An Novel Performance-Driven Topology Design Algorithm

ABSTRACT

1. INTRODUCTION

As technology advances into the nanometer regime, interconnect delay has become the dominant factor in circuit delay. Therefore, effective performance-driven interconnect design has become crucial for the timing closure. The main techniques for performance-driven interconnect design are topology design, buffer insertion, device sizing and wire sizing. Given a good topology, there are many efficient algorithms for buffer insertion and wire sizing. However, topology design in itself is a very hard and time-consuming step. For nets with low degree[1] such as 2-pin or 3-pin nets, it is easy to find the interconnect tree. But for high-degree nets, constructing a good topology efficiently is very challenging. However, finding good topologies for these nets are very critical because they are more likely to be in a critical path because they are inherently slow.

Rectilinear minimum spanning tree (RMST) is a class of topologies widely used in practice. It has efficient algorithms to get the optimal solution. However, the wirelength of RMST can be as much as 1.5 times that of Rectilinear Steiner minimal tree (RSMT) [1]. Therefore RSMT is another class of well researched topologies. But since RSMT is a NP-complete problem [2], no efficient algorithm exists for optimal solution. For optimal RSMT algorithm, the fastest implementation is currently the GeoSteiner package [3, 4]. A lot of work focused on efficient approximation algorithms on constructing RSMT. Batched 1-Steiner heuristic [5] and the heuristic by Mandoiu et. al. [6] are two well-known near-optimal algorithms. Recently, FLUTE [7, 8] has been proposed as a very fast and accurate RSMT algorithm aiming at VLSI applications based on table lookup.

In addition to wirelength-driven topologies such as RMST and RSMT, many performance-driven topology design techniques have also been proposed. The SERT algorithm of Boese et. al. [11] produces the routing tree for performance. Later, Cong et. al. [12] proposed A-tree algorithm to find a min-area shortest paths tree. In [13], Permutation-constrained routing trees (P-tree) algorithm reported better area objectives than SERT and A-tree. Alpert et al. [14] proposed AHHK trees as a direct trade off between Prim's MST algorithm and Dijkstra's shortest path tree algorithm. It has been used in C-tree algorithm [15] for timing-driven Steiner tree construction. However, all these algorithms are not very efficient so that using them to find topologies for a huge number of high-degree nets becomes very expensive.

[1]The *degree* of a net is the number of pins in the net.

Although most of the nets in a design are with low degree, there are still a significant amount of high-degree nets (12% nets have degree ≥ 8 [7]). Hence, our goal is to develop a fast performance-driven topology design algorithm so that we can apply it to a large amount of nets to achieve good timing property.

In this paper, we present a novel method to design the performance-driven topology for nets efficiently. We first analyze the possibility of extracting common properties of the nets for performance-driven topology design. Then we choose the A-tree as the basic topology and develop a very fast algorithm to construct A-tree based on table lookup and net-breaking. Finally we apply postprocessing techniques on the obtained A-tree to further improve the timing for the net.

Our main contributions include the following:

- A fast algorithm to construct the A-tree *potentially optimal wirelength vector* (POWV) [7] and topology table for all the nets with degree less than a given value.
- A fast algorithm to construct A-tree for any nets using table lookup and net-breaking techniques.
- A performance-driven postprocessing technique, which is not stick to the A-tree topology, to further improve the timing for the net.

Experimental results show that our new algorithm can get very good topologies for the nets in terms of performance. Moreover, for the high-degree nets, our algorithm can be ?00$\times$ faster than the timing-driven tree construction algorithm in C-tree. Therefore, it is very suitable in performance-driven topology design for a large number of nets.

The remainder of the paper is organized as follows. In Section 2, a brief review of the FLUTE [7, 8] is provided for better understanding the notion and idea in this paper. Section 3 describes the fast algorithm to generate A-tree lookup table. We present the algorithm to construct A-tree using table lookup in Section 4, and a performance-driven postprocessing technique in Section 5. In Section 6, experimental results are shown.

2. PRELIMINARY

In this section, we review the basic notions used in FLUTE and discuss the possibility to handle the performance-driven topology design using table lookup idea.

The table lookup idea in FLUTE [7, 8] makes the RSMT construction very efficient. By using vertical sequences to represent pin configurations, it can represent infinite number of degree-n nets with n! groups. For each group, the wirelength of all possibly optimal routing topologies along the Hanan grid [16] can be written as a small number of linear combinations of distances between adjacent Hanan grid lines. Each linear combination can be expressed as a vector of the coefficients which is called a *potentially optimal wirelength vector* (POWV). The few POWVs for each group can be generated once by an efficient algorithm based on boundary compaction technique. Each POWV and one corresponding topology are stored into a lookup table. To get the RSMT for a net, the algorithm just compute the wirelengths corresponding to the POWVs for the group the net belongs to. Then pick the one with the best wirelength. Therefore, for the low-degree nets, RSMT can be directly found in the table. However, since the lookup table will be impractically large for high-degree nets, Hence, the high-degree nets are recursively divided into sub-nets by net-breaking techniques so that the lookup table can be used.

The main notion we want to recall here are *vertical sequence* and *boundary compaction*. *vertical sequence* is used to group the nets. Consider an n-pin net. Let x_i be the x-coordinate of some vertical Hanan grid line such that $x_1 \leq x_2 \leq ... \leq x_n$. Similarly, let y_j be the y-coordinate of some horizontal Hanan grid line such that $y_1 \leq y_2 \leq ... \leq y_n$. Assume the pins are indexed in ascending order of y-coordinate. Let s_i be the rank of pin i if all pins are sorted in ascending order of x-coordinate. $s_1 s_2 ... s_n$ is call *vertical sequence*. In fact, *vertical sequence* defines the relative positions for the pins in the net. All the nets with the same *vertical sequence* fall in one group in the lookup table. The notations are illustrated in Figure 1.

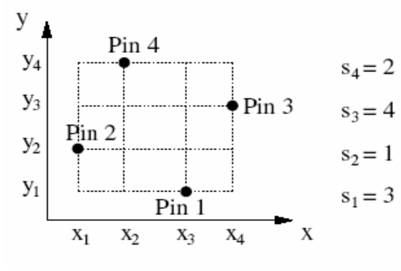

Figure 1: Illustration of some notions.

In addition, we also briefly restate the *boundary compaction* technique because it is a major technique used in generating the A-tree lookup table. For a given group (i.e. *vertical sequence*), the *boundary compaction* technique reduces the grid size by compacting any one of the four boundaries, i.e., shifting all pins on a boundary to the grid line adjacent to that boundary. The set of routing topologies of the original problem can be generated by expanding the routing topologies of the reduced grid back to the original grid. Figure 2 uses the compaction of left boundary to illustrate the idea.

FLUTE provides a very efficient way to generate the RSMT. However, RSMT is not suitable for performance-driven topology design. In RSMT, a lot of detours could be generated from source to sinks, this may result in bad timing results for some sinks despite its good wirelength. A simple example is shown in Figure 3. Sink $t4$ is the critical sink here. We can see that there is detour from the source s to the critical sink which harms the timing result. Therefore, we need some other types of toplogies for our purpose. However, we still want to borrow the idea of table lookup idea of FLUTE because of its great efficiency. So we need to extract the common property of infinite number of nets to construct the lookup table, what we need is some general topology other than the topologies dependent on net specific parameters.

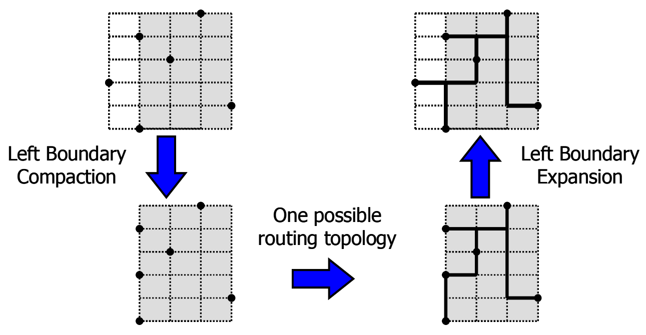

Figure 2: Boundary Compaction.

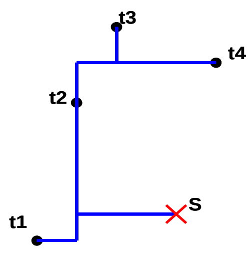

Figure 3: Detour in RSMT.

A-tree has some good properties for performance-driven interconnect design. First, any A-tree is a shortest path tree (SPT). Thus, we will not have the concern about detour from source to any sink. In addition, it has been shown in [12] that minimizing total wirelength of an A-tree lead to simultaneous optimization of different components of sink delays. Such a harmony would be impossible to achieve for general routing topologies. However, finding A-tree with minimum wirelength is also NP-complete problem [10]. Therefore, several heuristics [9, 12] have been proposed. Hence, we choose A-tree as our basic topology and try to find A-tree using table lookup technique. We will discuss the details in the following parts.

3. A-TREE LOOKUP TABLE FOR LOW - DEGREE NETS

In this part, we follow the notions used in FLUTE [7, 8]. It is shown that the set of all degree n nets can be partitioned into n! groups according to their *vertical sequence*. For each group, a few POWVs and their corresponding topologies are stored in the lookup table.

In our work, we also use the *vertical sequence* to group the net, which is the same as in FLUTE. However, It is not enough to just have *vertical sequence* for A-tree and the source pin also needs to be specified. That is to say, not only the relative positions of pins but also the source pin position are used to define which group the net falls in. Given a group and the source pin, we observe that A-trees can be obtained by applying boundary compactions on a net if we never compact a boundary with the source on it. Hence, we can generate the A-trees and their corresponding wirelength vectors very efficiently. Then we prune the redundant ones to get the POWVs with corresponding A-tree topologies. Note the POWV in our work is a concept different from that in FLUTE. In FLUTE, POWV is defined as potentially optimal wirelength vector that can produce the optimal wirelength for a rectilinear Steiner tree. Here in our paper, it is the potentially optimal wirelength vector that can produce the optimal wirelength for an A-tree. Although the definition of POWV is slightly different, the way to compute it and all the operations on it are the same.

In FLUTE, each group has a set of corresponding POWVs. But in our case, each group *vertical sequene* are divided into d (net degree) subgroups. For those subgroups 1 to d, the corresponding source pin is pin 1 to pin d. For each subgroup, there are a set of POWVs correspondingly. Another difference is that in FLUTE, only one arbitrary topology is generated and stored for a POWV because the wirelength is the objective. In our case, since we want to explore different topologies for good performance, we want to save all possible topologies corresponding to each POWV. Therefore, we need to look at topologies as well as WVs (wirelength vectors) when generating the table. In this sense, constructing the A-tree table is more complicated than constructing FLUTE tables. In order to generate the A-tree table efficiently, we propose another algorithm other than the algorithm Gen-WVs(G) in FLUTE [7].

A basic observation is that we generate every A-tree by a sequence of compactions. We define the *compacting sequence* as the sequence of compaction operations that compacts the original pin configuration into a single node. Hence, one *compacting sequence* corresponds to one A-tree topology. A direct idea for finding different topologies is looking at the different compacting sequence. Unfortunately, the number of *compacting sequences* is huge. For one particular group, the number of different sequences = $4^{2\times(d-1)}$, where d is the net degree. If d=9, # sequences = 4^{16}! Actually, among these sequences, a lot of them are not feasible because we have to perform $d-1$ times of horizontal compactions (left or right) and $d-1$ times of vertical compactions (top or bottom). Therefore, the number of feasible sequences for one 9-pin net = $\binom{16}{8} \times 2^8 \times 2^8$ =843448320. And this is just for one group, the total # sequences for all 9-pin nets is 9! times this number.

Although the number of *compacting sequences* is huge, we still have hope because we only want to save the different topologies that can result in POWVs. Therefore, most of the *compacting sequences* can be pruned. But since there are so many sequences, directly generating all sequences and prune them is not practical at all. Our idea to generate and prune the sequences is using a graph called *Configuration Graph* (CG). The major advantage of this method is that it enable the pruning as early as possible which saves significant amount of computation time and memory space.

First, we define some terms. A *pin configuration* is the configuration of a set of pins on the Hanan grid. If we apply a sequence of boundary compactions on a specific *pin configuration*, we will get another *pin configuration.* The new *pin configuration* can have the same or less pins than the original because some pins may collapse together. In *Configuration Graph*, every node corresponds to a *pin configuration.* So we call these nodes *Configuration Nodes* (CN). There are two kinds of special nodes in the *Configuration Graph.* One is the CN corresponding to the original pin configuration without any boundary compaction. We call it Start Node because any boundary compaction operation starts with it. The other type is the CN with the *pin configuration* as a single point. We call them End Nodes because any *compacting sequence* will end with this CN. Note that an A-tree topology is obtained when reaching an End Node. A *Partial wirelength Vector* (PWV) is the WV with undecided entries obtained after a sequence of compactions. For example, if a full WV is 12211121, a PWV could be $1xx111x1$ (x means undecided). The undecided part corresponds to the horizontal edges or vertical edges that have not been created by boundary compaction. For each CN, a set of PWVs are also associated with it. They are the PWVs created by boundary compactions that create the *pin configuration* associated with the CN. If compacting the *pin configuration* associated with a CN can get the *pin configuration* of another CN, an edge is created from the first CN to the second and the direction of compaction (left, right, top or bottom) is associated with the edge. An example of *Configuration Graph* is shown in figure 4.

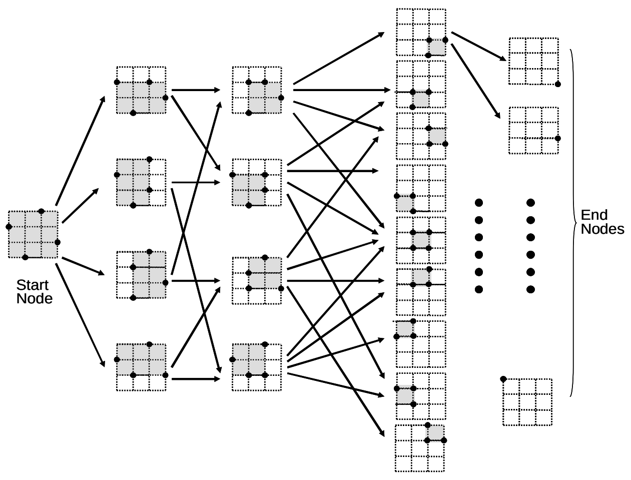

Figure 4: Configuration Graph.

LEMMA 1. *The bounding box of a* pin configuration *in original pin Hanan grid defines the* pin configuration.

In other words, after a sequence of compaction, if the original *pin configuration* is transformed into a new *pin configuration* with bounding box, the new *pin configuration* is independent on the compactions performed.

Proof idea: As shown in Figure 5, the whole grid is the Hanan grid of original pin configuration and the center grey part is the new configuration with bounding box B obtained by some *compacting sequence* Q. It is easy to see that all the pins in the four corner region (1) are compacted to the four corners in the new configuration. And the four boundary regions (2) are compacted to the closest boundary with the unique position. The center region (3) is not changed. So every pin has the unique pin position in the compacted configuration. No matter what the *compacting sequence* Q is, every pin has the same position in this configuration.

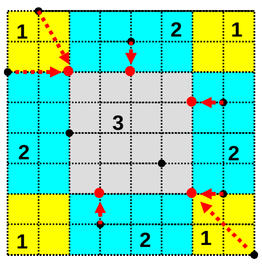

Figure 5: Lemma 1 Proof.

From Lemma1, we know the number of nodes and edges are small numbers of CG is small. It is just the different number of different bounding box we can find in the Hanan grid. The Configuration Graph can save a lot of memory space and runtime while doing the pruning. # CN $= \sum_{i=1}^{d}\sum_{i=1}^{d}(d+1-i)(d+1-j)$, if $d = 9, \#CN = 2025$.

LEMMA 2. *If a PWV at a CN is worse than the other, it cannot be part of any POWV (it can be pruned).*

PROOF. Prove by contradiction, assume a PWV V_1 at a CN is worse than the other V_2, but it is part of a POWV V. From Lemma1 we know that the undecided part of WV is the same for V_1 and V_2 because of the same *pin configuration*. Let V_b=$V - V_1$. Then $V_2 + V_b$ is better than $V_1 + V_b$. A contradiction with V is a POWV. □

From Lemma2, we can use the CG to prune the *compacting sequences* efficiently. We can do pruning at each node with "PWV dominance". We say a PWV is dominated by the other one if it corresponds to more wirelength, i.e., it has the same or bigger value on all entries in WV. This kind of pruning does not wait until the full WV is obtained. It can prune the bad WV as early as possible and accelerates the pruning process a lot.

In order to further reduce the complexity of *Configuration Graph*, we start from a new Start Node that is obtained by compacting the original *pin configuration* once in all 4 directions (left, right, top, and bottom). We have the following lemma about this new Start Node.

LEMMA 3. *No POWV will be lost by start compacting at the new start node.*

The proof is similar to the Lemma2 in [6].

Since we begin from this new Start Node with the size $(d-2)\times(d-2)$ *pin configuration*, the length of *compacting sequences* is reduced from $2(d-1)$ to $2(d-3)$. And # CN can be reduced to $\sum_{i=1}^{d-2}\sum_{i=1}^{d-2}(d-1-i)(d-1-j)$, if d=9, # config = 784.

Our algorithm to generate the *Configuration Graph* is shown in figure 6.

In step 1, we do the left, right, top, and bottom boundary compaction on original *pin configuration* to create the start

```
Input: G is a grid with some pins at grid nodes
Output: Configuration Graph H
1. Perform left, right, top and bottom boundary compaction sequentially on G
   and get the new grid G'.
2. Create the start node N0 for H and make G' as its pin configuration and
   PWV0 = 1xx...x11xx...x1 as its PWV.
3. Gen-Config(N0, N0, PWV0)

Gen-ConfigNode(Px, Nx, PWVx)
1. Add PWVx to the current list of PWV associated with Nx, prune the
   redundant PWVs in the list with "PWV dominance". If Nx≠N0 and PWVx is
   not pruned, create a edge from Nx to Px.
2. if the pin configuration of Nx is a single node
3.    return;
4. else
5.    Nxl = compact-left(Nx)
6.    PWVxl = PWVx + (PWV caused by left boundary compaction)
7.    Gen-ConfigNode(Nx, Nxl, PWVxl)
8.    compact-right(Nx)
9.    PWVxr = PWVx + (PWV caused by right boundary compaction)
10.   Gen-ConfigNode(Nx, Nxr, PWVxr)
11.   compact-top(Nx)
12.   PWVxt = PWVx + (PWV caused by top boundary compaction)
13.   Gen-ConfigNode(Nx, Nxt, PWVxt)
14.   compact-bottom(Nx)
15.   PWVxb = PWVx + (PWV caused by bottom boundary compaction)
16.   Gen-ConfigNode(Nx, Nxb, PWVxb)
```

Figure 6: Algorithm for generating Configuration Graph.

pin configuration for the CG. From Lemma3, it is safe to start with this configuration to construct the CG.

In step 2, we create the start node N0 with *pin configuration* obtained in step1 and set up the initial PWV as $1xx...x11xx...x1$ because we have four edges left, right, top and bottom due to the first four boundary compaction.

In step 3, we recursively generate new nodes for *Configuration Graph* using boundary compaction technique. And for each CN, we will prune the redundant PWVs associated with it and only keep the best PWVs. From Lemma2 we know this kind of pruning is safe. The recursive compaction ends at end nodes (where no compaction can be applied and the full WV and topology are obtained).

After we get the *Configuration Graph*, we can use it to find the A-tree topologies efficiently. First, it is obvious that any path from the start node to an end node corresponds to a compacting sequence and thus a tree topology. But since our goal is to find A-trees with the source as specific pins, we need to specify the source and find the A-trees corresponding to the source. We have the following lemma for the trees generated by our algorithm.

LEMMA 4. *Any tree topology generated by a* compacting sequence *associated with a path from the start node to an end node is an A-tree with the source at the position corresponding to the end node.*

It is easy to see that any boundary compaction will generate edges toward the source in the Hanan grid.

Therefore, for any pin as the source, we can easily find the POWVs for the A-trees. We just look at the CN corresponding to the position of that pin and all the POWVs associate with that CN are the POWVs for A-trees source at the pin. Since every pin can be the source of A-tree, we need to save all the POWVs for A-trees source at each pin. Hence, we categorize these POWVs by the pins. When we construct topology for a specific net with a given source, we only need to look at the POWVs corresponding to the source pin in the table. We can get POWVs for every pin as the source simultaneously once we have the *Configuration Graph* instead of generating POWVs for A-trees source at each pin separately. This is another advantage of the *Configuration Graph* approach.

Although we find the POWVs for A-trees using *Configuration Graph*, we have not finished our work for finding the A-tree topologies corresponding to these POWVs. The reason is that many compacting sequences can result in the same POWV, but they can give different topologies. Therefore, we want to find all different topologies corresponding to every POWV and save them in table.

We study the topologies generated by different *compacting sequences* corresponding to POWVs and find that most of them are redundant. There are two kinds of redundancy. First, different compacting sequences generate the same topology. Second, different *compacting sequences* generate "equivalent" topologies in terms of both wirelength and timing. Two topologies are equivalent when they are the same in all node positions (pins and Steiner nodes) in Hanan grid and the connections between nodes. The only difference between equivalent topologies is the routes between the node pairs. To eliminate these two types of redundancy, we introduce the concept of "abstract topology". An abstract topology for a net is the topology on the Hanan grid that fixes the positions for all the nodes (pins and Steiner nodes) and the connections between these nodes. The difference between an abstract topology and a normal topology on the Hanan grid is that the abstract topology does not specify how the connection is embedded in Hanan grid. If two *compacting sequences* generate the same topology or equivalent topologies, their corresponding abstract topology is the same. Therefore, we only need to record the different abstract topologies for POWVs. Figure 7 illustrate the concept of abstract topology for a 6-pin net. Although the concept of abstract topology is very simple, it can save huge amount of table space. For example, consider a 9-pin abstract topology with 7 steiner nodes, 15 two-pin nets in the tree. If there are 2 different routing for 10 two-pin nets in 15, # embedded topologies = 2^{10} = 1024. If we just directly save the different topologies, we may need thousands of times space than just saving abstract topologies.

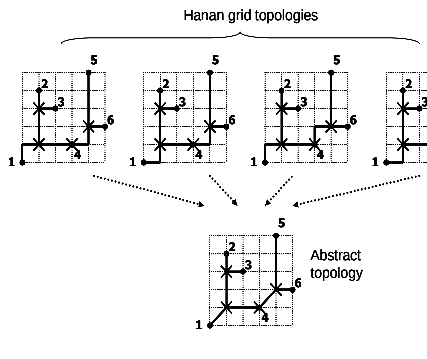

Figure 7: Abstract topology

Now the way we find different abstract topologies is as following. We start from the end node corresponding to the pins and for every POWV, trace back in the reverse direction of edges until reaching the start node. Hence the *compacting sequence* is found. Since the *compacting sequence* gives us the topology of the tree, we can get the abstract topology from the topology. If the abstract topology is different from the others, we keep it in the table corresponding to that POWV. Otherwise, simply discard it. After the whole process, we get all the different abstract topologies for every POWV corresponding to every pin position.

However, there is still a problem in generating and comparing the abstract topologies. To know whether a topology is redundant, we need to first find the abstract topology for the topology and compare it to all the abstract topologies found. This tree generation and comparison take a lot of runtime. Therefore, we want to make it easier and faster. The way we do this is using Topology Signature. A Topology Signature of a topology (for a given *pin configuration*) is the positions of the Steiner nodes in the topology. The following theorem gives us a better way to comparing two abstract topologies.

THEOREM 1. *Theorem1: For A-trees generated by our algorithm, two trees A and B has the same topology signature* $\leftrightarrow$ *A and B has the same abstract topology.*

Proof sketch:

$\leftarrow$

Obviously, from the definition, the abstract topology decides the Steiner node positions. One abstract topology corresponds to only one signature (Steiner node positions).

$\rightarrow$

LEMMA 5. *In A-trees generated by our algorithm, if fixing the positions for all the nodes (pins and Steiner node), every node v has a unique path to the source s. (The path is an ordered sequence of nodes in the tree with v as the first and s as the last.)*

This lemma can be proved by studying the way to connect a tree node to the next along its path to the source in the tree.

From Lemma5, we know topology signature (Steiner node positions) defines the connections of the tree. Hence, it defines the abstract topology (node positions and connections).

Theorem1 tells us that abstract topology and topology signature has one-to-one correspondence. Topology signature is really the "signature" for topologies. Therefore, instead of finding all different abstract topologies, we only need to find the topologies with different topology signatures. For the topologies generated by different *compacting sequences*, it is enough to compare their positions of Steiner nodes. After we find all the topologies with different topology signatures, we save their corresponding abstract topologies in the table. There are two advantages to save abstract topologies instead of topologies. First, the abstract topology needs less memory than a topology embedded in Hanan grid. Second, there is flexibility when embedding all the two-pin nets in abstract topologies.

Since we can find POWVs for A-trees from CG and employ the topology signature idea to find different abstract topologies for them, we save both the POWVs and the corresponding abstract topologies in the table.

Table 1 gives the statistics of our POWV table and topology table for nets up to degree 9. We also observed in experiments that all POWVs for net with degree up to 9 have

Degree n	# groups n!	# POWVs in a group		totoal # abstract topologies
		Max.	Ave.	
4	24	8	6	12
5	120	18	12.625	101
6	720	36	25.306	911
7	5040	70	50.685	11252
8	40320	144	99.482	162952
9	362880	282	193.189	2694985

Table 1: Statistics for POWV and topology table.

only one topology signature. Thus, we have the following conjecture.

Conjecture1: For A-trees generated by boundary compaction, any POWV has only one topology signature. That means one POWV corresponds to only one abstract topology.

Another stronger conjecture we have is that the topology signature is also applicable for all potentially optimal A-trees and potentially optimal Rectilinear Steiner trees on the Hanan grid.

4. A-TREE TOPOLOGY CONSTRUCTION AND NET-BREAKING

In last section, we construct the POWV table and corresponding A-tree topology table for the nets up to some degree D. Therefore, for any net with degree no more than D, we can find the corresponding group index based on the *vertical sequence* of the net. Having the group index and source pin, we directly look up the POWV table to find the corresponding POWVs and compute their wirelength based on the real geometric information of the net. Then we pick the POWV with best wirelength and look up the topology table for the A-tree topology corresponding to it.

Similar to FLUTE, it is not practical to generate table for high-degree nets because of the huge table size and extremely long runtime. Therefore, a high-degree net will be divided into several sub-nets with degree less than D to which the table lookup can be applied.

The net-breaking method we use is similar to that in [8]. However, since we are generating A-tree instead of RSMT, different heuristics need to be applied and the source needs to be considered when breaking a net.

We can still use the optimal net-breaking algorithm proposed in [8] because it will not affect the A-tree property. However, after the breaking, only one sub-net has the source in it. For the other sub-net, we need to specify a source pin. It is very simple in this case that we make the extra pin introduced in both sub-nets as the source for the sub-net without the original source in it. If we construct A-trees for both sub-nets with the sources as the original source and new source respectively, the combined tree is still an A-tree.

If there is no optimal breaking for a net, we will break the net in x or y direction. However, we cannot directly break the net at some pin and combine the two trees for the two sub-nets to obtain the whole tree as in [8] because it will not result in an A-tree. Therefore, with the A-tree constraint, instead of including a pin in the original net in both sub-nets, we introduce an extra pin to the net and include it in both sub-nets. This extra pin will become the source of the subset which does not have the original source in it. The position of this extra pin is found by a "source propagation" technique. Assume we already find the breaking direction and position, the source propagation is the operation to project the source on the breaking line. The projection of the source is the extra pin introduced, note as "propagated source". Figure 8 illustrates the "propagated source" idea.

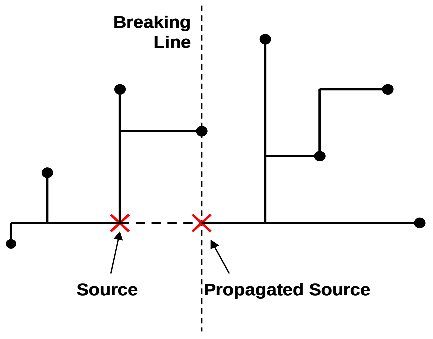

Figure 8: Source propagation.

Lemma 6. *Lemma, breaking a net at the "propagated source" will generate an A-tree by combining the A-trees of both sub-nets. (The sources of two sub-nets are the source of original net and the propagated source).*

Proof. Let N, N_1 and N_2 be the original net and two sub-nets after breaking and let S, S_2 be the source of N and the extra pin. Without loss of generality, we assume S is in N_1 after breaking. If T_1 is an A-tree for N_1 with source S and T_2 is an A-tree with source S_2, all the nodes in N_1 have the shortest path to S and all the nodes in N_2 have the shortest path to S_2. Since S_2 is in N_1, S_2 has the shortest path to S (which is a straight line). It is obvious that all the nodes in N_2 has the shortest path to S. Therefore, A_1+A_2 is an A-tree for net N. $\square$

However, this "source propagation" technique may not give a good position for the "propagated source" in some cases. For example, in figure 8, assume all the pins in the right subnet have bigger y-coordinates than the source. If we put the extra pin at the original position, it could lead to extra wirelength. In order to solve this problem, we slide the "propagated source" along the breaking line until it has the same y-coordinate as the pin with the smallest y-coordinate in the right subnet. It is easy to see that this operation will not affect the A-tree property of the whole tree. Apparently, this idea can be used no matter the net is broken in what direction and the source is in which subnet.

For the heuristics of choosing a good direction and position to break the net, we use the similar heuristics as in [8]. We make modifications on the original heuristics computing the breaking score so that it adapts to the new breaking method for A-tree.

After the A-tree of the whole net has been obtained by merging all the subtrees, we apply a simple heuristic to repair the errors caused by the nonoptimality of the table and net-breaking. For each node in the tree (on Hanan grid), we try to connect it to the closest point in the tree and in the direction of the source. This will further improve the wirelength and still maintain the A-tree property.

5. PERFORMANCE-DRIVEN POSTPROCESSING

So far, we can construct an good A-tree for any given net by net-breaking and table lookup. However, the topology we got is still a generic A-tree without consideration of the timing properties of the specific net, such as load capacitance and required time. In general, A-tree is a good topology in performance-driven routing when treating all the sinks as the same criticality. However, for a specific net, different sinks have different capacitive load, required time and distance to the source. This makes the problem more complicated to find a good topology in terms of performance.

Since we already have the A-tree as a good initial topology, it will be very convenient if we can make improvement on the obtained A-tree to achieve better timing. In addition, we are aiming at some very efficient heuristics so that it can match with our fast table lookup based A-tree construction technique. Therefore, we propose a performance-driven post-processing heuristic to modify the tree topology for better timing result.

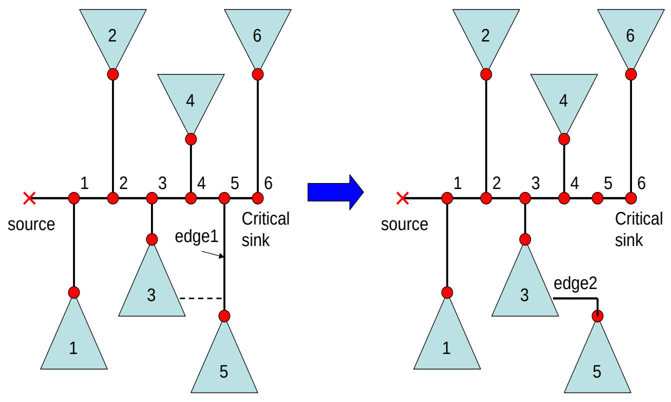

Figure 9: Branch Moving.

Our heuristic is called "branch moving", which change the tapping point for some branches in the tree. At this stage, we no longer restrict the topology to be an A-tree. The basic idea is to change the load distribution on the critical path to cut down the delay on critical sinks. To easily understand the technique, let us look at a simple example. As illustrated in figure 9, we have a tree topology (left) and know the critical sink by timing analysis. Now we want to look at the possibility to improve the timing for the critical sink. We first label the tree nodes on the critical path with numbers. These numbers represent the distances from the source to the nodes. The bigger the label on the node, the farther to the source. We employ Elmore delay model for our delay computation. Therefore, the delay on the critical sink is the sum of a series of RC terms, $Delay(Critical\ sink) = \sum_{i=1}^{6} R_i C_i$, where R_i is the path resistance from source to node i, C_i is the download capacitance of the subtree rooted at node i (excluding the critical path). If we change the tapping point of some branch, the delay on the critical sink will be changed too. Hence, we try to move the branches so that the delay on the critical sink is reduced. For instance, in figure 9 (left), we find a possible edge (dashed line) which moves the branch tapped to node 5 to the subtree tapped to node 3. It is easy to get the change on the critical sink. $\Delta d = R_3(C_5 + C_{edge2}) - R_5(C_5 + C_{edge1})$. Therefore, we can quickly find the delay change on the critical sink when we move a branch.

In fact, this "branch moving" technique is very flexible. You can find an edge (not existing in the original tree) between any two node on the tree and try to connect them. This operation will form a loop. In order to maintain the tree topology, we can break a edge in the formed loop to obtain a new tree. However, there are too many choices for the edge to be connected and broken. In our implementation, we constrain all the edges in the tree on the Hanan grid. Therefore, We find all the edges on the Hanan grid which is not in the tree. Then we measure the "benefit" to connect any of the candidate edges and break another edge in the formed loop. Here, the "benefit" is the delay reduction on the critical sink. Among all the candidates, we simply pick the one with best "benefit" and update the tree. We apply this "branch moving" iteratively until no improvement.

So far, we have introduced the "branch moving" technique to improve the critical sink delay. However, there are several problems that need careful consideration. First, we should not touch the nodes on the critical path. Otherwise, we will create detour from source to the critical sink. Second, although reducing critical sink delay is the major objective here, we do not want to increase the wirelength too much for two reasons: 1. more wirelength corresponds to more routing resources, 2. more wirelength could increase the delay for the whole tree because the increased capacitive load. Hence, we add a weighted wirelength part in the "benefit" to discourage the wirelength increase. Finally, moving a branch can reduce the critical sink delay, but it could also cause other sinks to become critical. Therefore, we need to add constraints on "branch moving". When picking the candidate edge, we look at the two nodes that the edge is to connect. If any of the downstream sinks of these two nodes will become critical after "branch moving", we will not consider this edge.

6. EXPERIMENTAL RESULTS

We test our topology design algorithm on 12 nets from industrial circuits. These nets are the critical nets in the design and have negative slacks. Two metal layers are used for routing the nets. We perform all experiments on a 750MHz Sun Sparc-2 machine.

We compare our algorithm to the C-tree [12] and FLUTE [6]. Since C-tree algorithm is a combination of timing-driven Steiner tree construction and buffer insertion, we turned off the buffer insertion by specify no buffer library. FLUTE is used to generate high-quality RSMTs.

The results are summarized in table 2. We list the tree topology wirelength (WL), worst negative slack (WNS), total negative slack (TNS) and runtime for all 12 nets. Compared to C-tree, the topologies generated by our algorithm can achieve better or comparable TNS and WNS. And the wirelength of our topologies are better except for net10. Compared to FLUTE, although FLUTE can get much better wirelength than topologies generated by our algorithm, we are nearly always better in TNS and WNS. In addition, for the high-degree nets such as net4, net5 and net7, the

	degree	WL(10^{-2}um)			WNS(ps)			TNS(ps)			Runtime(ms)		
		Our	C-tree	FLUTE	Our	C-tree	FLUTE	Our	C-tree	FLUTE	Our	C-tree	FLUTE
net1	9	10752	10752	10416	-0.59	-0.70	-3.29	-1.13	-1.28	-5.06	0.068	20	0.006
net2	9	12096	12096	10752	-0.97	-0.97	-0.87	-0.97	-0.97	-0.87	0.054	10	0.004
net3	38	20997	23349	19653	-5.66	-5.71	-5.55	-5.66	-5.71	-5.55	0.512	90	0.272
net4	58	48384	53760	36960	0.00	-1.98	-21.61	0.00	-1.98	-144.31	1.399	380	0.615
net5	21	18144	19152	15456	-16.32	-15.62	-20.72	-32.34	-31.10	-41.03	0.304	60	0.094
net6	9	10416	10752	10080	-3.81	-3.91	-4.20	-7.31	-7.52	-8.07	0.110	20	0.007
net7	51	40320	42336	28896	-1.24	-2.14	-9.76	-1.24	-2.14	-26.41	1.256	590	0.419
net8	6	4844	4844	4844	-4.64	-4.64	-4.64	-9.61	-9.61	-9.61	0.070	10	0.003
net9	9	9548	9548	9212	-5.82	-5.93	-6.44	-8.39	-8.49	-9.80	0.119	30	0.006
net10	9	9828	9016	8008	-1.25	-0.52	-4.78	-1.25	-0.52	-6.66	0.133	20	0.006
net11	8	10556	10892	10416	0.00	0.00	0.00	0.00	0.00	0.00	0.053	10	0.004
net12	7	13440	13440	13440	-0.99	-0.99	-0.99	-0.99	-0.99	-0.99	0.053	10	0.004

Table 2: Experimental results.

TNS and WNS of FLUTE topologies are much worse. The reason for this is that in the high-degree nets, the chance of having long detours to critical sinks will be more than that in a low-degree net. From the resulted topologies, we observed a lot of detours happen from source to critical sinks. This supports our idea of RSMT has no guarantee on timing. For some low-degree nets, RSMT can get better timing results mainly because of the less capacitive load of the driver and no major detour from source to critical sinks. For the runtime, our algorithm is several hundreds times faster than C-tree and in the same order as FLUTE. For the low-degree nets, since we can directly find the A-tree topology from lookup table, the overhead of other parts such as timing analysis and post-processing is much more. But for high-degree nets, we need to break the net into several sub-nets and merge these sub-nets to form an A-tree. The overhead of other parts are just comparable to finding the A-tree.

7. CONCLUSION

8. REFERENCES

[1] F. K. Hwang. On Steiner minimal trees with rectilinear distance. *SIAM Journal of Applied Mathematics*, 30:104-114, 1976.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, NY, 1979.

[3] D. M. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plance Steiner tree problems: A computational study. In D. Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*, pages 81-116. Kluwer Academic Publishers, 2000.

[4] Geo Steiner - software for computing Steiner trees. http://www.diku.dk/geosteiner/.

[5] J. Griffith, G. Robins, J. S. Salowe, and T. Zhang. Closing the gap: Near-optimal Steiner trees in polynomial time. *IEEE Trans. Computer-Aided Design*, 13(11):1351-1365, November 1994.

[6] I. I. Mandoiu, V. V. Vazirani, and J. L. Ganley. A new heuristic for rectilinear Steiner trees. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, 1999.

[7] Chris Chu. FLUTE: Fast Lookup Table Based Wirelength Estimation Technique. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, pages 696-701, 2004.

[8] Chris Chu, Yiu-Chung Wong. Fast and Accurate Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. In *Proc. Intl. Symp. on Physical Design*, pages 28-35, 2005.

[9] S. Rao, P. Sadayappan, F. Hwang and P. Shor. The Rectilinear Steiner Arborescence Problem. Algorithmica, pages 277-288, 1992.

[10] W. Shi and C. Su. The Rectilinear Steiner Arborescence Problem is NP-complete. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 780-787, 2000.

[11] K. D. Boese, A. B. Kahng, G. Robins. High-Performance Routing Trees with Identified Critical Sinks. In *Proc. IEEE/ACM Design Automation Conf.*, pages 182-187, 1993.

[12] J. Cong, K. S. Leung, and D. Zhou. Performance-Driven Interconnect Design Based on Distributed RC Delay Model. In *Proc. IEEE/ACM Design Automation Conf.*, pages 606-611, 1993.

[13] J. Lillis, C.-K.Cheng, T.-T. Y. Lin, and C.-Y. Ho. New Performance Driven Routing Techniques with Explicit Area/Delay Tradeoff and Simultaneous Wire Sizing. In *Proc. IEEE/ACM Design Automation Conf.*, pages 395-400, 1996.

[14] C. J. Alpert, T. C. Hu, J. H. Huang, and A. B. Kahng. A Direct Combination of the Prim and Dijkstra Constructions for Improved Performance-Driven Global Routing. UCLA CS Dept. TR-920051, 1992.

[15] C. J. Alpert, et. al. Buffered Steiner Trees for Difficult Instances. In *Proc. Intl. Symp. on Physical Design*, pages 4-9, 2001.

[16] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal of Applied Mathematics*, 14:255-265, 1966.