A Novel Performance-Driven Topology Design Algorithm

Min Pan, Chris Chu

Electrical and Computer Engineering Dept.
Iowa State University, Ames, IA 50011
Email: panmin, cnchu@iastate.edu

Priyadarshan Patra

Intel Corporation
Hillsboro, OR 97124
priyadarshan.patra@intel.com

Abstract— **This paper presents a very efficient algorithm for performance-driven topology design for interconnects. Given a net, it first generates A-tree[1] topology using table lookup and net-breaking. Then a performance-driven post-processing heuristic not restricting to A-tree topology improves the obtained topology by considering the sink positions, required time and load capacitance to achieve better timing. Experimental results show that our new technique can produce topologies with better timing and is hundreds of times faster than traditional approach.**

I. Introduction

With technology scaling, interconnect delay has become the dominant factor in circuit delay, making effective performance-driven interconnect design vital for the timing closure. Topology design and buffer insertion are two main techniques for performance-driven interconnect design. Alpert et. al. [2] showed that the two-step approach of (1) constructing a Steiner tree, and (2) then running van Ginneken style buffer insertion, can be as good as the slower simultaneous approach. However, topology design, i.e. finding a good Steiner tree, itself is a difficult and time-consuming step. For nets with low degree[2], such as 2-pin or 3-pin nets, finding good topologies is easy. But for high-degree nets, constructing good topologies efficiently is both challenging as well as critical, for they are likely to be the cause of critical paths.

Rectilinear minimum spanning tree (RMST) is a class of topologies widely used in practice because efficient algorithms are available for their solution. However, the wirelength of an RMST can be as much as 1.5 times that of rectilinear Steiner minimal tree (RSMT) [3]. RSMT is another class of well-researched topologies. But RSMT construction being NP-complete [4], no efficient algorithm exists, and a lot of work has focused on approximation algorithms for it. Batched 1-Steiner heuristic [5] and the heuristic proposed 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 for VLSI applications based on a table lookup technique.

In addition to wirelength-driven topologies such as RMST and RSMT, many timing-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. [1] proposed A-tree algorithm to find a min-area shortest paths tree. In [12], Permutation-constrained routing trees (P-tree) algorithm reported better area objectives than SERT and A-tree. Alpert et al. [13] proposed AHHK trees as a direct trade-off between Prim's MST algorithm and Dijkstra's shortest path tree algorithm, and used in the C-tree algorithm [2] for timing-driven Steiner tree construction. However, all of these algorithms are not very efficient to address large industrial designs with a substantial number of high-degree nets, especially for an integrated route-and-place flow. Although most of the nets in a design are of low degree, there are still a considerable number of high-degree nets (12% nets have degree ≥ 8 [7]). And these high-degree nets are more likely to be timing-critical. Hence, our goal is to develop a fast, performance-driven topology design algorithm applicable to optimizing delay properties of a large-class of nets early-on in the design cycle.

This work was partially supported by the SRC under Task ID 1206 and NSF under grant CCF-0540998.

[1]In [1], it defined that a rectilinear Steiner tree is an A-tree if every path connecting the source and any node on the tree is a shortest path.

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

In this paper, we present a novel method to efficiently design performance-driven topology for nets. We develop a very fast algorithm to construct an A-tree based on table lookup and net-breaking. Then we apply post-processing techniques, not restricted to A-trees anymore, on the obtained A-tree topology to further improve the timing for the net.

Our main contributions include the following:

- A very efficient algorithm to construct the A-tree *potentially optimal wirelength vector* (POWV) [7] and topology table for all the nets with degree up to a certain value
- A fast A-tree construction algorithm using table lookup and net-breaking techniques for high-degree nets
- A performance-driven post-processing technique, which modifies the A-tree topology to further improve the timing

Experimental results show that our new algorithm can generate topologies with better timing than the timing-driven tree construction algorithm in C-tree [2]. Moreover, our algorithm is 371× faster. Therefore, it is very suitable for performance-driven topology design for a large number of nets, in an integrated physical design flow.

The remainder of the paper is organized as follows. In Section 2, we discuss the topologies for the performance-driven interconnect design. Section 3 describes the fast algorithm to generate A-tree lookup table. In Section 4, we present the algorithm to construct A-tree using table lookup and net-breaking. In Section 5, a performance-driven post-processing technique is proposed. Experimental results are shown in Section 6.

II. Topology for performance

As mentioned earlier, RSMT is a class of widely used topologies with good wirelength metric. However, an RSMT may contain many detours from the source to some sinks resulting in bad timing for them. A simple illustrative example is shown in Figure 1. Sink t_4 is the critical sink here. We can see that there is detour from the source s to t_4 which harms the timing result. Therefore, despite its good wirelength, it is not suitable for performance-driven topology design. And we need some other types of topologies for the timing purpose.

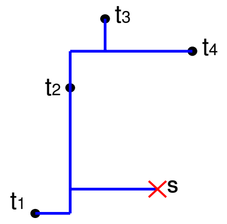

Fig. 1. Detour in RSMT.

A-tree is a class of topologies with good properties for performance-driven interconnect design. First, an A-tree is a shortest path tree (SPT), thus no detours between the source and a sink. In addition, it has been shown in [1] that minimizing total wirelength of an A-tree leads to simultaneous optimization of different components of sink delays. Such a harmony would be impossible to achieve for general routing topologies. Hence we focus on A-trees and their subsequent refinement as our goal. However, finding A-tree with minimum wirelength is an NP-complete problem [10]. Inspired by the table lookup idea of FLUTE [7], [8], we propose a very efficient way to construct A-trees using table lookup techniques to be discussed in detail below.

III. A-Tree Lookup Table Generation

In this section, we focus on A-tree lookup table generation. We first discuss how to group infinite number of nets into finite number of groups so that a practical lookup table can be constructed. Then a *Configuration Graph* approach is proposed to generate the lookup table efficiently. Finally, we introduce the concepts of *Abstract Topology* and *Topology Signature* to reduce the complexity in topology table generation.

A. A-tree Lookup Table Organization

FLUTE [7], [8] is a lookup table based RSMT algorithm. It is shown that the set of all degree d nets can be partitioned into $d!$ groups according to the relative positions of their pins. The relative positions of pins is defined by *vertical sequence*. Consider an d-pin net. Let x_i be the x-coordinate of some vertical Hanan grid line such that $x_1 \leq x_2 \leq ... \leq x_d$. Similarly, let y_j be the y-coordinate of some horizontal Hanan grid line such that $y_1 \leq y_2 \leq ... \leq y_d$. 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. $s_1 s_2 ... s_d$ is the *vertical sequence*. As illustrated in Figure 2. All the nets with the same *vertical sequence* fall in one group in the lookup table. For each group, the wirelength of all possibly optimal routing topologies along the Hanan grid [14] can be written as a small number of linear combinations of distances between adjacent Hanan grid lines [7]. 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. Each POWV and one corresponding topology are stored into a lookup table. To get the RSMT for a net, the algorithm computes the wirelengths corresponding to the POWVs for the group the net belongs to, and picks the one with the best wirelength.

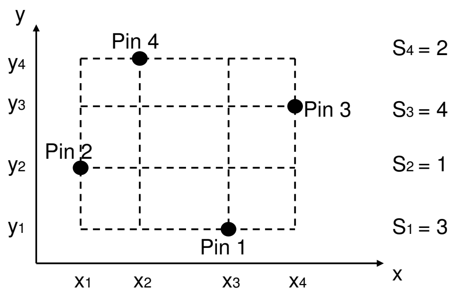

Fig. 2. Illustration of some notions.

We also use the *vertical sequence* to group the nets. However, for A-tree, only the *vertical sequence* is not enough to group the nets. The reason is that not only the relative pin positions but also the source pin location define the group of nets sharing the same POWVs (*potentially optimal wirelength vectors*) and topologies. Therefore, we first divide all the nets with degree d into $d!$ groups according to their *vertical sequence*. Then we further divide every group into d subgroups. For subgroup $1, 2, ..., d$, the corresponding source pin is pin $1, 2, ..., d$, respectively. For each subgroup, we will have a set of POWVs and their corresponding topologies stored in the table. Note that in FLUTE, a POWV represents **rectilinear Steiner trees** which can potentially have minimum wirelength. In contrast, in this work a POWV represents **A-trees** that can produce optimal wirelength.

Our POWV table stores POWVs for every subgroup. Moreover, while the FLUTE table contains only one topology for each POWV, in the current work we efficiently store *all* topologies for a POWV. This allows us to explore a very large set of topology alternatives for better timing and good wirelength – although they may all have same wirelength. In this sense, constructing the A-tree table is more sophisticated than constructing RSMT tables of FLUTE.

B. Configuration Graph

Since there are a lot of possible topologies for each subgroup (defined by *vertical sequence* and the source pin) and the number of subgroups ($d \times d!$ for degree-d nets) are huge, the table generation can be very time-consuming. An efficient way needs to be developed instead of directly enumerating all possible topologies. *Boundary compaction* [7] is a very efficient technique to generate topologies. For a net, the *boundary compaction* technique reduces the Hanan 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 3 uses the compaction of left boundary to illustrate the idea.

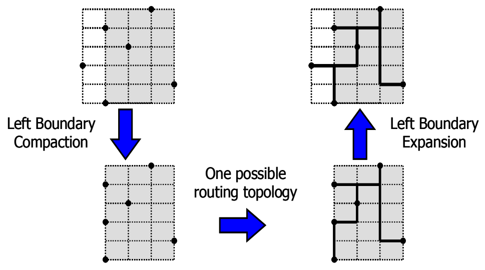

Fig. 3. Boundary Compaction.

We observe that most of the A-tree topologies can be generated by *boundary compaction*. Hence, we employ *boundary compaction* to generate the A-tree topologies. We recursively compact any boundary without the source on it until the grid is compacted into a single node (source). The edges created during the process form an A-tree with the source being the final left node. If we choose different ways for compaction, we will obtain different A-tree topologies. We define the *compacting sequence* as the sequence of compaction operations that reduces the original grid to a single node. Hence, one *compacting sequence* corresponds to one A-tree topology source at the single node left after the compactions. A direct idea for finding different topologies is to look at the different *compacting sequences*. Unfortunately, the number of *compacting sequences* is huge. For each group, the number of different sequences = $\binom{2(d-1)}{d-1} \times 2^{d-1} \times 2^{d-1}$ 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 group of 9-pin nets $= \binom{16}{8} \times 2^8 \times 2^8 = 843448320$. 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 store the different topologies that potentially can result in best wirelength. 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. Our idea to generate and prune the sequences is using a graph called *Configuration Graph*. We will show that we can generate POWVs for all subgroups in one group (same *vertical sequence* but different source) simultaneously.

First, we define some terms. A *Pin Configuration (PC)* is the configuration of a set of pins on the Hanan grid. This configuration only defines the pin positions on the Hanan grid without considering any real geometry size. If we apply boundary compaction on a *PC*, we will get a new one. The new *PC* has no pin on the compacted boundary and can have the same or less pins than the original because some pins may collapse together.

If the original *PC* is transformed into a new *PC* with a specific bounding box by a sequence of compaction, the new *PC* is independent on the compactions performed, as stated in Lemma1.

Lemma 1: The bounding box of a PC in the original Hanan grid defines the PC.

Proof idea: As shown in Figure 4, the whole grid is the Hanan grid of original pin configuration and the center region 3 is the new configuration with bounding box for the center 16 small squares 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 touched. 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.

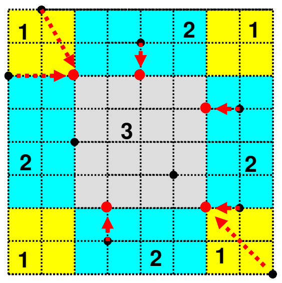

Fig. 4. Lemma 1 Proof.

In *Configuration Graph*, every node corresponds to a *PC*. 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 *PC* for a *vertical sequence*. We call it *Start Node* because any boundary compaction operation starts with it. The other type is the *CN* with the *PC* in which all the pins are compacted to a single point on the grid. We call them *End Nodes* because any *compacting sequence* will end with such a *CN*. Note that an A-tree topology is obtained when reaching an *End Node*. A *Partial wirelength Vector* (PWV) is the Wirelength Vector (WV) with undecided entries obtained after a sequence of compactions. For example, if a full WV is $(1221, 1121)$, a PWV could be $(1xx1, 11x1)$ (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 associated with it. They are the PWVs corresponds to the edges created by *compacting sequences* that can result in the the *PC* associated with the *CN*. If compacting one boundary of the *PC* associated with a *CN* can get the *PC* of another *CN*, an edge is created from the first *CN* to the second. An example of *Configuration Graph* is shown in Figure 5.

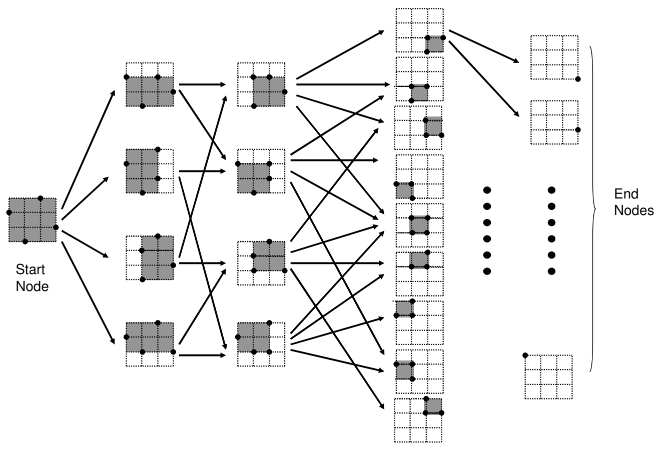

Fig. 5. Configuration Graph.

From Lemma 1, we know the number of *CN*s in *Configuration Graph* is a small number. It is just the number of different bounding boxes we can find in the original Hanan grid. $\#CN = \sum_{i=1}^{d}\sum_{i=1}^{d}(d+1-i)(d+1-j)$, if $d = 9, \#CN = 2025$. Actually, we can even do better. Instead of using the original *PC* as the *Start Node*, we start from a new *Start Node* that is obtained by compacting the original *PC* once in all 4 directions (left, right, top and bottom). We have the following lemma for the new *Start Node*. The proof is similar to the Lemma 2 in [7]. The only difference is that when the source pin is on one of the 4 boundaries, we will treat the new source for the reduced grid at the position of the pin created by compacting the original source.

Lemma 2: No POWV will be lost by starting boundary compaction at the new Start Node.

Since we begin from this new *Start Node* with the bounding box size $(d-2)\times(d-2)$, 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 $d = 9, \#CN = 784$.

Configuration Graph allows us to do the pruning very efficiently. We have the following lemma.

Lemma 3: 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 Lemma 1 we know that the undecided part of WV is the same for V_1 and V_2 because of the same *PC*. 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 Lemma 3, we can use *Configuration Graph* to prune the PWVs using "PWV dominance" at each *CN*s efficiently. 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 has been generated. It can prune the bad WV as early as possible and accelerates the pruning process a lot. Another advantage for the *Configuration Graph* approach is that if we construct the *Configuration Graph* for a given *vertical sequence*, we already obtain POWVs for all the subgroups corresponding to different source pins. We will see this later.

Hence, we construct *Configuration Graph* for any given group (*vertical sequence*) as following. We start from the new *Start Node* mentioned in Lemma 2. Its corresponding PWV is $(1xx...x1, 1xx...x1)$ because we have four edges due to the first 4 *boundary compactions*. Then, we compact the four boundaries of the current *PC* to get four new *CN*s, their corresponding PWVs, and four edges. Similarly, we just recursively apply boundary compaction on the new created *CN*s and generating more *CN*s. Note that compacting different *CN*s can result in the same *CN* but different PWVs. Therefore, we need to prune the PWVs using "PWV dominance" at each node. Only the PWVs left after pruning associated with a *CN* will be used to generate further PWVs when compacting this *CN* to generate new *CN*. This recursive new *CN* generation will stop when the new generated *CN* is an *End Node*, where no compaction can be applied. After generating all the *CN*s and their corresponding PWV list, we obtain the whole *Configuration Graph* and can easily find the A-tree topologies from it.

It is easy to see that any path from the *Start Node* to an *End Node* corresponds to a *compacting sequence*, hence a tree topology. But our goal is not to find any tree topology but to find A-trees with a specific source pin. We have the following lemma for the generated tree topologies.

Lemma 4: Any tree topology generated by a compacting sequence corresponding to 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.

Therefore, for any pin as the source, we can easily find the POWVs for the A-trees. We just need to look at the *End Node* corresponding to the position of the source pin and all the POWVs associated with that *End Node* are the POWVs for A-trees with the source pin. Since we have *End Node*s corresponding to every position in the Hanan grid, POWVs for every pin as the source can be obtained simultaneously from the *Configuration Graph*.

C. Abstract Topology and Topology Signature

By now, we can obtain the POWVs for A-tree topologies from *Configuration Graph*. But unlike FLUTE, we are not satisfied with storing one arbitrary topology corresponding to each POWV. Instead, we want to explore good A-tree topologies for performance. Therefore, we want to find all different A-tree topologies corresponding to each POWV and store them in the table.

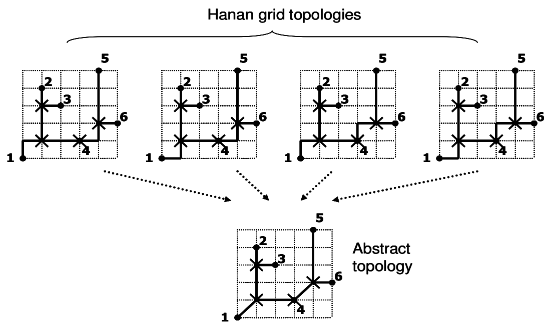

Fig. 6. Abstract topology.

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 different but *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) on Hanan grid and the connections between nodes. The only difference between *equivalent* topologies is the embedding for the connections. 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 on Hanan grid. If two *compacting sequences* generate the same topology or *equivalent* topologies, their corresponding *Abstract topology* are the same. Therefore, we only need to store the different *Abstract topologies* for POWVs. Figure 6 illustrates 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 connections. If there are 2 different routing on Hanan grid for 10 two-pin connections in 15, # embedded topologies = 2^{10} = 1024. If we just directly save the different topologies, we may need thousands of times space than just storing *Abstract Topologies*.

The way we find different *Abstract Topologies* is as following. We start from the *End Node* corresponding to the source pin and for every POWV, trace back in the reverse direction of edges until reaching the *Start Node*. This back trace will form different paths corresponding to different *compacting sequences*. Since each *compacting sequence* corresponds to an A-tree topology, we can get all the possible A-tree topologies for each POWV. Then we can compare their *Abstract Topologies* and just store the different *Abstract Topologies* in the table.

However, there is still one problem in generating and comparing the *Abstract Topologies*. To know whether a topology is redundant, we need to first find its corresponding *Abstract Topology* and compare it to all the other *Abstract Topologies* already found. This topology generation and comparison take a lot of runtime. We want to make it easier and faster. So we introduce the *Topology Signature*. A *Topology Signature* of a Hanan grid 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 find whether two tree topologies have the same *Abstract Topology*.

Theorem 1: For A-trees generated by boundary compaction, two trees A and B has the same Topology Signature if and only if A and B has the same Abstract Topology.

Because of the page limit, we have to skip the proof. Theorem 1 tells us that *Abstract Topology* and *Topology Signature* has one-to-one correspondence. So *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 simply compare their Steiner nodes positions on Hanan grid. After we find all the topologies with different *Topology Signatures*, we store their corresponding *Abstract Topologies* in the table.

TABLE I
STATISTICS FOR POWV.

Degree	# groups	# POWVs in a group		
n	n!	Max	Avg	Min
2	2	1	1	1
3	6	1	1	1
4	24	8	6	1
5	120	18	12.63	1
6	720	36	25.31	1
7	5040	70	50.69	1
8	40320	144	99.55	1
9	362880	282	193.19	1

Till now, we can generate A-tree POWVs and *Abstract Topologies* and store them in the table grouped by the *vertical sequence* and the source pin. Table I gives the statistics of our POWV table for the nets up to degree 9. And we observed in experiments that all POWVs for the nets up to degree 9 have only **ONE** *Abstract Topology*. With our algorithm based on *Configuration Graph*, it only takes less than 15 minutes to generate the table for all nets up to degree 9 compared to many hours for generating FLUTE table up to degree 9. The table size for POWV and *Abstract Topologies* up to degree 9 are 21MB and 75MB, respectively. Note that this table only needs to be generated once. And after loaded into the memory, it can be used for as many times as wanted.

IV. A-TREE TOPOLOGY CONSTRUCTION AND NET-BREAKING

In last section, we construct the A-tree POWV table and corresponding 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 the 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.

However, it is not practical to generate table for high-degree nets because of the huge table size. 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 different from that in [8] because 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]. If a net satisfies that all the pins in the net can be partitioned into two sets which reside in two diagonal regions, it can be optimally broken into two sub-nets formed by these two sets. An extra pin is introduced in both sub-nets. The pin is positioned at the bounding box corner of one sub-net which is closest to the other sub-net. After the breaking, only one sub-net contains the source. 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, 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 the pin where the net is broken in both sub-nets, we introduce an extra pin and include it in both sub-nets. This extra pin will become the source of the subset that does not have the original source in it. The position of this extra pin is found by a "source propagation" technique. Assume the breaking direction and position are known. We first project the source on the breaking line to get the initial position of the new source. However, this position may not be good in some cases. For example, in Figure 7, all the pins in the right sub-net have bigger y-coordinates than the source. If we put new source at the position of the projection of the source on the breaking line, it could lead to unnecessary extra wirelength. In order to solve this problem, we slide the source projection along the breaking line until it has the same y-coordinate as the pin with the smallest y-coordinate in the right sub-net. 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 in what direction the net is broken and which sub-net the source is in. The new source found is noted as "propagated source".

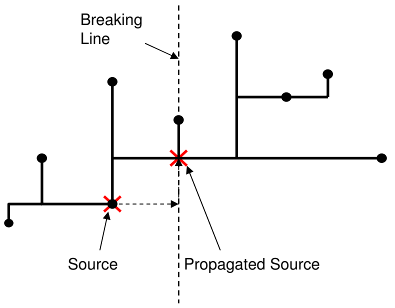

Fig. 7. Source propagation.

Lemma 5: Breaking a net at the "propagated source" generates an A-tree by combining the A-trees of both sub-nets. (The sources of two sub-nets are the original source 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. ■

For the heuristics of choosing a good direction and position to break the net, we apply a slightly different heuristics from that in [8]. We also compute a score which is a weighted sum of three components. For the first and third component, we follow the way in [8] to find them. But for component two, since the real breaking point is the "propagated source", we will consider the lengths of the segments adjacent to the "propagated source" other than those adjacent to the pin on the candidate breaking line.

After we break the net into sub-nets with degree no more than D, we can look up the table to find out the topologies for them. Finally, we merge these subtrees to form the whole A-tree.

After the A-tree for the net is obtained, we apply a heuristic to repair the errors caused by the nonoptimality of the table and net-breaking. For each node on the tree (pins and Steiner nodes), we try to connect it to the closest point on the tree and in the direction of the source. This will further improve the wirelength and still maintain the A-tree property.

V. Performance-driven Post-processing

So far, we can construct a good A-tree topology for any given net by net-breaking and table lookup. However, the topology is still a generic A-tree without consideration of the timing properties of a specific net. In general, A-tree is a good topology in performance-driven routing if there is no difference between all the sinks in criticality. However, for a specific net, different sinks have different capacitive load, required time and distance to the source. This makes it 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 our fast table lookup based A-tree construction technique. Therefore, we propose a performance-driven post-processing heuristic to modify the tree topology to achieve better timing result.

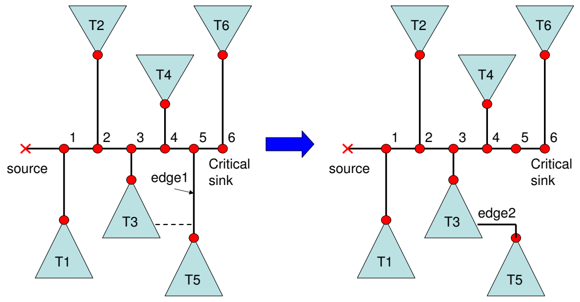

Fig. 8. Branch Moving.

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 A-tree. The basic idea is to change the load distribution on the critical path to reduce the delay on critical sinks. To easily understand the technique, let us look at a simple example. As illustrated in Figure 8, 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 use Elmore delay model for our delay computation. Therefore, the delay on the critical sink is the sum of a series of RC terms, $Delay(Critical\ sink\ 6) = \sum_{i=1}^{6} R_i C_i$, where R_i is the path resistance from source to node i, C_i is the downstream capacitance of the subtree T_i 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 as well. Hence, we try to move the branches so that the delay on the critical sink is reduced. For instance, in Figure 8 (left), we find a possible edge (dashed line) which connect the branch tapped to node 5 to the subtree tapped to node 3. It is easy to compute the delay 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 moving 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 an 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 and power, 2. more wirelength could increase the delay for the whole tree for 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

TABLE II
EXPERIMENTAL RESULTS. * AVERAGE IS OVER ALL 29 TESTCASES.

	deg	Tree Wirelength			WNS(ps)				TNS(ps)				Runtime		
		Our	Ctree	FLUTE	Our		Ctree	FLUTE	Our		Ctree	FLUTE	Our	Ctree	FLUTE
					A-tree	Final			A-tree	Final					
t1	9	1	1.029	0.914	-0.97	-0.80	-0.97	-0.87	-0.97	-0.80	-0.97	-0.87	1	111	0.11
t2	38	1	1.112	0.936	-5.66	-5.40	-5.71	-5.55	-5.66	-5.40	-5.71	-5.55	1	191	0.57
t3	58	1	1.176	0.809	0.00	0.00	-1.98	-21.61	0.00	0.00	-1.98	-144.3	1	704	1.15
t4	21	1	0.983	0.793	-16.32	-14.33	-15.62	-20.72	-32.34	-28.52	-31.10	-41.03	1	286	0.48
t5	9	1	1.032	0.968	-4.10	-3.81	-3.91	-4.20	-7.95	-7.31	-7.52	-8.07	1	250	0.13
t6	51	1	1.145	0.782	-1.82	0.00	-2.14	-9.76	-1.82	0.00	-2.14	-26.41	1	1255	0.89
n_1885	27	1	1.077	0.860	-4.56	-1.51	-3.73	-6.19	-4.56	-1.51	-3.73	-6.19	1	346	0.73
n_1898	39	1	1.052	0.907	-4.91	-2.73	-4.75	-8.85	-4.91	-2.73	-4.75	-8.85	1	304	0.87
n_2045	54	1	1.181	0.897	-22.71	-22.71	-25.29	-23.28	-126.0	-126.0	-155.4	-147.3	1	455	0.75
n_2049	45	1	1.158	0.924	-2.95	-0.62	-3.55	-5.43	-2.95	-0.62	-5.27	-7.97	1	468	0.84
n_2071	29	1	1.079	0.890	-12.99	-10.66	-14.51	-14.38	-12.99	-10.66	-14.51	-14.38	1	375	0.56
n_2072	69	1	1.180	0.845	-14.72	-12.09	-22.98	-61.55	-48.39	-37.92	-96.73	-1420	1	385	0.74
Avg.*	28	1	1.095	0.915	-7.38	-6.09	-7.41	-10.55	-21.96	-18.76	-22.87	-75.27	1	371	0.487

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.

VI. EXPERIMENTAL RESULTS

We test our new method on 2 sets of industrial nets. The first set is from a design in $65nm$ technology and the second is from a design in projected $45nm$ technology. Two metal layers are used for routing. These two sets of nets are critical nets extracted from the designs after the timing analysis. The first set has 17 nets and the second set has 12 nets.

We try to find performance-driven topology design algorithms in the public domain. However, most of these algorithms are together with other interconnect optimization techniques such as buffer insertion and wire sizing. Since our focus is topology design, it is very hard to find some direct comparison with these algorithms. Hence, we compare our new algorithm with C-tree [2] and FLUTE [8]. Both of them are downloaded from the GSRC Bookshelf [15]. Since C-tree algorithm is a combination of timing-driven Steiner tree construction and buffer insertion, we turned off the buffer insertion by not specifying any buffer library. In addition, we also turned off the sink polarity by setting all sinks the same polarity as source. FLUTE is used to generate near-optimal rectilinear Steiner minimal trees. All results are generated on a 750MHz Sun Sparc-2 machine.

The result for 6 nets from each set are reported in table II. We compare the tree wirelength, worst negative slack (WNS), total negative slack (TNS) and runtime for the three algorithms. Note that we report WNS and TNS for A-trees obtained by our algorithm and the final trees obtained after post-processing. We can see the post-processing technqiue is very effective in reduce WNS and TNS. The wirelength and runtime are normalized to our algorithm. For all the 29 nets, our algorithm always achieves the best WNS and TNS among the three algorithms. We also take the average on all the 29 nets for these measurement. On average, C-tree uses 9.5% more wirelength than ours as FLUTE uses 8.5% less wirelength. And WNS and TNS of trees generated by our algorithm are 82.2% and 57.7% that of C-tree and FLUTE, respectively. From the comparison to FLUTE, we can see that although the tree generated by our algorithm has more wirelength than RSMT, their performance is better. This verifies our proposition that RSMT may not be good for timing. For the runtime, we are just slightly slower than FLUTE and 371 times faster than C-tree. Note that for the nets with degree more than 60, the runtime of our algorithm is about 1ms, which means we can handle 1000 that kind of high-degree nets in one second.

VII. CONCLUSION

In this paper, we proposed a novel method for interconnect topology design. First, a table lookup based A-tree algorithm finds good A-tree topologies very efficiently. Then a performance- driven post-processing further improves the timing by modifying the A-tree topology based on the physical information such as sink positions, capacitive load and required time. Experiments show that the proposed method produces very promising results in both solution quality and runtime.

Extending the topology design techniques, our future work will address interconnect optimization by including buffer insertion and wire-sizing.

REFERENCES

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

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

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

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

[5] J. Griffith, G. Robins, and J. S. Salowe. 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*, 157-162, 1999.

[7] Chris Chu. FLUTE: Fast Lookup Table Based Wirelength Estimation Technique. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, 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*, 28-35, 2005.

[9] S. Rao, P. Sadayappan, F. Hwang and P. Shor. The Rectilinear Steiner Arborescence Problem. Algorithmica, 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*, 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.*, 182-187, 1993.

[12] 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.*, 395-400, 1996.

[13] C. J. Alpert, et. al. A Direct Combination of the Prim and Dijkstra Constructions for Improved Performance-Driven Global Routing. UCLA CS Dept. TR-920051, 1992.

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

[15] A. E. Caldwell, A. B. Kahng, and I. L. Markov. VLSI CAD Bookshelf. http://www.gigascale/org/bookshelf.