S-Tree: A Technique for Buffered Routing Tree Synthesis

Milos Hrkic
mhrkic@cs.uic.edu

John Lillis
jlillis@cs.uic.edu

University of Illinois at Chicago, CS Dept., Chicago, IL 60607

ABSTRACT

We present the S-Tree algorithm for synthesis of buffered interconnects. The approach incorporates a unique combination of real-world issues (handling of routing and buffer blockages, cost minimization, critical sink isolation, sink polarities), robustness and scalability. The algorithm is able to achieve the slack comparable to that of buffered P-Tree [7] using less resources (wire and buffers) in an order of magnitude less cpu time.

Categories and Subject Descriptors

B.7.2 [**Hardware**]: Integrated Circuits—*Design Aids*

General Terms

Algorithms

Keywords

timing optimization, Steiner trees, routing, buffer insertion.

1. INTRODUCTION

In the deep submicron era effective performance driven interconnect synthesis has become crucial for achieving chip-level timing closure. When synthesizing an interconnect structure for a timing critical net, there are a number of degrees of freedom which may be exploited including *buffer insertion*, *wire tapering* and *topology*. In the past ten years we have seen the growth of a fairly substantial body of work in the area. Many of the practical buffer insertion techniques in use today can be traced to the seminal work of van Ginneken [10] which proposed a dynamic programming algorithm for inserting buffers into a *given* topology. In addition, the buffer insertion techniques for two-pin nets has received some attention (e.g., [11], [5]). In the area of routing topology construction, there are several works of particular interest to this paper. These include the *P-Tree* based methods [8], methods for timing driven routing of two-pin nets [4], [11] and methods which combine buffer insertion and topology construction (e.g., [7], [9], [3]).

In this paper we address the interconnect synthesis problem with a number of criteria and objectives in mind which we believe to be of practical importance but not adequately addressed in the literature. An overview of the issues emphasized is as follows.

Simultaneous Buffer Insertion and Tree Construction

We believe that substantial overall improvements in solution quality can be achieved by not viewing the interconnect synthesis problem as a two phase process (routing topology construction followed by buffer insertion) but as a simultaneous approach. While some past works have attempted such a unification (e.g., [7], [3]), those methods do not meet some of our other objectives (e.g., scalability and critical sink isolation).

Handling Routing and Buffer Blockages

In a real design there are typically limitations on where wires can be routed and where buffers can be inserted. We address these issues explicitly in the proposed algorithms by adopting a general graph model for our routing target.

Effective Isolation of Critical Subtrees

One potential weakness of some topology constructions is that they are oblivious to sink criticality. For example in the P-Tree method, a sink permutation is formed where consecutive subsequences of sinks are candidates for subtrees. Since the sink permutation is determined by the relative *physical locality* of the sinks and is oblivious of such criteria as sink criticality (i.e. *temporal locality*), some very high performance and/or cost effective solutions may fall outside the solution space. It is instructive to consider the case in which buffers are used to decouple a non-critical group of sinks. In such a case (in order to conserve scarce buffering resources) it may be preferable to disregard (to a limited degree) the physical locality of such sinks at the expense of additional wire-length so that they may be "tapped-off" with a single buffer. A similar phenomenon occurs when sinks have specified signal polarity requirements as pointed out recently in [1]. The algorithm presented herein is designed to deal with both issues.

Ability to Handle Relatively High Fanout Nets

It is often the case that in a typical modern design flow we see some signal nets with relatively high fanout (e.g., 15 pins or more). Such nets may result from the removal of buffers introduced by logic synthesis producing nets that will be re-buffered with more physical information. Such nets are often

This work was supported in part by NSF CAREER Award CCR-9875945 and in part by SRC under contract 2001-TJ-914.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

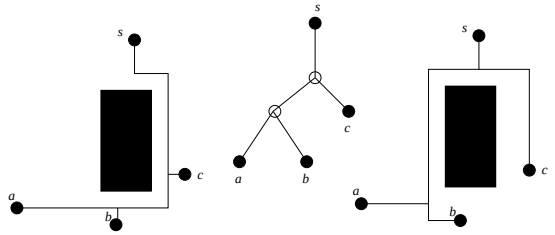

Figure 1: A topology for a 3-sink net and two physical embeddings of that topology. Pin s is the driver.

crucial to overall system performance. Prior methods which exhibited significant robustness and generality of problem formulation tended to not scale well to such size of nets (e.g., [7], [3] which are high-degree polynomial and exponential respectively).

Cost/Performance Tradeoffs

We recognize that while our algorithms focus on the one-net problem, a real CAD system's objective is to drive an entire design to timing closure. Thus, multiple nets must compete for wiring and buffering resources and it is clearly not sufficient to, for example, maximize the performance of a particular net without paying attention to some measure of cost incurred. Further, when one considers that recent estimates indicate that, in the near future, perhaps more than 700K buffers will be required simply to buffer global or near-global interconnects [2], it becomes clear that the over-use of buffering resources will have dramatic consequences.

With these issues in mind we have developed an algorithm called *S-Tree*. There are three degrees of freedom exploited by S-Tree: *Topology*, *Embedding* (placement of Steiner nodes in a target graph) and *Buffer placement*. An overview of how these degrees of freedom are exploited is given in the next section. We first focus primarily on the solution space covered by the algorithm and then sketch some of the implementation details.

2. SOLUTION SPACE DESCRIPTION

We begin with the problem of mapping the Steiner nodes of a given topology to the vertices of a target graph (typically a partial grid graph). Fig. 1 illustrates the corresponding solution space; for the given topology in the middle, we show two possible embeddings. Even though the topology is fixed, there is clearly some flexibility in the embedding space which may be useful particularly in timing driven applications. This flexibility however is certainly limited and precludes certain (potentially useful) solutions (e.g. it may be useful depending on timing requirements to isolate sink a completely with its own path from the driver s; this is not possible for this topology).

Finding an optimal embedding (and buffer assignment) for a given topology is in itself an interesting problem. However, we have generalized the concept in S-Tree to provide more flexibility in the topology space (in fact, exponential flexibility).

Given this notion of topology embedding, the S-Tree algorithm provides additional flexibility in the topology space. Suppose that in addition to a topology T, we are also given a partitioning of the sinks into two disjoint sets S_1 and S_2 (this partitioning may be arbitrary, in our implementation

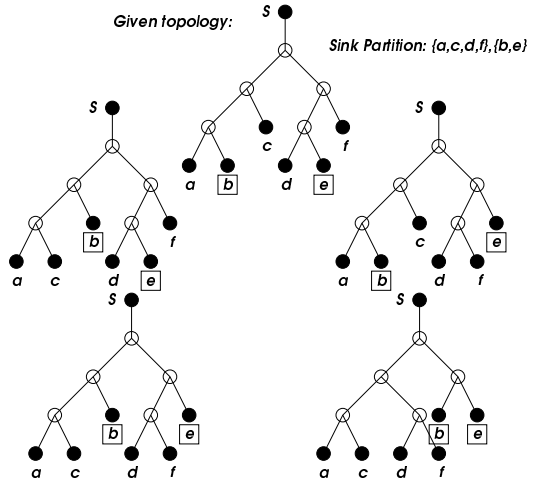

Figure 2: S-Tree topology solution space example.

we partition sinks based on timing criticality). Now consider a subtree in T rooted at vertex u and with left and right children $l(u)$ and $r(u)$ (if they exist). Some sinks in the subtree belong to S_1 and others to S_2. If we have two sets of topologies L and R (covering disjoint sets of sinks), then $L \times R$ is the set of all topologies where a root has a member of L as its left child and a member of R as its right child (basically a cross-product; if one of the sets is empty, the output is the other non-empty set). Given these notions, let $P_{12}(u)$ be the set of topologies in the S-Tree space for the subtree rooted at u and covering all sinks in the subtree. Let $P_1(u)$ be the set of topologies for the subtree rooted at u and covering only those sinks in the subtree which are in partition S_1. Similarly $P_2(u)$ is the set of topologies rooted at u and covering sinks from S_2. Since u must be either in S_1 or S_2, the base case is trivial: If $u \in S_1$ and u is a sink then $P_{12}(u) = P_1(u) = \{u\}$ and $P_2(u) = \emptyset$. Case where u is a sink in S_2 is handled similarly. The following recurrence relations establish these sets:

$$P_1(u) = \{P_1(l(u)) \times P_1(r(u))\}$$
$$P_2(u) = \{P_2(l(u)) \times P_2(r(u))\}$$
$$P_{12}(u) = \{P_{12}(l(u)) \times P_{12}(r(u))\} \cup \{P_1(u) \times P_2(u)\}.$$

The expansion in solution space comes from $P_{12}(u)$. It allows us to "promote" one of the subsets and *stitch* it to the root (giving rise to the S-Tree name).

Fig. 2 illustrates the solution space for a 6-sink topology with a given sink partitioning. Naturally, the given topology (on top) is included in the space in addition to the other four shown. Note that while three topologies are isomorphic to the given topology, some of the sinks have been re-labeled (b and e now being closer to the root and c and f farther).

Two notions motivate this idea. First, in a dynamic programming framework, the optimal solution in the expanded solution space can be found with only a small amount of additional work vs. the totally fixed topology case. Second, it is well-suited to timing related issues where it is often desirable for groups of critical or non-critical sinks to be in the same subtree. If S_1 and S_2 are critical and non-critical sinks, then the stitching operation enables this quite naturally.

To illustrate the second point, consider Fig. 3. In the figure we see a given topology and a sink partitioning. Now suppose that sink b is highly critical. It is then likely that

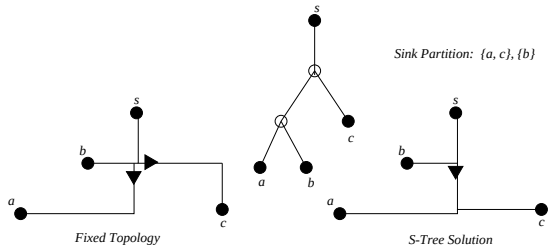

Figure 3: Illustration of critical sink isolation and buffer savings in S-Tree.

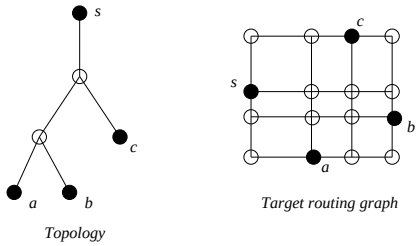

Figure 4: Abstract topology and target graph.

we desire an unbuffered path from s to b which decouples all off-path capacitance with one or more buffers. For the given topology, the best we can do is illustrated on the lower-left. In S-Tree however, the solution on the lower-right becomes possible. Not only will the S-Tree solution have lower delay to sink b, it also uses just a single buffer. We expect that such savings will be increasingly important as more and more buffers are needed in typical designs and conservation of buffering resources becomes crucial.

Given this overview of S-Tree, we can now informally define the optimization problem: Given technology parameters, sink timing requirements, a buffer library, a target routing graph, a topology and a sink partition (S_1, S_2), find a topology in the corresponding space, its embedding and buffer assignments which minimizes cost (e.g., number of buffers, total area, total wire length) subject to timing constraints being met.

2.1 Dynamic Programming Overview

The Dynamic Programming algorithm used for S-Tree is similar in flavor to several past works (e.g., [6], [8]). A candidate solution for a buffered subtree rooted at some vertex in the target routing graph will be represented by its *signature* (p, c, q) indicating that this candidate subsolution incurs cost p, has upward capacitance c and has required arrival time q at its root. Given this notion of a signature, a subsolution is *non-dominated* if no other solution is superior in *all three* dimensions. Any dominated solution may be discarded. Note that while, as many papers have noticed, only c and q are necessary to assure a maximum q solution, all three parameters appear necessary if we want to avoid excessive cost overhead. This increases run-time complexity, but we believe it is truly necessary for practical solutions.

Now let u be a vertex in the given topology T and v be a vertex in the target routing graph G (Fig. 4). We then define the following sets.

$A_1(u, v)$: the set of non-dominated solutions for subtree rooted at u in T, with root embedded at v in G *and* connecting only sinks in S_1 contained in that subtree.

```
Subroutine: GenDijkstra(A^b, G)
        A^b: Joined solutions
        G: Target routing graph
d1      A ← ∅
d2      for each solution in A^b(v)
d3        Insert A^b(v) into Queue
d4      endif
d5      while Queue ≠ ∅
d6        v_1 ← Top Queue
d7        if Solution(v_1) is not suboptimal
d8          A(v_1) ← A(v_1) ∪ Solution(v_1)
d9          for each egde (v_1, v_2) adjecent to v_1
d10           v_2 ← Augment(v_1, v_2)
d11           Insert v_2 into Queue
d12         endfor
d13       endif
d14     endwhile
d15     return A
Subroutine: ComputeA(T, G)
        T: Topology subtree
        G: Target routing graph
c1      for each vertex v ∈ G
c2        A_1^b ← Join(A_1.left, A_1.right)
c3        A_2^b ← Join(A_2.left, A_2.right)
c4      endfor
c5      A_1 ← GenDijkstra(A_1^b, G)
c6      A_2 ← GenDijkstra(A_2^b, G)
c7      for each vertex v ∈ G
c8        A_12^b ← Join(A_12.left, A_12.right) ∪ Join(A_1, A_2)
c9      endfor
c10     A_12 ← GenDijkstra(A_12^b, G)
c11     return A
Subroutine: ComputeAll(T,G)
        T: Topology subtree
        G: Target routing graph
b1      if(leaf(T))
b2        A(T) ← ComputeInitial(G)
b3      else
b4        A(T.left) ← ComputeAll(T.left, G)
b5        A(T.right) ← ComputeAll(T.right, G)
b6        A(T) ← ComputeA(T, G)
b7      endif
b8      return A(T)
Algorithm: S-Tree(T, G, s, S_1, S_2)
        T: Topology
        G: Target routing graph
        s: source node
        S_1, S_2: sink prtitions
a1      A(T) ← ComputeAll(T, G)
a2      Final ← AugmentForDriver(A(T, s))
a3      return Final
```

Figure 5: Basic S-Tree Algorithm.

$A_2(u, v)$: Similarly defined except limited to sinks in S_2.

$A_{12}(u, v)$: Similarly defined, but topologies must connect *all* sinks in the subtree.

We apply dynamic programming techniques to compute these sets in bottom-up fashion (an algorithm overview appears in Fig. 5). When all sets are established, the candidate solutions in $A_{12}(u_s, v_s)$ (where u_s is the root of the topology and v_s is the location of the driver in the target graph) are augmented to consider the effect of the driver (additional load-dependent delay) which then results in the overall set of non-dominated solutions with respect to cost p and required-time q. These form a tradeoff curve from which a solution can be selected.

In order to compute these sets it is useful to define a similar group of sets in which the vertex v in the target graph is constrained to be a branching point (basically, a Steiner). Let these sets be $A_1^b(u, v)$, $A_2^b(u, v)$ and $A_{12}^b(u, v)$.

Proceeding bottom-up in the topology, suppose we are at some vertex u; we visit each graph vertex v and first compute $A_1^b(u, v)$ and $A_2^b(u, v)$ by joining solutions from the

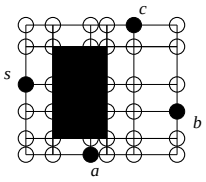

Figure 6: Target routing graph construction.

appropriate subtrees (a solution in $A_1^b(u,v)$ is formed by joining a solution in $A_1^b(l(u),v)$ with one from $A_1^b(r(u),v)$ where $l(u)$ and $r(u)$ are u's left and right children in T).

We then compute $A_{12}^b(u,v)$ by first joining $A_{12}(l(u),v)$ with $A_{12}(r(u),v)$ and then $A_1(u,v)$ with $A_2(u,v)$; the non-dominated solutions in the resulting union is retained. It is worth mentioning that in *Join* step (line c2 in Fig. 5) to guarantee optimality we have to construct a cross product of all costs p and join sets in respect of that cross product. Also pruning dominated solutions is not trivial any more. Methods and data-structures from [6] have been used.

As a preprocessing step we construct a graph on which topology will be embedded. We extend grid lines from the driver and every sink in all directions (4 in single layer, 3-6 in multilayer environment). Grid line intersections become candidates for branch points (Fig. 4). In the similar way we introduce grid lines that follow the contours of routing and buffer blockages (Fig. 6). Then we remove vertices and edges covered by routing blockages and mark vertices covered by buffer blockages as infeasible for buffer insertion.

Once we have a target graph and A^b set of branching solutions we must "augment" them to form "single-stem" solutions $A_1(u,v)$, $A_2(u,v)$ and $A_{12}(u,v)$. Such solutions eventually must reach a branch point computed in the previous step. This is accomplished efficiently through what can be viewed as a generalized version of Dijkstra's algorithm. We let the branching point solutions form the initial wavefront and expand in a manner similar to [11] and [4]. In order to do that we first introduce a virtual vertex x in target graph and add directed edges from that vertex to every other vertex labeling them with appropriate A^b solution (Fig. 7). So if we are generating solutions for vertex u in topology then the edge from x to v in target graph will have label $A^b(u,v)$. Starting from vertex x we update labels on all other vertices, insert them into priority queue, expand them in best cost first order and continue to update neighboring vertices until the queue becomes empty.

When we run our algorithm in trade-off mode, instead of a single 'best' solution we generate a set of non-dominated solutions, and instead of a single label we have a list of non-dominated labels which are expanded by best cost first.

3. DISCUSSION

Implementation Issues

Some important implementation details have not been mentioned in the interest of space. For instance, efficient determination of the dominance property can be non-trivial and the method and data-structures of [6] have been employed. In addition, sometimes a subtree's sinks are from only one set. Careful bookkeeping can exploit this situation to avoid redundant computations.

Initial Topology

Although initial topology is a parameter to S-Tree algorithm, it has to be generated somehow. In our implementation we used MST based approach and a P-Tree$_A$ topology, both discussed in [8].

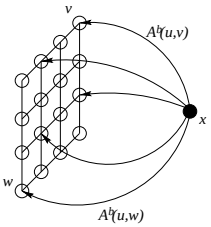

Figure 7: Illustration of generalized Dijkstra's algorithm. From vertex x directed edges with weight of $A^b(u,v)$ go to every vertex v in the target graph.

Speedup Techniques

It is often the case that a candidate solution, while non-dominated can be discarded by computing various lower-bounds or putting a threshold on the global cost allowable. We have incorporated such strategies and achieved substantial speedups. Also a buffer library pruning (discarding buffers which are dominated by other buffers) as a preprocesing step improves runtime.

Solution Quality

Careful choice of buffer candidate locations may improve solution quality. Segmenting long wires and/or inserting additional grid lines from buffer blockages may yield better solutions. These simple modifications do have impact on runtime but do not affect overall algorithm complexity.

Sink Partitioning

Currently we use simple heuristics to partition critical from non-critical sinks. We compute estimated delays from the source to each sink, adjust the given required time by these estimates and then rank the estimated achievable slacks. Partitioning with respect to sink polarity is trivial.

Generalizations

We have presented the algorithm where there are two sets of sinks. However there is no reason (except computational complexity as there is a run-time term which is exponential in the number of sets) that we cannot use three or more sets. In fact, in the limit where we have n singleton sets, the topology becomes irrelevant and we obtain optimality.

Modifications

There are a number of basically trivial modifications to the algorithm which are important in practice. For instance it is desirable to be able to handle inverters as buffers. This is easily done through previously studied techniques in buffer insertion. Also buffer insertion step is done in generalized Dijkstra's algorithm in a way similar to [5] which implicitly allows buffer cascading.

Algorithm Complexity

Due to limited space we constrain our algorithm complexity analysis to minimum wire length mode only. Assume that we have n sinks. Then topology still has $O(n)$ vertices including sinks. For every vertex u in topology and every vertex v in target graph we construct $A^b(u,v)$ and $A(u,v)$ set of solutions. Size of the target graph is $O(n^2)$ (if we have b blockages then it is $O((n+4b)^2)$). Computing $A^b(u,v)\ \forall v$

takes $O(n^2)$ time. Computing $A(u,v)\ \forall v$ involves running generalized Dijkstra's algorithm which runs in $O(E \log V)$ time where E is the number of edges and V number of vertices in the graph. In our case $E \approx O(n^2)$ (sparse graph) and $V \approx O(n^2)$ we have $(O(n^2 \log n^2))$ what is $O(n^2 \log n)$. Computing these sets for every u we have $O(n(n^2+n^2 \log n))$ what is $O(n^3 \log n)$. In trade-off mode we have an additional multiplicative factor because we do not keep a single solution for every (u,v) pair but a set of non-dominated solutions, yielding a pseudo-polynomial complexity.

4. EXPERIMENTS

We have implemented the S-Tree algorithm and performed some initial experiments in Linux environment on a PC with a 766MHz CPU. The three main criteria of interest are (1) run-time scalability, (2) solution quality in terms of required-time and (3) solution quality in terms of cost (wire length and buffer usage). Our experiments are compared with a P-Tree based approach [7] which is known to produce high quality solutions particularly for uniform required times.

In our experiments we used two different ways to derive the initial topology fed to S-Tree. First is induced by the minimum spanning tree (same tree used to construct initial permutation in P-Tree [8]). For the other we used the P-Tree$_A$ (wire-length only) algorithm to derive the initial topologies (because of the pseudo-polynomial nature of timing-driven versions of the various algorithms, this is reasonable as S-Tree still dominates the overall run-time). In test results the first method is referred to as S-Tree, and the second one as S-Tree+ (the runtime of P-Tree$_A$ is included). Wire length is reported in microns, slack in pico seconds, and execution time in seconds of CPU time.

We compared performance of all 3 timing driven algorithms with simultaneous tree construction and buffer insertion on 2 sets of nets. Nets in the first set (6-24 'a' and 'b') have uniform random distribution of required arrival times for sinks (Tab. 1). This is the unfavorable setup for S-Tree since P-Tree's larger solution space (which in general does not cover S-Tree's solution space) allows more exploration when it is not obvious which paths will become critical at the end. S-Tree was able to achieve maximum slack very close to the one of P-Tree on all nets using same number of buffers, less wire, and an order of magnitude less cpu time. On net 6b it gave exactly the same solution and on net 12a slightly faster one. Only on net 12b, 3% more wire was used, but solution was generated 4 times faster. For net sizes 21 and 24, P-Tree was not able to give buffered solutions in reasonable amount of time while S-Tree was able to complete and meet sink timing requirements.

Second set of nets (6-16 crit) contains a few highly critical sinks among non-critical ones (Tab. 2). Here we compared S-Tree against P-Tree (both with buffer insertion) on how much resources they require to achieve given slack. For a better understanding of wire usage, minimum wire length achievable by both algorithms is also included. S-Tree is a clear winner in all 3 dimensions. It was able to achieve same slack using less wire, less buffers and using an order of magnitude less cpu time.

5. EXAMPLE

To demonstrate how S-Tree works in practice we have constructed a simple example instance. Some of the solution

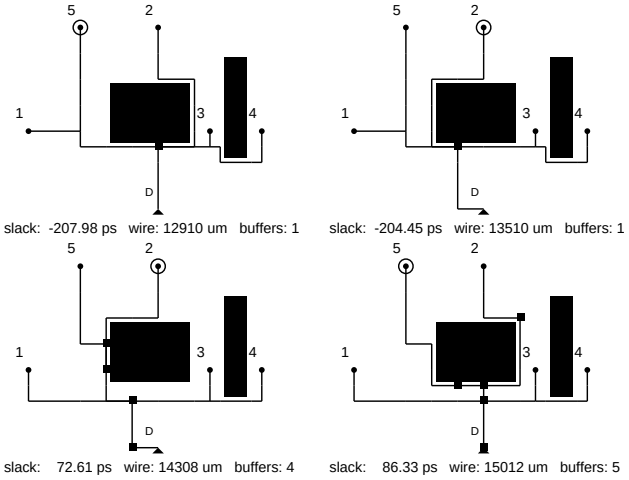

Figure 8: Example topology.

topologies are shown in Figure 8. Driver is represented by a triangle, sinks by circles and inserted buffers by squares. Sink that is critical for that particular solution is circled. Also each solution's slack is reported in pico seconds, total wire length in microns, and total number of used buffers. In this example, due to timing requirements sinks are partitioned in the following way: $S_1 = \{1,3,4\}$ and $S_2 = \{2,5\}$. Solution on the top left obeys initial topology, which is in this case constructed using geometric MST approach. As the solutions move on trade-off curve towards more expensive (and faster) ones, it can be clearly seen how that initial topology "breaks" and at the end we have sinks 2 and 5 grouped in the same subtree, isolated from the others. Also note how current critical sink changes from one solution to the other.

6. CONCLUSIONS

We have presented the S-Tree algorithm for synthesis of buffered interconnects. The approach incorporates a unique combination real-world issues (handling of routing and buffer blockages, cost minimization, critical sink isolation, sink polarities), robustness and scalability. The algorithm is able to achieve the slack comparable to that of buffered P-Tree using less resources (wire and buffers) in an order of magnitude less cpu time. The saving on resources becomes even larger for nets with non-uniform timing requirements.

7. REFERENCES

[1] C. Alpert, et. al. "Buffered Steiner Trees for Difficult Instances," ISPD-01, pp. 4-9.

[2] J. Cong, "Challenges and Opportunities for Design Innovations in Nanometer Technologies," *Frontiers in Semiconductor Research: A Collection of SRC Working Papers.* SRC, 1997.

[3] J. Cong, X. Yuan, "Routing Tree Construction Under Fixed Buffer Locations," DAC-2000, pp. 368-373.

[4] S.-W. Hur, A. Jagannathan, J. Lillis, "Timing-Driven Maze Routing," *IEEE Transactions on Computer Aided Design* Feb. 2000, vol. 19, no. 2, pp. 234-241.

[5] A. Jagannathan, S.-W. Hur, J. Lillis, "A Fast Algorithm for Context-Aware Buffer Insertion," DAC-2000, pp. 368-373.

Table 1: Performance of different algorithms on 6-24 pin nets

	net 6a							net 6b						
Alg.	0 buf		3 buf		5 buf		cpu	0 buf		1 buf		3 buf		cpu
	wl	slk	wl	slk	wl	slk		wl	slk	wl	slk	wl	slk	
PTree	18034	-682	20310	517	20014	608	0.126	20440	-945	25396	183	17902	488	0.071
STree	17799	-646	19397	541	19718	606	0.045	20440	-945	17902	174	17902	488	0.033
STree+	17799	-646	19397	541	19718	606	0.045	20440	-945	17902	174	17902	488	0.033

	net 9a							net 9b						
Alg.	0 buf		4 buf		9 buf		cpu	0 buf		3 buf		6 buf		cpu
	wl	slk	wl	slk	wl	slk		wl	slk	wl	slk	wl	slk	
PTree	22157	-941	28917	605	37468	704	1.943	29819	-2891	40156	-14	49972	83	1.336
STree	22157	-941	28508	575	32859	696	0.411	34401	-3495	34282	-117	30128	48	0.528
STree+	23764	-946	26830	598	32859	696	0.357	34401	-3495	30128	-161	29450	23	0.554

	net 12a							net 12b						
Alg.	0 buf		3 buf		7 buf		cpu	0 buf		5 buf		8 buf		cpu
	wl	slk	wl	slk	wl	slk		wl	slk	wl	slk	wl	slk	
PTree	36220	-2947	32173	-110	50694	299	11.98	31714	-2245	28664	415	30370	461	9.000
STree	35703	-2929	36949	38	36949	303	1.529	31382	-2849	35959	294	36125	435	2.251
STree+	35703	-2929	36949	38	36949	303	1.543	31145	-2684	29549	322	31654	440	2.052

	net 15a							net 15b						
Alg.	0 buf		5 buf		8 buf		cpu	0 buf		3 buf		7 buf		cpu
	wl	slk	wl	slk	wl	slk		wl	slk	wl	slk	wl	slk	
PTree	41410	-3650	46366	312	49111	399	58.69	32076	-1839	36306	525	45358	652	45.83
STree	49424	-3451	39501	209	46802	385	6.283	35303	-2009	43003	487	46691	641	7.041
STree+	52536	-3638	38701	134	45731	353	7.481	33980	-1930	42843	487	42700	641	6.813

	net 18a							net 18b						
Alg.	0 buf		5 buf		11 buf		cpu	0 buf		5 buf		8 buf		cpu
	wl	slk	wl	slk	wl	slk		wl	slk	wl	slk	wl	slk	
PTree	42817	-3650	48972	265	53062	464	209.6	41265	-3544	49166	120	54590	181	187.4
STree	35325	-3784	43696	62	49776	450	14.28	43004	-3757	46037	36	49840	180	17.01
STree+	35287	-3760	43696	62	49776	450	14.72	42384	-3720	48283	32	47738	177	17.98

	net 21a							net 21b						
Alg.	0 buf		3 buf		7 buf		cpu	0 buf		5 buf		9 buf		cpu
	wl	slk	wl	slk	wl	slk		wl	slk	wl	slk	wl	slk	
PTree	-	-	-	-	-	-	-	-	-	-	-	-	-	-
STree	61769	-3376	49528	159	50387	783	46.14	55245	-4478	47434	29	45281	183	50.37
STree+	45708	-2836	45480	409	47421	1137	38.75	49930	-4094	48422	61	49898	184	38.89

	net 24a							net 24b						
Alg.	0 buf		5 buf		11 buf		cpu	0 buf		6 buf		12 buf		cpu
	wl	slk	wl	slk	wl	slk		wl	slk	wl	slk	wl	slk	
PTree	-	-	-	-	-	-	-	-	-	-	-	-	-	-
STree	69602	-5960	62186	-146	62176	143	84.66	56775	-4385	64101	99	60947	305	76.32
STree+	69602	-5960	60760	-482	60094	-292	101.24	52377	-4254	62822	-98	62891	239	77.93

Table 2: Usage of resources for critical nets PTree vs STree

nets	slack	minWL	PTree			STree		
			wl	buf	cpu	wl	buf	cpu
6crit	521	10500	15500	6	0.026	10500	5	0.011
8crit	680	7000	10000	6	0.027	10000	4	0.004
10crit	673	10000	18000	6	0.125	14000	5	0.026
12crit	784	11000	18000	9	0.643	15000	5	0.033
14crit	614	12000	20000	7	0.702	18000	5	0.059
16crit	1567	14400	23200	7	4.982	21200	6	0.382

[6] J. Lillis, C.-K. Cheng, T.-T. Y Lin, "Optimal Wire Sizing and Buffer insertion for Low Power and a Generalized Delay Model," *IEEE Journal of Solid State Circuits*, 31 (3): pp. 437-447, March 1996.

[7] J. Lillis, C.-K. Cheng, T.-T. Y. Lin, "Simultaneous Routing and Buffer Insertion for High Performance Interconnect," *Proc.* 6^{th} *IEEE Great Lakes Symposium on VLSI*, Ames, Iowa, Mar. 1996, pp. 148-153.

[8] J. Lillis, C.-K. Cheng, T.-T. Y Lin, "New Performance Driven Routing Techniques With Explicit Area/Delay Tradeoff and Simultaneous Wire Sizing," DAC-96.

[9] T. Okamoto, J. Cong, "Buffered Steiner Tree Construction with Wire Sizing for Interconnect Layout Optimization," ICCAD-96, pp. 44-49, 1996.

[10] L.P.P.P. van Ginneken, "Buffer Placement in Distributed RC-tree Networks for Minimal Elmore Delay," ISCAS-90, pp. 865-868, 1990.

[11] H. Zhou, D.F. Wong, I.M. Liu, A. Aziz, "Simultaneous Routing and Buffer Insertion with Restrictions on Buffer Locations," DAC-99, pp. 96-99.