Variation Tolerant Buffered Clock Network Synthesis with Cross Links

Anand Rajaram
Dept. of ECE, University of Texas,
Austin, Tx 78712
Texas Instruments Inc., Dallas, Tx 75243
anandr@mail.utexas.edu

David Z. Pan *
Dept. of ECE, University of Texas,
Austin, Tx 78712
dpan@ece.utexas.edu

ABSTRACT

Clock network synthesis is a key step in the ultra deep submicron (UDSM) VLSI Designs. Most existing clock network synthesis algorithms are designed for nominal operating condition, which are insufficient to address the growing problem of process, voltage and temperature (PVT) fluctuations. Link based clock networks have been suggested as a possible way of reducing skew variability [1–3]. However, [1,2] deal with only unbuffered clock networks, making them impractical. In [3], the problem of constructing a link based buffered clock network has been addressed . But [3] requires special kind of tunable buffers, which might consume more area/power and might not be available for all designs. Also, [3] uses SPICE for tuning the locations of internal nodes and buffer delays, thereby making it slow even for clock networks with a few hundred sinks. In this paper, we propose a unified algorithm for synthesizing a variation tolerant, balanced buffered clock network with cross links. Our approach can make use of ordinary buffers and does not require SPICE for clock network synthesis. SPICE based Monte Carlo simulations show that our methodology results in a buffered clock network with 50% reduction in skew variability with minimal increase in wire-length, buffer area and CPU time.

Categories and Subject Descriptors

B.7.2 [**INTEGRATED CIRCUITS**]: Design Aids

General Terms

Algorithms, Performance

Keywords

VLSI CAD, Physical Design, Clock Network, Non-tree Clocks.

*This work is partially supported by SRC and IBM Faculty Award.

ISPD'06, April 9–12, 2006, San Jose, California, USA.

1. INTRODUCTION

Clock distribution networks (CDNs) are of great importance in any synchronous VLSI chip because the pace of almost every data transfer is determined by the clock signal. As one of the largest and fastest switching nets in any design, the CDN has tremendous influence on the over all performance of the chip [4]. Realizing its importance, numerous works have focused on the problem of clock network synthesis [7–18]. In the majority of these works, only traditional parameters like skew, wire/buffer area and power are considered. However, in the sub-100 nm technologies, variation effects like manufacturing variations [5, 20], temperature changes and power supply noise [6] are becoming very significant. Hence, it is vital to address the variation effects during clock network synthesis. Most of the existing algorithms, including recent ones like [15–17] do not consider variation effects and use only the nominal values of device/interconnect parameters to achieve the target skews. The main drawback of such an approach is that even if clock skew constraints are met at design time (for nominal values of device/interconnect parameters), PVT variations can introduce unwanted clock skew during the chip fabrication, thereby affecting performance and timing yield.

Link-based clock network [1–3] has been proposed as a possible method to reduce skew variability. However, [1, 2] address only unbuffered clock networks, making them impractical. The work of [3] attempts to solve the problem of constructing a link based buffered clock network in which special tunable buffers are used to obtain a low-skew (in SPICE) buffered clock network. Once a good low-skew buffered clock network is obtained, [3] uses the algorithms of [2] to select the links to be inserted. Finally, SPICE simulations are used to tune the buffers and internal node locations. However, the use of tunable buffers might result in increased area and power consumption. Moreover, the tunable buffers might not be available in all design libraries. Also, because of the use of SPICE for tuning, [3] is slow for even clock network of a few hundred sinks.

In this work, we propose an efficient algorithm for synthesizing a link-based clock network in which we have attempted to overcome the above mentioned drawbacks of [3]. The important contributions of this work are:

- Our methodology uses ordinary buffer cells and does not require any special tunable buffer cells unlike [3].
- We adapt and modify the efficient & accurate delay evaluation method of [19] to consider the slew and resistive shielding effects during clock network synthesis,

thereby avoiding the use of SPICE for constructing the clock network. This also helps us to overcome the inaccuracy of the simple Elmore delay model, which is used by most clock tree synthesis algorithms including the recent ones like [15–17]. Thus, our algorithm is both fast and accurate.

- We propose a new merging scheme for clock tree synthesis to obtain a *link insertion friendly* balanced clock tree.

Monte Carlo simulations in SPICE show that our algorithm can reduce skew variability by 50% on an average with no penalty in resources or runtime.

2. BACKGROUND AND MOTIVATION

In this section, we briefly review concept of link insertion for the sake of completeness. Then we discuss the reasons as to why link insertion in a buffered clock network is a non-trivial problem under accurate delay model. We also review existing buffered clock tree synthesis algorithms and point out the reasons that motivate us to propose a new clock tree synthesis algorithm to obtain a *link insertion friendly* buffered clock tree.

2.1 Link Insertion Overview

The key idea of the link-based clock network [1,2] is to obtain a non-tree by inserting links in a given clock tree such that redundant source to sink paths are established. The links introduce a strong correlation between the different sink delays, thereby reducing the skew variability. An example of link insertion in an unbuffered clock tree is shown in Figure (1) (a). In this tree, the sinks 4 and 5 are located physically very close to each other but still they share no common path other than the clock source S. As demonstrated in [1, 2], adding a link (shown in dotted lines) between nodes such as 4 and 5 result in reducing the effect of variation factors on the clock skew.

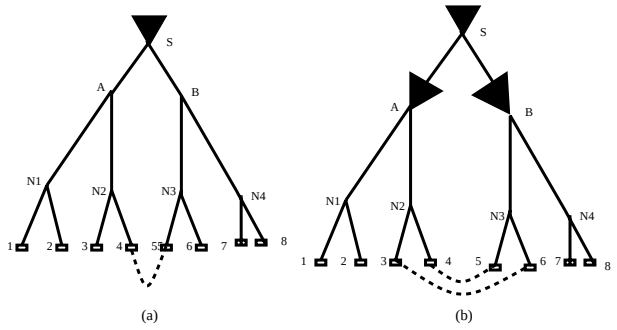

Figure 1: An example of link-based non-tree. (a) Unbuffered case. (b) Buffered case.

2.2 Challenges in Buffered Clock Tree Link Insertion

The algorithms used in [1, 2] is applicable only to unbuffered clock networks and cannot be applied to buffered case because of the reasons discussed below.

Chicken-egg problem: Inserting a link in a buffered clock tree introduces a chicken-egg problem between the location of buffer driving the linked nodes, the input slew of the buffers and the downstream delays. For example, in Figure (1) (b), the link between nodes 4 and 5 increases the loads driven by the buffers A and B, which affects the locations and input slews to the buffers, which in-turn affects the delays from buffer A(B) to sink node 4(5). But, the skew between sink nodes 4 and 5 needs to be evaluated in order to select them for link insertion, thus completing the cyclic dependency. This fact has also been noted in [3].

Accuracy of delay model: Accurate delay models must be considered while inserting links in a buffered clock tree because of the following reasons:

- As shown in [1], a link will be beneficial only when it is inserted between two sink nodes with zero or near-zero skew . While Elmore delay has been shown to have good fidelity for the unbuffered clock trees w.r.t SPICE in [1,2], the fidelity is poor for a buffered clock tree as demonstrated in [18].
- Adding links between two sinks driven by different buffers introduces the problem of multi-driver nets. For example, in Figure (1) (b), the link between nodes 4 and 5 has two drivers, namely buffers A and B. If the links are not selected considering accurate delays, then it is possible to insert links between nodes whose delay values are quite different. This might increase the nominal short circuit power of the clock network because of the virtual shorting of Vdd and Vss (Source and Ground) that might occur when one of the drivers gets turned much ahead of others or vice versa.

Thus, it is clear from the above points that link insertion for buffered clock tree is a non-trivial problem. In order to insert links in a buffered clock tree, we required the buffered clock tree to have a good nominal skew in an accurate delay model. By making sure that the nodes have very similar accurate delay values, we not only make the skew variability better, but also reduce the nominal short-circuit power consumption.

2.3 Existing clock tree synthesis algorithms

One of the pioneering algorithms for clock routing was proposed in [7], in which a zero skew clock routing was obtained by recursively merging a pair of zero skew subtrees until a single clock tree is obtained. The zero-skew principle in [7] was extended in the DME algorithm [8] for wire length minimization. However, [7,8] addressed only the problem of an unbuffered clock tree. The problem of constructing a zero skew buffered clock tree under Elmore delay model was solved in [9, 10]. The optimal clock buffering for a given topology and buffer library was solved in [11]. A heuristic for synthesizing a low-power buffered clock tree using the Elmore delay model was proposed in [12]. Buffer and wire sizing were done so as to reduce power and maintain the zero skew property under Elmore delay. None of the above clock tree synthesis algorithms consider clock signal slew *during* the synthesis of the clock tree.

To our best knowledge, [13] was the first work that explicitly considered slew during buffer insertion. But it assumes a *given* unbuffered clock tree, which may result in very sub-optimal solution compared to simultaneous buffering and clock routing as shown in [9]. In [18], a SPICE based, time domain based buffer/wire sizing has been proposed which results in greatly reduced skew in SPICE. To the best of our knowledge, [18] is the only work that aims to reduce the actual clock skew in SPICE. However, since it directly uses SPICE for tuning the clock network, the runtime may be prohibitively high. In [14], the important problem of clock

buffer load imbalance is addressed. In the previous algorithms like [9], different clock buffers at a given level from the clock source may drive different loads. The methodology in [14] attempts to solve this problem by using a clustering approach. But such a clustering and load balancing approach usually results in excessive wire length due to wire snaking when the two clusters to be merged do not have similar target delays.

In the recent works of [15, 16], the problem for optimal buffer/wire sizing in clock network has been studied under the Elmore delay model. In [17], a new merging scheme has been proposed for prescribed skews which usually results in considerably less wire-length compared to the other algorithms. However, this algorithm results in highly unbalanced clock structure. The balanced clock structure issue has been addressed in [9, 14] at the cost of excessive total wire lengths compared to [17].

2.4 Requirements of a Link Insertion Friendly Buffered Clock Tree Synthesis

Based on our discussions in section 2.2, the requirements of a *link insertion friendly* clock tree synthesis algorithm are:

Accurate and fast delay model during synthesis: This requirement implies that the effect of slew and resistive shielding are to be considered, which will ensure that the sink node pairs are selected based on accurate skew values. It also implies that SPICE should not be used for the clock tree synthesis as it may increase the runtime considerably. However, as discussed in section 2.3, most of the existing buffered CTS algorithms, including the recent ones like [15–17], use Elmore delay or use SPICE for synthesis like [18].

Balanced clock tree without excessive wire snaking: We define a balanced clock tree as one in which identical buffers are inserted at a given level from the clock source. Also, a balanced buffered clock tree will have the same number of buffer levels from the source to each clock sink. Figure (2) (a) and (b) shows an example of unbalanced and balanced buffered clock tree respectively.

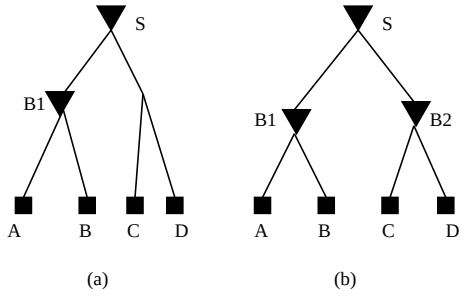

Figure 2: (a) An example of an unbalanced buffered clock tree; (b) An example of a balanced buffered clock tree.

A key advantage in having a balanced clock structure is that it will be much more tolerant to variation. For example, if the nominal delays from source to sinks in both Figure (2) (a) and (b) is 100 ps and if the nominal gate delay is 20ps. Under nominal conditions, both Figure (2) (a) and (b) will have identical skews. However, when variation effects become more and more severe, the scaling of the device delays and interconnect delays need not match. For example, due to certain change in operating temperature or voltage, the device delay doubles and interconnect delay becomes one half of the nominal values. Then the balanced structure is much more tolerant to the variation. In the UDSM technologies, it is becoming increasingly difficult to capture all the different variation effects. This further motivates us to use the balanced clock structure to improve tolerance to variations. Many recent clock tree synthesis algorithms like [15, 16, 18] use buffers of different sizes and tune them in such a way that the skew and delay targets are met at the nominal values of device and interconnect parameters. However, due to the PVT variations, significant skew can be generated in such clock trees.

Also, the total wire-length of the clock network should be as less as possible in spite of maintaining a balanced structure. The reason for this requirement is that most of the existing algorithms that obtain a balanced clock structure like [9, 14] achieve load balancing by clustering methods, which often result in excessive wire snaking. This is undesirable because excessive wire-length increases the total power and resource consumption. It may be emphasized here that, even thought the idea of a balanced clock tree is well known, to the best of our knowledge, there is no work that guarantees the balanced nature of the resulting clock tree without performing clustering.

A fact to be noted here is that none of the existing clock tree synthesis algorithms satisfy all the above requirements. Though each of the above requirements have been addressed in bits and pieces, to the best of our knowledge, there is no unified clock tree synthesis algorithm that addresses all the above issues in a systematic way. In this work, we propose such a unified clock tree synthesis methodology, which will result in a *link insertion friendly* buffered clock tree.

3. ITERATIVE DELAY EVALUATION AND BACKWARD SLEW PROPAGATION

The work of [19] introduces a fast, accurate and iterative delay evaluation procedure which has the elegance and simplicity of Elmore delay with much improved accuracy. The method of [19] is mainly for delay analysis and cannot be directly applied for clock tree synthesis. This is because [19] uses a technique called *slew propagation* in which the slew is propagated from the signal source to the sinks. But, during bottom-up clock tree synthesis, the slew at the source in unknown and hence the method of [19] cannot be used. To overcome this, we solve the inverse of the *slew propagation* called *backward slew propagation* in which we propagate the slew targets in a bottom-up fashion, which can be applied during clock tree synthesis. In this section, we briefly review the iterative delay evaluation of [19] followed by the explanation of our backward slew propagation method.

3.1 Iterative delay and slew evaluation

Ideally, we would like to have a delay evaluation procedure that is as efficient and elegant as Elmore delay while accounting for resistive shielding and signal slew effects. The iterative delay estimation procedure of [19] is such a delay model, used in IBM's physical design closure tool. The procedure explicitly considers the signal slew in delay evaluation and accounts for the interdependence between the input signal slew of a node and the *effective* load seen by the node. However, the procedure is mainly for delay evaluation. In

this paper, we extend it for the purpose of clock tree synthesis by introducing the notion of *required slew* similar to the concept of *required skew*.

Consider the Figure (3) of a simple RC network connecting nodes v and a. An input ramp voltage with a signal transition time of t_v is applied at the node v. The transition time at the output node of the RC segment, namely node a is given by t_a. According to Elmore delay, the total down stream capacitance *seen* by the node v is C. However, because of the resistive shielding effect of the resistance R, only a fraction of the capacitance C is actually seen by the node v, which is usually referred to by the name *effective capacitance* [19]. According to [19], the value of this effective capacitance is give as:

$$C_{eff} = K * C \tag{1}$$

where K is the scaling factor defined as:

$$K = 1 - 2x(1 - e^{-\frac{1}{2x}}), \quad where \quad x = \frac{RC}{t_v} \tag{2}$$

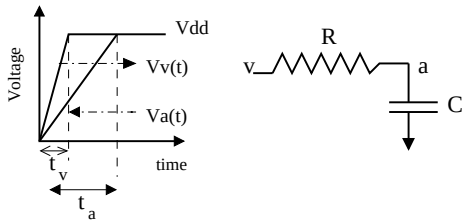

Figure 3: (a) Definitions of transition times for nodes v and a. (b) A simple example of RC network.

It should noted that the value of the effective capacitance seen by node v and the slew rate at v are interdependent. The output slew rate of the CMOS buffer depends on both the input slew *and* the load capacitance [19]. From Equations (1) and (2), the effective load capacitance seen by the buffer output depends on the slew at the buffer output. This factor introduces a chicken-egg problem which is addressed in [19] using an iterative delay evaluation technique.

3.2 Backward propagation of slew

In order to consider the node slews during the clock tree synthesis, we need to calculate the signal slew rate during the bottom-up topology generation phase of the DME [8, 17] algorithm. However, by definition, the slew rate at a child node can be calculated only when the slew rate at the parent node is known. For example, in Figure (3), the slew rate at node a can be obtained only when the slew rate at node v is known. An efficient method for obtaining the transition times at the nodes of the clock tree for a given transition time at the source node has been proposed in [19]. Considering the Figure (3), the transition time at node v is given as t_v. Given t_v and the R, C values, the transition time time at node a can be obtained using the method of [19] as:

$$t_a = \frac{t_v}{1 - x(1 - e^{-\frac{1}{x}})}, \quad where \quad x = \frac{RC}{t_v} \tag{3}$$

In order to consider the slew *during* clock tree synthesis, we would like to get an inverse of Equation (3). That is, we would like to get the value of t_v for a given value of t_a. Such an inverse expression will enable us to consider slew during the bottom up phase of clock tree synthesis. Such an inverse expression can be obtained as follows: define a new variable called y and using 3, we have:

$$y = \frac{RC}{t_a} = \frac{RC(1 - x(1 - e^{-\frac{1}{x}}))}{t_v}$$

which can be simplified to

$$y = x(1 - x(1 - e^{-\frac{1}{x}})) \tag{4}$$

The plot of Equation (4) is shown in Figure (4). As it can be seen from the plot, the value of y reaches a saturation point after the value of x reaches a value of roughly 20. The saturation value of y is 0.5, which can also be verified by applying the Taylor series approximation for the term $e^{-\frac{1}{x}}$ as $1 - \frac{1}{x} + \frac{1}{2x^2}$. Using this approximation in Equation (4) will reduce the value of y to 0.5. A key use of the above observation is that for a given value of x, there is an unique value of y and vice versa. Thus, when we are given the *required* slew value at output node, we can obtain the value of y, which can be used to uniquely determine the value of x, which in turn can be used to obtain the *required* input slew. In other words, if we have a slew requirement at the child node a in Figure (3), using that we can uniquely obtain the *required* slew value at the parent node v. This technique can be used to build a buffered clock tree with simultaneous slew considerations in a bottom-up fashion.

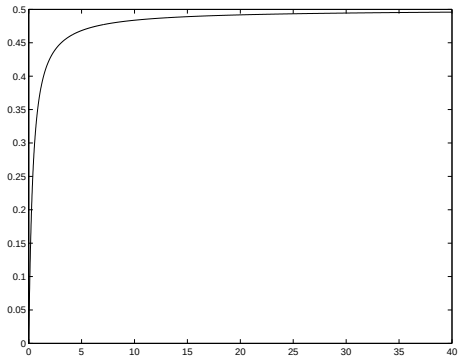

Figure 4: Plot of x (ratio of RC and input slew) versus y (ratio of RC and output slew) of Equation (4).

4. LINK INSERTION FRIENDLY CLOCK TREE SYNTHESIS

In this section, we introduce our link insertion friendly clock tree synthesis algorithm in which we have attempted to simultaneously consider all the requirements outlined in section 2.4. To the best of our knowledge, this is the first work in which all these factors are considered in a unified and systematic way. First, we will consider the problem of merging two subtrees using the backward slew propagation algorithm of the previous section. Then we introduce our novel merging scheme which guarantees the construction of a perfectly balanced buffered clock tree while simultaneously reducing the wire-length and maintaining load balance.

The high level framework of our algorithm is similar to the DME based algorithms like [9,17] in which the first step is the topology generation phase in which different subtrees

are merged recursively based on a merging cost. After all the subtrees are merged into a single tree, a top down embedding is done to finalize the locations of the clock tree nodes.

4.1 Subtree merging with backward slew propagation

Consider Figure (5) in which two subtrees T_i and T_j (rooted at nodes i and j respectively) are to be merged to form a new subtree T_p with node p as the root. In the traditional merging, the lengths of segments $l_{p,i}$ and $l_{p,j}$ are determined in such a way that the Elmore delay from v to the sinks of both T_i and T_j are identical. During this step, the entire downstream capacitance at nodes i and j are considered. However, the delay evaluation method of [19] considers only the *effective* capacitance at the subtrees T_i and T_j while determining the edge lengths. The delay from node p to nodes i and j are given as [19]:

$$D(p,i) = \frac{1}{2}rcl_{p,i}^2 + rl_{p,i}C_{eff1} \quad (5)$$

$$D(p,j) = \frac{1}{2}rcl_{p,j}^2 + rl_{p,j}C_{eff2}$$

where, r and c are the unit length resistance and capacitance, respectively. C_{eff1} and C_{eff2} are the effective downstream capacitances of nodes i and j respectively. It may be noted that for clock sinks, the value of effective capacitance is equal to the load capacitance.

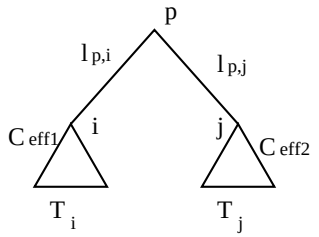

Figure 5: An example of subtree merger using effective downstream capacitance

In order to balance the *effective delays* of the two subtrees, the following equation must be satisfied:

$$D_i + D(p,i) = D_j + D(p,j) \quad (6)$$

where D_i and D_j are the delays from nodes i and j to their respective sink nodes. The edge lengths $l_{p,i}$ and $l_{p,j}$ can be obtained by solving Equation (6) with the condition that $l_{p,i} + l_{p,j} = L$, where L is the Manhattan distance between the nodes (or the merging segment of the nodes) i and j. Wire snaking can be used to match the delays if wire-lengths greater than L is required [17]. Once the appropriate segment lengths have determined, the *required slew* at the parent node p can be calculated using Equations (3) to (4).

Figure (6) explains this step in detail. A point to be noted regarding bottom-up transition time limit propagation is that, during merging of two subtrees with different transition time limits, two independent transition limits (one for each child node) can be obtained for the new root p, denoted by t_p^i and t_p^j in the Figure (6). Since the transition time limits are defined as the maximum signal rise time acceptable at a particular node, we pick only the tighter requirement of the two. Also, selecting the lesser transition time might impact the zero skew property within the subtree that has bigger transition time. However the effect of this is minimal based on our experimental experience.

Procedure: $FindSlew(T_p)$
Input: A subtree rooted at node p
Output: The signal transition time limit at p.

1. If p is a sink
   t_p = Transition time limit set by user.
   return.
2. $i = LeftChild(p); j = RightChild(p)$.
3. $t_i, t_j \leftarrow$ Transition time limit at nodes i and j.
4. $R1 = rl_{p,i}; R2 = rl_{p,j}$.
5. $C1 = C_{eff1} + 0.5cl_{p,i}; C2 = C_{eff2} + 0.5cl_{p,j}$.
6. $y1 = \frac{R1C1}{t_i}; y2 = \frac{R2C2}{t_j}$ (Similar to eqn.2)
7. For $y1$ and $y2$, obtain the corresponding unique values of $x1$ and $x2$ using eqn.(4).
8. Using $x1$ and $x2$, obtain transition time limits at node p w.r.t nodes i and j as:
   $t_p^i = \frac{R1C1}{x1}; t_p^j = \frac{R2C2}{x2}$
9. return $min(t_p^i, t_p^j)$

Figure 6: Procedure to evaluate the signal transition values of a node given the transition values of the child nodes.

Procedure: $FindEffectiveCapacitance(T_p)$
Input: A subtree rooted at node p
Output: The effective downstream capacitance at node p.

1. If p is a sink
   C_{eff} = Sink load capacitance
   return.
2. $i = LeftChild(p); j = RightChild(p)$
3. $C_{eff1}, C_{eff2} \leftarrow$ Effective downstream capacitance of i, j
4. $R1 = rl_{p,i}; R2 = rl_{p,j}$.
5. $C1 = C_{eff1} + 0.5cl_{p,i}; C2 = C_{eff2} + 0.5cl_{p,j}$.
6. t_p = transition time limit of node p
7. $K1 = \frac{R1C1}{t_p}; K2 = \frac{R2C2}{t_p}$;
8. $C_{eff} = K1C1 + K2C2 + 0.5c(l_{p,i} + l_{p,j})$; return.

Figure 7: The procedure to evaluate the effective downstream capacitance recursively.

Once the *required slew* information at the root node p is available, the effective downstream capacitance at node p can also be calculated as demonstrated in Figure (7).

Thus, using the algorithms of Figures (6) and (7), we can merge a give pair of subtrees and obtain the values of slew and effective downstream capacitance of the new subtree. In order for this method to be applied in a recursive fashion, the slew requirements at the clock sink nodes must be predefined by the user. This can be used during the bottom-up clock tree construction as shown in the next section.

4.2 Balanced CTS algorithm

As discussed in section 2.4, one of the key disadvantages of several existing algorithms is the difficulty in getting a balanced clock tree without a wire-length penalty. We propose to address this key problem using a novel merging scheme, which is explained below.

In any merging scheme, node pairs to be merged are selected as per a cost function. In most of the traditional merging schemes like [8], node pairs that are physically closest are merged together with the intention of reducing the total wire length. But, as noted in [17], this might result in excessive

wire snaking when the nodes to be merged do not have similar delays. The algorithm in [9] selects the node pairs that result in the smallest delay after the merger. This generally results in a more balanced tree. However, the wire-length consumed is generally more. In [17], the pair that results in the minimal merging wire-length are merged. Since this is in some ways similar to the minimum spanning tree algorithm (which at each step selects the new edge with minimal cost), it results in much a lower wire-length when compared to the approaches of [8] and [9]. However, as noted in [17], it might result in an highly unbalanced clock tree. In our work, we modify the cost function of [17] such that a balanced structure is obtained while wire-length is also reduced.

Top level algorithm: The top-level steps involved in our buffer insertion flow are given below:

1. Initialize a list F as an empty list. This list will contain all the *flagged, unmerged* nodes. A *flagged* node is one that cannot be merged with any of the other unmerged nodes without violating the limit on effective downstream capacitance (which is the maximum driving capability of the buffer used).
2. Initialize a list U with the set of all the sink nodes. This list will store all the *unmerged, unflagged* nodes.
3. While $(Sizeof(U) + Sizeof(F) > 1)$ Do
   (a) (T_i, T_j) = GetSubTreesToBeMerged(U) using steps in Figure (8).
   (b) If $(T_i, T_j) \neq$ NULL
      i. Merge the subtrees to get a new subtree T_k. Obtain the values of *required slew* and C_{effk} for node k using Figures (6) and (7).
      ii. Remove T_i, T_j from U.
      iii. Add T_k to list U.
   (c) else if ((T_i, T_j) = NULL) AND $(Sizeof(U) + Sizeof(F) > 1)$
      i. Insert buffers at all the nodes of F.
      ii. Update the values of delay, slew and effective downstream capacitance for all nodes $\in F$ using the delay characteristics of the buffer.
      iii. Move all the nodes in list F to list U and empty list F.
4. Perform top down embedding.

The key step in the above procedure is the step 3(a) which selects the node pairs to be merged. This step is explained in Figure (8). For node-pair selection, we use similar cost function as in [17] with an important change. In [17], a buffer will be inserted in a node *as and when* the node downstream capacitance exceeds a certain limit. But such an approach will result in an highly unbalanced clock tree.

In our algorithm, we insert buffers only when there is no node pair that can be merged without violating the *effective downstream capacitance limit*. To enforce this requirement, we maintain two separate lists - one called F which will have a list of $flagged$ nodes and another list called U in which we will store the list of $unflagged$ nodes. For node pair selection, we consider only the list U. If, for a particular node $i \in U$, we are not able to identify a suitable node pair for merger without exceeding the capacitance limit, we add that node to the list of *flagged* nodes F and remove i from U. We repeat the node-pair selection process until the list U becomes empty or contains a single element that cannot be merged with any other node. At that stage, we add buffers to *all the unmerged nodes* of F, update their delays, slews and effective downstream capacitances and transfer all the nodes to the list U. This cycle continues till there is only a single clock tree.

Procedure: $GetSubTreesToBeMerged(U)$
Input: Set of all unmerged subtrees
Output: The two subtrees to be merged

1. $PairsFound = 0$
2. While (PairsFound $\neq 1$) AND $(Sizeof(U) > 1)$ Do
   (a) T_i = subtree with min root-sink delay in U
   (b) $MergingCost = \infty$
   (c) For each subtree $T_k \in U$ and $T_k \neq T_i$
      i. $cost = MergingCost(T_i, T_k)$ defined in Fig. 9
      ii. if $cost < MergingCost$
         $MergingCost = cost; T_j = T_k$.
   (d) if $MergingCost \neq \infty$
      PairsFound = 1
      else
      Remove T_i from U; Add T_i to F.
3. if $MergingCost \neq \infty$
   return (T_i, T_j)
   else
   Transfer the possible single node $\in U$ to list F.
   return $NULL$.

Figure 8: The algorithm for selecting the subtrees to be merged.

Procedure: $MergingCost(T_i, T_j)$
Input: A pair of subtrees
Output: The merging cost of the subtree pair

1. Cost = Total wire length required to merge T_i and T_j
2. EDSC = Effective downstream capacitance of the parent node *assuming* the merging of subtrees T_i and T_j using steps of Figure (7)
3. If $EDSC <$ Capacitance Limit
   return Cost
   else
   return ∞

Figure 9: The Merging cost for two subtrees.

A point that may be noted here is that the $MergingCost$ algorithm of Figure (9) returns a value of ∞ when a possible merger of two node pairs i and j causes the effective capacitance limit to be violated. Thus, only node pairs that result in a node with lesser effective capacitance than the preset limit are merged.

Merits of our algorithm: An obvious advantage of the above procedure is that it will, by construction, result in a perfectly balanced clock tree. This is because buffers are added only in the step 3(c) of the top-leve algorithm in which *all* the unmerged nodes are buffered. As a result, the number of buffers from the clock source to *every* sink will be the same, thus satisfying one of the important objectives of our work.

A less obvious advantage of the proposed merging scheme is that, on the average, all the nodes to be *flagged* are mostly in the same ballpark as 1/2 times the effective capacitance limit used in the Figure (9). This results in similar equivalent capacitance loads for all the buffers at a given level. This helps to a great extent in reducing the actual SPICE skew. It may be noted here that works of [9, 14] also target the objective of balancing the loads for buffers at a given

level. However, they obtain the balancing by adding excessive wire capacitance, which results in a big increase in total wire-length. In our scheme, since we merge nodes considering the wire length cost, our algorithm generally results in considerably lesser wire-length that [9, 14]. Our top-down embedding after obtaining the topology is identical to the DME algorithm [8].

5. LINK INSERTION FLOW

We adopt the following approach to address the challenges discussed in section 2.2:

- The problem of inaccurate delay model and the chicken-egg relationship between link and buffer slew is addressed by using the iterative delay evaluation procedure of [19] and the bottom-up clock tree synthesis flow described in section 4.2.
- The problem of excessive nominal short-circuit power is addressed by considering both the spatial proximity of the nodes *and* the proximity in terms of delays during link node pair selection. This approach also makes sure that link addition does not affect the skew between other sink pairs adversely. More specifically, we use a modified cost function for the MST algorithm of [2] by making the link insertion cost as a weighted function of *both* link length and accurate delays (obtained using the algorithm of [19]) of the clock tree end points before link insertion.

The top-level algorithm of our link insertion for buffered clock tree is similar to the top-level algorithm for the unbuffered case with certain important differences like the use of accurate delay models and the use of link insertion friendly clock tree synthesis methodology. The major steps in constructing a linked buffered clock tree are:

1. Construct a balanced buffered clock tree using the flow described in Section 4.
2. Select the sink node pairs for link insertion using a modified form of algorithm in [2] with the cost function as weighted function of link length and proximity of accurate delays for the node pairs. The accurate endpoint delays are obtained using the algorithm of [19] for delay evaluation.
3. Since the endpoint locations are fixed, the capacitance value of each link can be calculated once the node pairs have been selected. Using the link capacitance values as extra load capacitance at the selected sinks, construct another buffered clock tree with the same topology as the one constructed in step(1). This new buffered clock tree will be equivalent to the first buffered clock tree *plus* the link capacitance.
4. Add the link resistances to the new clock tree built in step (3). The final result will be equivalent to the buffered clock tree constructed in the first step *plus* the link capacitance *and* link resistances. This is our final buffered, linked clock network.

It may be noted here that, even though the work of [3] has a similar objective of obtaining a link based buffered clock network, our approach differs from that of [3] in the following aspects:

- We use ordinary buffer cells unlike [3] which requires special tunable buffer cells.
- We use the iterative delay evaluation procedure of [19] during clock tree synthesis instead of SPICE. This makes our algorithm both fast and accurate.
- We propose a new merging scheme that results in a balanced buffered clock network that is inherently friendly for link insertion.

6. EXPERIMENTAL RESULTS

In order to verify the variation tolerance of our new buffered clock tree and the linked buffered clock tree approaches, we run SPICE based Monte Carlo simulations (500 trials) considering both interconnect and device variations. We assume that interconnect width, load capacitance, device channel lenght and oxide thickness vary with a Gaussian distribution with $\sigma = 5\%$. We implemented our algorithms in C++ and experiments were run with a 3.25GHz, 2Gb memory Linux system. We use the same benckmarks as in [7].

It will be apt to compare our results with the results of [3] because of near-identical objective of our work and the work of [3]. However, the exact details of the tunable buffers in [3] were unavailable to us for direct comparison. As a result, we compare our results with the algorithms in [9] and [17]. We chose these two algorithms for comparison because the algorithm in [17] will result in a clock tree with greatly reduced wire length consumption because it is very similar to minimum spanning tree construction. So it can be a good benchmark to do the wire-length comparisons. The algorithm in [9], due to its balanced nature, is likely to yield a good and balanced clock tree with reduced skew variability. Thus, comparing our results with these two algorithms will give us appropriate benchmarks for both wire-length and skew. It may be noted that, for the major part, the code for our algorithms and our implementation for [9, 17] are identical *except* the merging schemes used. So the difference in runtime and results can be directly attributed to the different merging schemes and delay model.

Since the results of [17] are expected to yield the minimum wire length and worst skew (because of its unbalanced clock trees), we use [17] as the baseline for comparing our results. The skew variation and resource consumption for [17] are shown in table 1. While selecting the clock trees for different algorithms, we made sure that all of them meet the slew requirement of 100ps on the clock tree points. As a result, the skew across benchmarks of different sizes are comparable for a given algorithm. We also made sure that the clock tree with minimal resources that met the slew criterion was selected for each algorithm so as to ensure a fair comparison between the different algorithms.

Table 2 shows the results of our new algorithms and the algorithm in [9] *scaled* in terms of the results of [17] (All the columns except the # Buf and CPU have been scaled). The Method column specifies the method for which results have been given. We have identified the algorithm of [9] and [17] as CTS- [9] and CTS- [17] respectively. We identify our algorithms as CTS and Link+CTS. The wire length consumption is shown under the column titled WL. The '# Buf' column gives the number of buffers for the particular clock tree. The NS, WCS and AS denote the 'Nominal Skew', 'Worst Case Skew' and 'Average Skew' in SPICE, the last two values obtained for 500 trials of Monte Carlo simulations in SPICE. The important observations from table (2) are as follows:

- From column 2 of table 2, it can be observed that our

buffered clock tree results in comparable wire-length to that of [17] and much less wire-length than [9].

- As expected, the skew values for [17] is the worst among all the algorithms. Also, it can be observed that our buffer insertion algorithm produces consistently better results than [9] in terms of skew variability reduction.
- The linked, buffered clock network has the best skew variability reduction among all the algorithms. Also, the percentage of extra wire-length consumed for link insertion is small and drops heavily as the size of the clock tree increases. This proves the effectiveness of link insertion for buffered clock trees.
- The CPU time consumed for the algorithm [17] is the lowest while our algorithms yields comparable CPU times. When compared to the run times of [9], the run times of our buffer insertion algorithm and the linked buffered clock network algorithm are much faster. Most notably, our runtimes are much lower compared to those reported in [3] because [3] uses SPICE.

TC	WL	# Buf	NS	WCS	AS	CPU(s)
r1	25937	16	100	190	76	0.06
r2	34110	28	96	222	60	0.36
r3	34353	36	101	196	52	0.71
r4	55115	78	176	362	76	3.46
r5	109722	163	110	226	56	9.4

Table 1: Skew variation and resource consumption results for the algorithm in [17]

TC	Method	WL	# Buf	NS	WCS	AS	CPU
r1	CTS-[17]	1.0	16	1.0	1.0	1.0	0.06
	CTS-[9]	5.2	18	0.57	0.72	0.46	1.1
	Our CTS	0.8	18	0.37	0.49	0.23	0.08
	Link+CTS	1.1	18	0.41	0.45	0.16	0.18
r2	CTS-[17]	1.0	28	1.0	1.0	1.0	0.36
	CTS-[9]	7.5	36	0.91	0.95	0.91	14
	Our CTS	1.8	40	0.62	0.60	0.59	0.42
	Link+CTS	1.8	40	0.65	0.39	0.37	0.52
r3	CTS-[17]	1.0	36	1.0	1.0	1.0	0.71
	CTS-[9]	9.6	41	0.59	0.57	0.61	44
	Our CTS	1.1	45	0.49	0.54	0.51	0.78
	Link+CTS	1.3	45	0.51	0.40	0.32	0.88
r4	CTS-[17]	1.0	78	1.0	1.0	1.0	3.46
	CTS-[9]	12.4	85	0.56	0.55	0.47	509
	Our CTS	2.1	83	0.34	0.42	0.36	3.94
	Link+CTS	2.1	83	0.41	0.33	0.25	4.41
r5	CTS-[17]	1.0	163	1.0	1.0	1.0	9.4
	CTS-[9]	9.1	174	0.79	0.55	0.49	2009
	Our CTS	1.5	183	0.46	0.38	0.35	10.12
	Link+CTS	1.5	183	0.48	0.30	0.28	11.62

Table 2: Skew variation and resource consumption results for our new algorithms and algorithms in [9] w.r.t. results of [17] in Table 1

7. CONCLUSIONS

A complete link based buffered clock network synthesis methodology using accurate and efficient delay model has been proposed. When compared to existing algorithms, there is 50% reduction in skew variability on average. The buffer/wire length cost and CPU time is also significantly less than most of the previous algorithms. Moreover, our methodology does not require any special tunable buffered cells and does not require SPICE simulation for clock tree synthesis, unlike [3], making our method general, fast and efficient.

8. REFERENCES

[1] A. Rajaram, J. Hu, and R. Mahapatra, "Reducing clock skew variability via cross links," in *Proceedings of the ACM/IEEE DAC*, San Diego, CA, June 2004, pages 18–23.

[2] A. Rajaram, D. Z. Pan, and J. Hu, "Improved Algorithms for Link-Based Non-Tree Clock Networks for Skew Variability Reduction," in *Proceedings of the ISPD*, San Francisco, CA, April 2005, pages 55–62.

[3] G. Venkataraman, N. Jayakumar, J. Hu, P. Li, S. Khatri, A. Rajaram, P. McGuinness, and C. Alpert, "Practical Techniques for Minimizing Skew and Its Variation in Buffered Clock Networks," in *Proc. of the ICCAD*, San Jose, CA, pages 592-596, November 2005.

[4] E. G. Friedman, "Clock distribution networks in synchronous digital integrated circuits," in *Proceedings of the IEEE*, vol. 89, no.5, pp.665–692, May 2001.

[5] R. Saleh, S. Z. Hussain, S. Rochel, and D. Overhauser, "Clock skew verification in the presence of IR-drop in the power distribution network," in *IEEE Transactions on CAD*, vol.19, no.6, pp.635–644, June 2000.

[6] W.-C. D. Lam, C.-K. Koh, and C.-W. A. Tsao, "Power supply noise suppression via clock skew scheduling," in *Proceedings of the IEEE ISQED*, San Jose, CA, March 2002, pp. 355–360.

[7] R.-S. Tsay, "Exact zero skew," in *Proceedings of the IEEE/ACM ICCAD*, Santa Clara, CA, November 1991, pp. 336–339.

[8] T.-H. Chao, Y.-C. Hsu, J.-M. Ho, K. D. Boese, and A. B. Kahng, "Zero skew clock routing with minimum wire-length," in *IEEE Transactions on CS-ADSP*, vol.39, no.11, pp.799–814, November 1992.

[9] Y. P. Chen, and D.F. Wong, "An algorithm for zero-skew clock tree routing with buffer insertion," in *Proceedings of the ED & TC*, Pairs, France, March 1996, pp. 230–236.

[10] S. Pullela, N. Menezes, and L. T. Pillage, "Low power IC clock tree design," in *Proceedings of the CICC*, May 1995, pp.263–266.

[11] J. Chung and C.K. Cheng, "Optimal Buffered Clock Tree Synthesis," in *IEEE ASIC conference*, Austin, TX, Sept. 1994, pp. 130–133.

[12] A. Vittal, and M. Marek-Sadowska, "Low-power buffered clock tree design," in *IEEE Transactions on CAD*, vol. 16, no. 9, pp. 965 - 975 , Sept. 1997.

[13] G. E. Tellez, and M. Sarrafzadeh, "Minimal buffer insertion in clock trees with skew and slew rate constraints" in *IEEE Transactions of CAD*, vol. 16, no.4, pp.333–342, April 1997.

[14] A. D. Mehta, Y. P. Chen, N. Menezes, D. F. Wong, and L. T. Pilegg, "Clustering and load balancing for buffered clock tree synthesis" in *Proceedings of the ICCD*, Austin, Tx, October 1997, pp. 217–223.

[15] J. Tai. Yan, C. W. Wu, K. P. Lin, Y. C. Lee, and T. Y. Wang, "Iterative convergence of optimal wire sizing and available buffer insertion for zero-skew clock tree optimization" in *Proceedings of Asia-Pacific Conference*, December 2004, pp.529–532.

[16] J. L. Tsai, T. H. Chen, and C. C. P. Chen, "Zero skew clock-tree optimization with buffer insertion/sizing and wire sizing" in *IEEE Transactions of CAD*, vol. 23, no. 4, pp.565 - 572, April 2004.

[17] R. Chaturvedi, and J. Hu, "Buffered clock tree for high quality IC design" in *Proceedings of the ISQED*, March 2004. pp. 381–386.

[18] K. Wang, and M. Marek-Sadowska, "Clock network sizing via sequential linear programming with time-domain analysis" in *Proceedings of the ISPD*, Monterey, CA, April 2003, pp. 182–189.

[19] R. Puri , D. S. Kung, and A. D. Drumm, "Fast and accurate wire delay estimation for physical synthesis of large ASICs" in *Proceedings of the GLSVLSI*, New York, NY, April 2002, pp. 30-36.

[20] S. R. Nassif, "Modeling and analysis of manufacturing variations," in *Proceedings of the IEEE CICC*, San Diego, CA, May 2001, pp. 223–228.