10(e) and (f), respectively.

```
Procedure: Partitioning(S, b⃗, k, g, C)
Input:     S /*The set of subtrees*/
           b⃗ /*The branch number planning*/
           k /*The current tree level*/
           g /*The target tree level*/
Output:    C /*Resulting cluster set*/
1   if k = g
2     then C ← {make cluster for S}
3     else
4       (x^c, y^c) ←geometric center point of nodes in S
5       a⃗ ←sort polar angles of nodes in S relative to (x^c, y^c)
6       c[1..|S|, 1..|S|] ←find diameter by Eq. (4) based on a⃗
7       s[1..b_k] ←divide S into b_k sub-sets based on c
8       for i ← 1 to b_k
9         do Partitioning(s[i], b⃗, k + 1, g, C')
10           C ← C ∪ C'
11  return
```

Fig. 11. Procedure of partitioning.

The partitioning procedure is summarized in Fig. 11. Lines 1 and 2 define the termination process of recursive dividing. If the termination condition does not meet, the dividing is performed recursively in lines 4–10. For the dividing, line 4 first determines the geometric center, (x^c, y^c), then line 5 sorts the polar angles of subtrees relative to (x^c, y^c). Line 6 performs dynamic programming to precalculate cluster diameters based on the sorted order. Since the target cluster size is S/b_k, only $c[i, j]$ with at most S/b_k subtrees between the ith and the jth subtrees is computed. Line 7 divides the input subtree set S into b_k clusters, where k is the current tree level for dividing. According to the dividing result $s[1..b_k]$, lines 8 and 9 further partition for each set in $s[1..b_k]$ recursively.

c) *Hybrid Grouping:* As aforementioned, the hybrid grouping estimates the distribution status of the subtrees and then adopts either clustering strategy or partitioning strategy. Typically, the clustering and the partitioning can lead to shorter maximum cluster diameter as subtree distributions are, respectively, locally concentrated and globally even. Thus, we propose a measurement to judge whether the distribution is globally, even so we can decide that strategy is appropriate. In the following, we present the proposed measurement in three steps: 1) defining a generalized measurement that characterizes the difference between the input values and the corresponding expected values; 2) specifying the subfunctions in the generalized measurement for identifying whether the vector is evenly distributed; and 3) utilizing the specified measurement to customize our measurement for proposed partitioning.

First, we define a generalized measurement function for any characteristic values of distribution as follows:

$$M(\vec{v}) = \frac{1}{R(\vec{v})} \cdot \frac{1}{|\vec{v}|} \sum_{i=1}^{|\vec{v}|} |v_i - E(\vec{v}, i)|. \tag{5}$$

In (5), $M(\vec{v})$ characterizes the difference between the sorted vector $\vec{v}$ and the expected vector of $\vec{v}$, where function $R(\vec{v})$ gives the value range of $\vec{v}$ and $E(\vec{v}, i)$ gives the expected ith value with respect to $\vec{v}$. There are two terms in (5): the first term $\frac{1}{R(\vec{v})}$ is the normalization factor and the second term $\frac{1}{|\vec{v}|}\sum_{i=1}^{|\vec{v}|} |v_i - E(\vec{v}, i)|$ is the average difference between the elements in $\vec{v}$ and their corresponding expected value. If the input $\vec{v}$ is very different from the expected values, the average difference would be much larger; thus, $M(\vec{v})$ reveals the phenomenon by outputting a relatively large value.

Second, we can use the generalized measurement function to identify whether the vector is evenly distributed by specifying corresponding $R(\vec{v})$ and $E(\vec{v}, i)$ as follows:

$$R(\vec{v}) = v_{|\vec{v}|} - v_1 \tag{6}$$

$$E(\vec{v}, i) = v_1 + (i - 1) \cdot \frac{R(\vec{v})}{|\vec{v}| - 1}. \tag{7}$$

In (6), the value range $R(\vec{v})$ is defined straightforwardly as the maximum minus the minimum, which is the difference between the last element and the first element of the sorted vector $\vec{v}$. To identify whether the vector is evenly distributed, we evenly divide the interval between the minimum and the maximum into $|\vec{v}| - 1$ subintervals and set the expected values to the ends of these subintervals. Considering that the values of $|\vec{v}|$ ends are sorted in a vector $\vec{v'}$, we set the expected ith value of $\vec{v}$ as the ith value of $\vec{v'}$, as shown in (7).

Finally, we utilize the specified measurement to determine whether the subtree distribution is sufficiently even for the proposed partitioning. Recall that the cake cutting first sorts the polar angles of subtrees relative to the geometric center and then divides subtrees along the sorted order. Therefore, for cake cutting, a distribution is considered even as subtrees are spread evenly in the polar coordinate system with respect to the geometric center. As a point is defined by a radius and a polar angle in the system, we apply the specified measurement to the two terms and combine the two measurements linearly. Equation (8) shows the customized measurement $M_c(S)$ as follows:

$$M_c(S) = \alpha \cdot M(\vec{r}) + \beta \cdot M(\vec{a}) \tag{8}$$

where $\alpha + \beta = 1$, and $\vec{r}$ and $\vec{a}$ are sorted vectors of radius and polar angles in the subtree set S, respectively. In our implementation, we think the weight of the two terms are the same, and thus we set $\alpha = 0.5$ and $\beta = 0.5$.

In conclusion, if the combined measurement (8) outputs a very small value, then we choose cake-cutting-based partitioning to group subtrees; otherwise, we choose the clustering. Actually, since the measurement can be customized for any method, this proposed hybrid solution has flexibility to take any techniques for clustering and partitioning. One can adjust (6) and (7) to define appropriate expected values, and also adjust (8) to combine any concerned terms. But here, we customize for our proposed clustering (Section III-B2a) and partitioning (Section III-B2b).

3) *Embedding-Region Construction:* After the clusters are constructed by hybrid grouping, we next construct embedding regions for the clusters in this stage. For ith tree level, we determine the common connection length, l_i, to uniformly connect subtrees within each cluster of the $(i+1)$th tree level. Then embedding regions for clusters are constructed to locate the potential embedding positions of the ith tree-level nodes. As mentioned earlier, the uniform wirelength of common connection is bounded by the cluster diameter. Hence, we

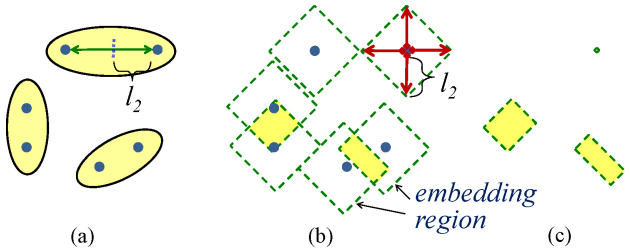

Fig. 12. Example for the embedding-region construction. (a) Six sinks and the clusters are given and BNP is $< 3, 2 >$. l_2 is the common connection length for connecting into the second tree level. (b) TRRs are extended by the connection length, l_2, for all sinks. (c) Embedding regions are constructed by intersecting the extended TRRs for each cluster.

assign the common connection length as the half length of the maximum cluster diameter. This is the exact required common length for a higher-level node to connect its children nodes in its own cluster.

To construct the embedding region for a higher-level node, we first extend the TRRs of its children nodes by the common connection length. Note that the difference of sink capacitances could affect the resulting skew. Thus, we further extend the TRRs of sinks to compensate the loading when handling the lowest level. For each sink, the capacitance of the extended length equals the difference between the maximum sink capacitance and the sink's capacitance.

An extended TRR is the valid embedding region for the parent node with respect to one child. Therefore, to establish new TRRs, the intersection is performed to find the mutually valid region among the extended TRRs of the children nodes.

Fig. 12 gives an example for the embedding-region construction. In Fig. 12(a), six sinks and the clusters are given by grouping stage, where BNP is $< 3, 2 >$. Based on this grouping result, the upper-right cluster has the maximum cluster diameter. Thus, the common connection length, l_2, is decided. Then we extend the TRRs of the subtrees, i.e., sinks, as shown in Fig. 12(b). Finally, the embedding regions of each cluster are constructed, as illustrated in Fig. 12(c), by intersecting the extended TRRs.

After new TRRs are established, we go back to grouping stage for constructing embedding regions of higher levels. As embedding regions of all nodes are ready, the procedure proceeds to node embedding to determine the exact locations of nodes and perform wiring.

4) *Node Embedding:* The exact locations of nodes are determined in this stage. At first, since we intend to minimize routing resource usage, the embedding of the tree root is set to the closest position of the embedding region with respect to the clock source. Then the embedding information is propagated to embed tree nodes level by level in the top–down manner. By following the common connection length obtained in the preceding stages, nodes of the $(i + 1)$th tree level are connected to the embedded nodes of the ith tree level with the exact length l_i. Nevertheless, positions with exact common connection length may not exist when the embedding regions are too close to their embedded parent nodes. As a result, snaking is performed to meet the required length.

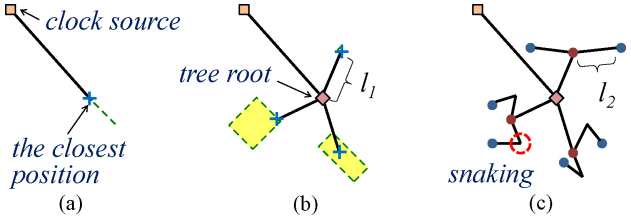

Fig. 13. Example for node embedding. (a) Clock source is connected to the closest position in the embedding region of the tree root, and meanwhile the tree root is embedded at this closest position. (b) Tree root is connected to the positions in the embedding regions of the second tree level with the exact length l_1, and meanwhile the nodes of the second level are embedded at these positions. (c) Embedded nodes of the second level are connected to the positions in the embedding regions of the third tree level with the exact length l_2, and wires are snaked if the positions with the exact length l_2 does not exist.

See Fig. 13 for an example of node embedding, based on the same instance of Fig. 12. First, the clock source is connected to the closest position in the embedding region of the tree root, and meanwhile the tree root is embedded at this closest position, as shown in Fig. 13(a). Next, starting from the embedded tree root, we search exactly the l_1 distance outward from the tree root in the embedding regions of the second tree level. The second tree-level embedding is illustrated in Fig. 13(b). Finally, the same strategy also applies for the third tree level, i.e., the sinks, as illustrated in Fig. 13(c). Note that some distance between sinks and their parents are less than the common connection length. Thus, we make snaking to meet the required length as shown in Fig. 13(c).

5) *Pseudosink Handling:* We extend the presented algorithm for the cases with nonzero pseudosinks. The extension for each stage of the tree construction are explained as follows.

a) For grouping, the clustering and partitioning schemes relax that the sizes of clusters can differ by at most one. In our implementation, the last p clusters chosen in grouping are set to have a smaller size for p pseudo sinks. Those clusters with smaller sizes are the sub-trees to which pseudo sinks are added.

b) For embedding-region construction, we do not construct embedding regions for pseudo sinks to reserve the flexibility of routing snaked wires.

c) For node embedding, we let the embedding regions of pseudo sinks cover the entire chip, and thus they can be placed at proper positions by the same node-embedding method mentioned in Section III-B4.

Also, dangling wires can be established and attached to proper subtrees successfully. Note that for physical implementation, one can insert some cells or flip-flops at the positions of pseudosinks, i.e., the ends of the dangling wires. After the insertion, the structures of paths from the clock source to all tree leaves could be more similar, and thus the skew could be further reduced.

C. Buffer Insertion

In order to meet the slew-rate constraint, buffers are often needed. Based on the constructed symmetrical tree topology, symmetrical buffering could be performed straightforwardly by aligning buffer distribution.

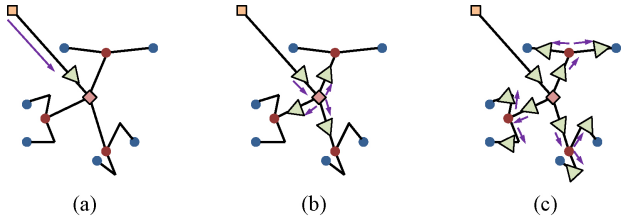

Fig. 14. Example for buffer insertion. (a) First buffer is inserted on the path between the clock source and the tree root. (b) Three identical buffers are inserted at the same distance from the tree root. (c) Resulting symmetrical buffered clock tree is established after buffer insertion on all the branches of the lowest tree level.

We adopt a top–down manner, i.e., a level-by-level way from the clock source to sinks, to insert identical buffers in terms of the type and the size at the same level. By tracing along the tree edges, once the slew rate is about to violate the constraint, identical buffers are inserted for all branches. The slew rate is approximated by accumulated capacitance starting from the latest inserted buffer.

The procedure of buffer insertion is illustrated in Fig. 14, based on the same instance in Figs. 12 and 13. In Fig. 14, the head of an arrow line denotes the tracing direction and the length of an arrow line indicates the accumulated capacitance. Starting from the clock source, we can see that the first buffer is inserted on the path between the clock source and tree root due to the heavy load of the upstream, as shown in Fig. 14(a). Then, the tracing is expanded to three directions at the branch of the root and three identical buffers are inserted at the same distance away from the tree root since the accumulated capacitance is approaching the limitation, as visualized in Fig. 14(b). The same process is repeated until the tracing meets leaves. Fig. 14(c) depicts the resulting configuration of buffer insertion on all the branches of the lowest tree level and the resulting symmetrical buffered clock tree.

D. Complexity Analysis

We briefly analyze the time complexity of the proposed algorithm. There are n sinks and p pseudosinks, where p is obviously $O(n)$. It is not difficult to see that the overall time complexity is dominated by the tree-construction stage, and the tree-construction stage is dominated by the grouping stage. Considering $|S|$ input subtrees for a certain tree level, we claim that the proposed grouping can run in $O(|S|^2)$ time. The analysis consists of three parts: 1) the measurement computation; 2) the clustering; and 3) the partitioning. For measurement, we need sorting, and thus the computation runs in $O(|S| \lg |S|)$ time. For the clustering, we construct $|S|/b$ clusters, and each cluster needs $O(b \cdot |S|)$ time, where b is the branch number, i.e., the cluster size. Thus, the clustering runs in $O(|S|^2)$ time. For the partitioning, the dividing needs $O(|S|^2)$ time dynamic programming, and thus the total time complexity of the partitioning is $O(|S|^2)$ by careful analysis. By combining these three parts, the grouping can run in $O(|S|^2)$ time for a certain tree level. Since branches in size two are always acceptable, the input size of grouping can be shrunk by a factor of at least two. Hence, we conclude that the time complexity of the tree-construction stage is $O(n^2)$ to the number of sinks n. Our algorithm thus runs in $O(n^2)$ time overall.

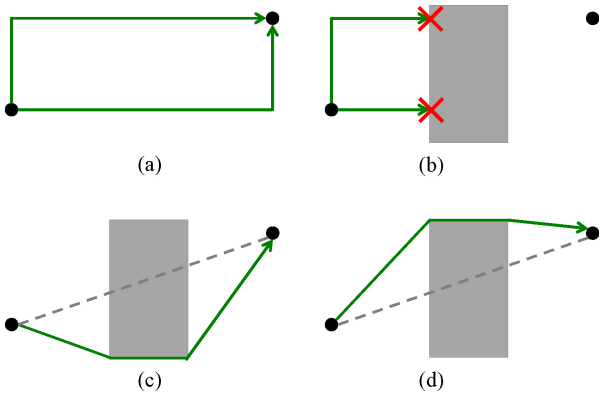

Fig. 15. Example for blockage-avoiding distance estimation. (a) Two L-shape paths are detected and both paths are not blocked. Distance value is the direct Manhattan distance. (b) Both two paths are blocked by blockages. Distance value is the the shortest blockage-avoiding distance of detour paths. (c) One detour path. (d) Another detour path. This path defines the blockage-avoiding distance in this example.

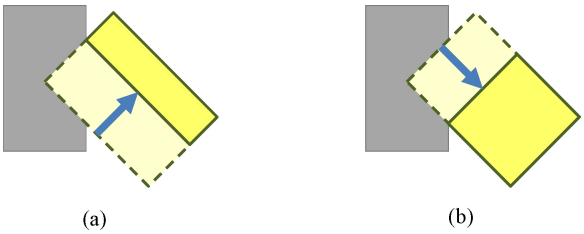

Fig. 16. Example for shrinking a TRR to avoid a blockage. (a) Pushing the lower-left boundary. (b) Pushing the upper-left boundary.

IV. Extensions

In the preceding section, our approach constructs symmetrical clock trees for the cases without blockages. Macros are often seen in modern chip designs, where the macros might be blockages for clock trees. Thus, we should also explore the clock-tree synthesis problem with blockages. To solve the problem, we could modify the processes in the tree-construction stage to avoid blockages. Since we insert buffers along the tree wires, the buffer insertion stage naturally avoids blockages if there are no overlaps between the tree wires and blockages.

Two extensions for our tree construction for blockage avoidance are explained as follows.

1) With the blockage consideration, the direct Manhattan distance might not be feasible since a blockage might block the routing paths. Thus, the distance estimation should be adjusted when grouping subtrees. The new distance is defined as the shortest blockage-avoiding distance. Fig. 15 shows an example for the procedure of the new distance estimation. First, the two L-shaped paths are detected, as shown in Fig. 15(a). If at least one path overlaps no blockages, the distance value is still set as the direct Manhattan distance; if both paths are blocked by blockages [see Fig. 15(b)], otherwise, detours should be considered for the distance estimation.

TABLE I
EXPERIMENTAL RESULTS FOR THE FOUR NONBLOCKAGE BENCHMARKS OF THE ISPD'09 CONTEST [23]

Case	# Sinks	[13] (Team 6 New Results)				[15] (Team 7 New Results)				[21] (Team 4 New Results)				Ours			
		Skew (ps)	CLR (ps)	Usage (fF)	CPU (s)	Skew (ps)	CLR (ps)	Usage (fF)	CPU (s)	Skew (ps)	CLR (ps)	Usage (fF)	CPU (s)	Skew (ps)	CLR (ps)	Usage (fF)	CPU (s)
f11	121	2.87	13.36	117 540	6488	7.12	18.77	101 500	2200	4.48	19.71	100 335	4639	0.08	24.89	94 589	0.02
f12	117	2.61	15.27	109 989	6564	3.06	15.50	96 248	1923	4.09	17.46	96 819	4231	0.15	26.73	83 468	0.02
f21	117	2.74	17.40	120 925	6673	3.02	17.04	110 459	2231	3.87	19.92	109 649	4629	2.07	28.42	105 244	0.01
f22	91	2.23	12.36	85 738	3618	4.11	16.25	69 017	1370	3.67	16.47	68 734	3937	1.12	23.90	72 079	0.01
Avg cmp		Omitted				Omitted				22.05	0.71	1.05	325 025	1.00	1.00	1.00	1

TABLE II
EXPERIMENTAL RESULTS FOR THE IBM BENCHMARKS [25]

Case	# Sinks	[21] W/o Simulation			[21] W/ Simulation			Ours		
		Skew (ps)	Usage (fF)	CPU (s)	Skew (ps)	Usage (fF)	CPU (s)	Skew (ps)	Usage (fF)	CPU (s)
r1	267	14.01	14 001	2	5.01	15 229	5126	1.61	14 426	0.04
r2	598	16.01	28 011	11	6.42	29 234	7374	1.78	29 947	0.10
r3	862	16.53	39 123	26	5.61	41 431	12 739	1.75	45 346	0.19
r4	1903	17.79	89 312	165	5.42	91 015	17 871	1.65	94 967	0.68
r5	3101	21.56	149 875	498	7.03	156 854	26 045	1.75	156 274	1.82
Avg cmp		10.04	0.93	163	3.44	0.98	61 906	1.00	1.00	1

As an example in Fig. 15(c) and (d), two detoured paths are tested and the new distance value is set to the shortest path length of the detoured paths, i.e., the path in Fig. 15(d) is selected. Typically, detoured paths result in larger distance values. Since the objective of the grouping is wirelength minimization, the grouped clusters could avoid blockages as expected.

2) To avoid embedding nodes in blockages, embedding regions should not be overlapped with blockages. Accordingly, the embedding-region construction shrinks TRRs to avoid blockages by pushing the boundaries. Fig. 16 gives an example, where pushing the two left boundaries of the TRR can avoid the blockage. However, pushing boundaries might lose some solution space of feasible embedding positions. For this reason, the resulting new TRR with a larger area is chosen [e.g., the TRR in Fig. 16(b) in this example].

V. EXPERIMENTAL RESULTS

The proposed approach was implemented in the C++ programming language on a 2.6 GHz AMD-64 workstation. We tested the quality of the resulting clock trees based on the four ISPD'09 Clock Network Synthesis Contest benchmarks with no blockages [23] and the IBM benchmarks [25]. Since our approach does not consider blockages, only four out of the seven ISPD'09 Contest benchmark circuits with no blockages, namely, f11, f12, f21, and f22, were used for the experiments. For fair comparison, we used ngspice [18] simulation based on the 45-nm process technology [19] to evaluate the quality, same as the contest.

We first compared our approach with state-of-the-art works in [13], [15], and [21] based on the ISPD Contest benchmarks. The clock-tree synthesizers in the works in [13], [15], and [21] are improved versions of the three ISPD'09 Clock Network Synthesis Contest winners, Team 6, Team 17, and Team 4, respectively. We mainly compared our results with the work in [21] that adopts clock skew as the first objective and optimizes the clock-latency range (CLR) by minimizing the clock skew. Here, CLR is referred to as the difference of signal arrival time under different supply voltages, which is the primary objective in the ISPD'09 Contest. However, it is known to this community that CLR might not be a good metric (to approximate the effects of process variation) since a circuit will not operate at different voltages at the same time. Therefore, we shall focus on the optimization of clock skew and just treat the CLR optimization as a side effect of the skew minimization, same as the work in [21]. Table I shows the results on clock skew (skew), the CLR, the total resource usage (usage), and the running time (CPU). Note that the results of the works in [13] and [15] are listed only for the reader's reference, since their first objective (for the contest) is the CLR minimization.

Compared with [21] that uses ngspice simulation to obtain better clock skews, our approach resulted in much smaller skew in very short running time. On average, the work in [21] resulted in 22.05X clock skew and needs 325025X running time over ours for the four contest benchmarks. The results also reveal that our approach can indeed achieve very small skew (especially, see the results for f11 and f12) because of the symmetrical structure with identical sink loads.

To show the scalability of our approach, we also experimented on the IBM benchmarks with more sinks. In Table II, we compared with [21] on clock skew (skew) reported from ngspice simulation, total resource usage (usage), and running time (CPU), based on two scenarios: performing the clock-tree synthesis with the Elmore delay model (i.e., without simulation) or with ngspice simulation. In either case, our approach resulted in much smaller skew and slightly larger resource usage in very short running time. On average, the work in [21] without (with) ngspice simulation results in 10.04X (3.44X) clock skew and needs 163X (61906X) running time over ours for the IBM benchmarks. In particular, our approach can complete the synthesis in less than 2 s and can obtain small skews reported from ngspice, with marginal overheads of snaking for symmetry. (Note that the skews reported in most of the earlier works on the IBM benchmarks were based on the timing models they adopted, which tend to be much smaller than simulation skews, as shown in Fig. 1.) The results show that our approach scales very well for larger designs.

To show the effectiveness of our proposed hybrid grouping presented in Section III-B2c, we compared with pure

TABLE III
EXPERIMENTAL RESULTS FOR THE IBM BENCHMARKS [25] AND ISPD'09 CONTEST BENCHMARKS

Case	Clustering			Partitioning			Hybrid Grouping		
	Wire (fF)	Skew (ps)	CPU (s)	Wire (fF)	Skew (ps)	CPU (s)	Wire (fF)	Skew (ps)	CPU (s)
f11	66 459	0.22	0.02	49 299	0.08	0.02	49 299	0.08	0.02
f12	65 152	0.18	0.02	44 988	0.15	0.02	44 988	0.15	0.02
f21	74 575	3.18	0.01	59 452	2.07	0.01	59 452	2.07	0.01
f22	40 934	1.55	0.01	36 452	1.12	0.01	36 452	1.12	0.01
r1	9418	1.62	0.04	9448	1.61	0.04	9367	1.61	0.04
r2	20 784	1.73	0.09	22 021	1.71	0.11	20 655	1.78	0.10
r3	29 829	1.62	0.16	30 301	1.76	0.28	29 627	1.75	0.19
r4	66 989	1.66	0.64	66 877	1.66	0.84	66 507	1.65	0.68
r5	110 187	1.80	1.78	114 803	1.75	1.37	107 718	1.75	1.82
Avg cmp	1.14	1.31	0.96	1.02	1.00	1.06	1.00	1.00	1.00

clustering (in Section III-B2a) and pure partitioning (in Section III-B2b) for grouping, based on routing wirelength (wire), clock skew (skew), and running time (CPU). For the implementation of hybrid grouping, since the maximum experimental value of the measurement, (8), is about 0.1, we took 0.05 as the threshold. That is, if the measurement resulted in a value larger than the threshold, the hybrid grouping adopted the clustering scheme; otherwise, it adopted the partitioning scheme. Table III reports the experimental results. Considering the characteristics of the benchmarks, the sinks of the ISPD'09 contest benchmarks are more evenly distributed in the chips, while those of the IBM benchmarks are more concentrated. The experimental results justify our observation in Section III-B2. That is, the pure partitioning led to averagely shorter routing wirelength than the pure clustering for the the ISPD'09 Contest benchmarks, but averagely longer routing wirelength for the IBM benchmarks. Since our proposed hybrid approach can adaptively make the choice, the hybrid grouping can achieve the smallest routing wirelength for both the ISPD'09 Contest benchmarks and the IBM benchmarks. Furthermore, the hybrid grouping maintained good clock skew with similar efficiency.

For reader's reference, we also compared with the DME algorithm [1], [2], [7] to show an empirically worst-case overhead of wirelength usage. We implemented the DME algorithm based on the Elmore delay model and tested the algorithm on both the ISPD benchmarks and the IBM benchmarks. The results show that the DME algorithm obtains averagely 39% shorter wirelength with averagely 646X larger skew than our proposed symmetrical structure.

In Fig. 17, the running time of our algorithm with hybrid grouping is plotted as a function of the number of sinks. To avoid possible misleading in the regression analysis because of the many small test cases, we ignored the benchmarks under 200 sinks and randomly generated larger test cases between 1000 and 10 000 sinks for the curve in Fig. 17. Empirically, the time complexity of our algorithm is about $O(n^{1.66})$ to the number of sinks n, with the least square analysis for the log–log plot of the function. The empirical time complexity is under the theoretical worst-case complexity of $O(n^2)$ presented in Section III-D. The experimental results show that our proposed algorithm is very effective and efficient.

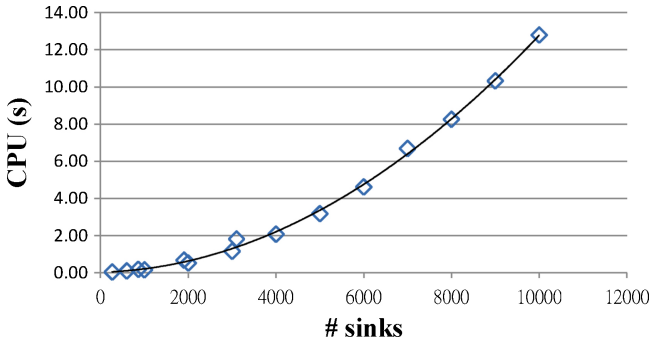

Fig. 17. Running time of our algorithm with hybrid grouping.

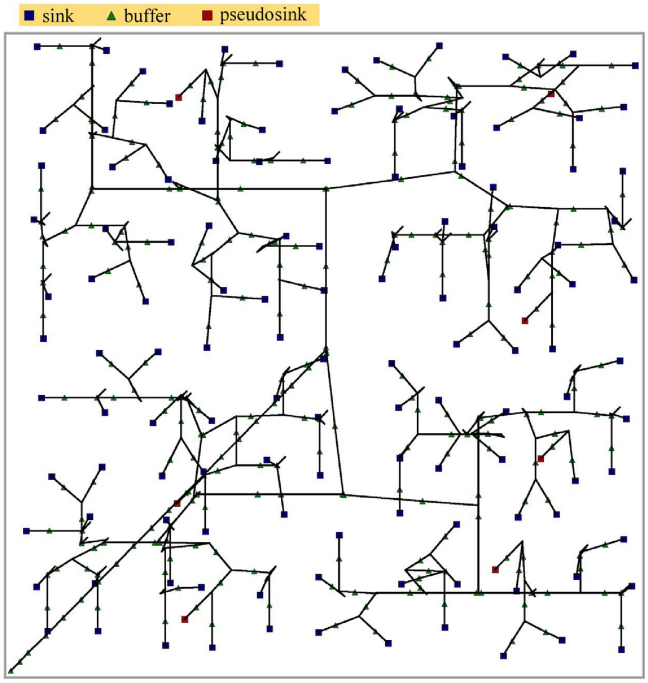

Fig. 18. Resulting clock tree for the benchmark circuit f11.

VI. CONCLUSION

We have presented a fast timing-model independent BCTS method to construct a symmetrical clock tree. At each level of a symmetrical clock tree, the number of branches, the wirelength, and the inserted buffers are almost the same. By symmetrically constructing a clock tree, the clock skew can be minimized without referring to simulation information. Experimental results showed that our approach could not only efficiently construct a buffered clock tree but also effectively minimize the clock skew with little wiring overhead.

With this symmetrical structure, in particular, a designer can get rid of insufficiently accurate timing models and prohibitively time-consuming simulation for large-scale clock-tree synthesis. We believe that the symmetrical structure provided a promising direction to solve the accuracy and scalability problems for current and future clock-network designs.

As the total capacitance usage significantly affects the power consumption of a clock network, future work shall improve the capacitance usage of wiring and buffering. For capacitance minimization, we consider the capacitance-usage-aware branch-number planning. We have an observation: different orders of branch numbers could affect the grouping

configurations, which results in different total wirelengths; moreover, different total wirelengths might need different buffering configurations, which requires different numbers of buffers. Accordingly, considering the wiring and buffering in earlier branch-number planning can further reduce the capacitance usage and thus the power consumption.

Another research direction lies in the optimization of the clock insertion delay (clock latency). The insertion delay is an important criterion to estimate the on-chip skew variation. That is, a certain percentage of the insertion delay can be treated as a range of the skew variability. If the insertion delay can be reduced, so is the skew variability. To optimize the insertion delay, new buffer insertion and gate-sizing techniques should be introduced. Thus, our future work shall develop new techniques to improve the buffer insertion stage for the insertion delay optimization.

Our future research direction also lies in the handling of useful skew. The formulation in this paper was focused on pure skew minimization. For some more sophisticated designs, useful skew might be required. To handle this problem, we should make the resulting skew within some predefined range for useful skew operation. Therefore, it is desirable to develop symmetrical clock-tree structures with predefined skew bounds for useful skew applications.

REFERENCES

[1] K. D. Boese and A. B. Kahng, "Zero-skew clock routing trees with minimum wirelength," in *Proc. ASIC Conf.*, 1992, pp. 17–21.

[2] T.-H. Chao, Y.-C. H. Hsu, and J.-M. Ho, "Zero skew clock net routing," in *Proc. Des. Automat. Conf.*, 1992, pp. 518–523.

[3] R. Chaturvedi and J. Hu, "Buffered clock tree for high quality IC design," in *Proc. Int. Symp. Qual. Electron. Des.*, 2004, pp. 381–386.

[4] Y. P. Chen and D. F. Wong, "An algorithm for zero-skew clock tree routing with buffer insertion," in *Proc. Eur. Conf. Des. Test*, 1996, pp. 230–236.

[5] Y.-Y. Chen, C. Dong, and D. Chen, "Clock tree synthesis under aggressive buffer insertion," in *Proc. Des. Automat. Conf.*, 2010, pp. 86–89.

[6] C.-M. Chang, S.-H. Huang, Y.-K. Ho, J.-Z. Lin, H.-P. Wang, and Y.-S. Lu, "Type-matching clock tree for zero skew clock gating," in *Proc. Des. Automat. Conf.*, 2008, pp. 714–719.

[7] M. Edahiro, "Minimum skew and minimum path length routing in VLSI layout design," *NEC Res. Develop.*, vol. 32, no. 4, pp. 569–575, 1991.

[8] M. Edahiro, "A clustering-based optimization algorithm in zero-skew routings," in *Proc. Des. Automat. Conf.*, 1993, pp. 612–616.

[9] EE Times. (2009, Apr. 7). *IBM Fellow: Moore's Law Defunct* [Online]. Available: http://www.eetimes.com/electronics-news/4082011/IBM-Fellow-Moore-s-Law-defunct

[10] W. C. Elmore, "The transient response of damped linear network with particular regard to wideband amplifier," *J. Appl. Phys.*, vol. 19, no. 1, pp. 55–63, 1948.

[11] M. Jackson, A. Srinivasan, and E. S. Kuh, "Clock routing for high performance ICs," in *Proc. Des. Automat. Conf.*, 1990, pp. 573–579.

[12] A. Kahng, J. Cong, and G. Robins, "High-performance clock routing based on recursive geometric matching," in *Proc. Des. Automat. Conf.*, 1991, pp. 322–327.

[13] D.-J. Lee and I. L. Markov, "Contango: Integrated optimizations for SoC clock networks," in *Proc. Des. Automat. Test Eur.*, 2010, pp. 1468–1473.

[14] D.-J. Lee, M.-C. Kim, and I. L. Markov, "Low-power clock trees for CPUs," in *Proc. Int. Conf. Comput. Aided Des.*, 2010, pp. 444–451.

[15] W.-H. Liu, Y.-L. Li, and H.-C. Chen, "Minimizing clock latency range in robust clock tree synthesis," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2010, pp. 389–394.

[16] J. Lu, W.-K. Chow, C.-W. Sham, and F.-Y. Young "A dual-MST approach for clock network synthesis," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2010, pp. 467–473.

[17] A. D. Mehta, Y.-P. Chen, N. Menezes, D. F. Wong, and L. T. Pileggi, "Clustering and load balancing for buffered clock tree synthesis," in *Proc. Int. Conf. Comput. Aided Des.*, 1997, pp. 217–223.

[18] *ngspice* [Online]. Available: http://ngspice.sourceforge.net

[19] *Predictive Technology Model* [Online]. Available: http://www.eas.asu.edu/$\textasciitilde$ptm

[20] A. Rajaram and D. Z. Pan, "MeshWorks: An efficient framework for planning, synthesis and optimization of clock mesh networks," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2008, pp. 250–257.

[21] X.-W. Shih, C.-C. Cheng, Y.-K. Ho, and Y.-W. Chang, "Blockage-avoiding buffered clock-tree synthesis for clock latency-range and skew minimization," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2010, pp. 395–400.

[22] X.-W. Shih and Y.-W. Chang, "Fast timing-model independent clock-tree synthesis," in *Proc. Des. Automat. Conf.*, 2010, pp. 80–85.

[23] C. N. Sze, P. Restle, G.-J. Nam, and C. Alpert, "ISPD2009 clock network synthesis contest," in *Proc. Int. Symp. Phys. Des.*, 2009, pp. 149–150.

[24] G. E. Tellez and M. Sarrafzadeh, "Minimal buffer insertion in clock trees with skew and slew rate constraints," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 16, no. 4, pp. 333–342, Apr. 1997.

[25] R.-S. Tsay, "Exact zero skew," in *Proc. Des. Automat. Conf.*, 1991, pp. 336–339.

[26] L. Xiao, Z. Xiao, Z. Qian, Y. Jiang, T. Huang, H. Tian, and F.-Y. Young, "Local clock skew minimization using blockage-aware mixed tree-mesh clock network," in *Proc. Int. Conf. Comput. Aided Des.*, 2010, pp. 458–462.

Xin-Wei Shih received the B.S. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2008. He is currently pursuing the Ph.D. degree with the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan.

His current research interests include clock-network synthesis for modern high-performance nanometer synchronous chip design.

Yao-Wen Chang (S'94–A'96–M'96–SM'12) received the B.S. degree from National Taiwan University (NTU), Taipei, Taiwan, in 1988, and the M.S. and Ph.D. degrees from the University of Texas at Austin, Austin, in 1993 and 1996, respectively, all in computer science.

He is currently the Director of the Graduate Institute of Electronics Engineering and a Distinguished Professor of the Department of Electrical Engineering, NTU. His current research interests include very large-scale integration physical designs and design for manufacturability. He has been working closely with industry in these areas. He has co-edited one textbook on electronic design automation (EDA) and co-authored one book on routing and over 200 ACM/IEEE conference or journal papers in these areas.

Dr. Chang was a first-place winner of the 2011 PATMOS Timing Analysis Contest and a four-times winner of the ACM ISPD Contests on Placement, Global Routing, and Clock Network Synthesis. He was a recipient of six Best Paper Awards (ICCD, etc.) and 20 Best Paper Award Nominations from DAC (five times), ICCAD (four times), and others in the past ten years. He has received many research and teaching awards, such as the Distinguished Research Award from National Science Council of Taiwan (twice), the IBM Faculty Award, the CIEE Distinguished EE Professorship, the MXIC Young Chair Professorship, and the Excellent Teaching Award from NTU (seven times). He is currently an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and the IEEE DESIGN AND TEST OF COMPUTERS. He is an Editor of the *Journal of Information Science and Engineering*. He was the General and Steering Committee Chair of ISPD and the Program Chair of ASP-DAC, FPT, and ISPD. He is currently an IEEE CEDA Executive Committee Member, the Vice Technical Program Committee Chair of the ICCAD Executive Committee, the ASP-DAC Steering Committee Member, and the ACM/SIGDA Physical Design Technical Committee Member. He was a Technical Program Committee Member of major EDA conferences, including ASP-DAC, DAC, DATE, FPL, FPT, GLSVLSI, ICCAD, ICCD, ISPD, SLIP, SOCC, and VLSI-DAT. He was an Independent Board Director with Genesys Logic, Taiwan, a Technical Consultant with Faraday, Taiwan, MediaTek, Taiwan, and RealTek, Taiwan, the Chair of the EDA Consortium of the Ministry of Education, Taiwan, and a member of the Board of Governors of Taiwan IC Design Society.