

# Buffer Tree Synthesis With Consideration of Temporal Locality, Sink Polarity Requirements, Solution Cost and Blockages

Milos Hrkic  
mhrkic@cs.uic.edu

John Lillis  
jlillis@cs.uic.edu

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

## ABSTRACT

We give an overview of a buffer tree synthesis package which pays particular attention to the following issues: routing and buffer blockages, minimization of interconnect and buffer costs, exploitation of temporal locality among the sinks and addressing sink polarity requirements. Experimental results demonstrate the effectiveness of the tool in comparison with previously proposed techniques.

## Categories and Subject Descriptors

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

## General Terms

Algorithms

## 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*, *topology* and *topology embedding*. The past ten years has seen the growth of a 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 [12] which proposed a dynamic programming algorithm for inserting buffers into a *given* rooted topology. In addition, buffer insertion techniques for two-pin nets have received some attention ([13], [7]). In the area of routing topology construction, there are several works of particular relevance to this paper. These include the *P-Tree* based methods [10], methods for timing driven routing of two-pin nets (including blockages) [6], [13], methods which combine buffer insertion and topology construction ([9], [11], [3]) and a recently proposed *S-Tree* method [4] which explicitly incorporates a notion of *temporal locality* among sinks.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISPD'02, April 7-10, 2002, San Diego, California, USA.

Copyright 2002 ACM 1-58113-460-6/02/0004 ...\$5.00.

In this paper we introduce the SP-Tree algorithm with a number of practical criteria and objectives. An overview of the issues emphasized is as follows.

### *Simultaneous Buffer Insertion and Tree Construction*

We believe that overall improvements in solution quality can be achieved by not viewing the interconnect synthesis as a two phase process (routing construction followed by buffer insertion) but as a simultaneous approach. While some past works have attempted such a unification ([9], [3]), it now appears that substantial improvements are possible.

### *Handling Routing and Buffer Blockages*

In real designs there are often 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 as our routing target.

### *Temporal Locality and Sink Polarity Requirements*

A potential weakness of some topology constructions is that they are oblivious to sink criticality. For example in the P-Tree method [10], 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 sink criticality (*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 allow some 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 (some sinks expecting an inverted signal). Two recent papers have addressed one or both of these issues. In [1], notions of temporal locality and polarity requirements were incorporated into a sink clustering algorithm (using a heuristic “similarity metric”) as part of a topology construction procedure; the resulting topology was handed to a fixed-topology buffer insertion tool. In [4], it was argued that notions of temporal and physical locality should be separated if robust and predictable behavior is to be expected; the result was the S-Tree algorithm which captured physical locality through a given topology and temporal locality through a sink partitioning. Together these two items created a generalized topology space capturing both requirements. Additionally simultaneous exploration of topology, embedding and buffering was performed.

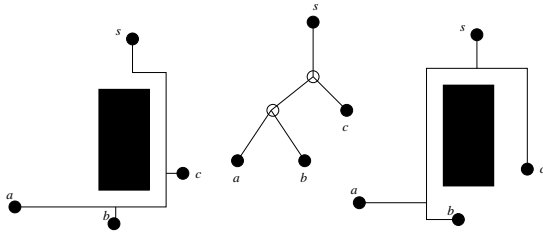


Figure 1: A topology for a 3-sink net and two physical embeddings of that topology. Pin  $s$  is the driver.

### 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 *SP-Tree*. The tool adopts the P-Tree idea of using a sink permutation to capture physical locality and the sink partitioning idea (or *stitching*) of the S-Tree. There are three degrees of freedom exploited by SP-Tree: *Topology*, *Embedding* (placement of Steiner nodes in a target graph) and *Buffer placement*. We first focus primarily on the solution space covered by the algorithm and then sketch some of the implementation details.

## 2. SOLUTION SPACE DESCRIPTION

To understand the SP-Tree topology space, it is instructive to examine the S-Tree and P-Tree spaces individually.

### 2.1 The S-Tree Space

The S-Tree algorithm uses a given topology to capture physical locality of sinks. Ignoring temporal locality for the moment, Fig. 1 illustrates the embedding space for a given topology where we show two possible embeddings. Even though the topology is fixed, there is clearly some flexibility in the embedding which may be useful particularly in timing driven applications. This flexibility however is 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).

In the S-Tree algorithm, besides a topology  $T$ , we are also given a partitioning of the sinks into two disjoint sets  $S_1$  and  $S_2$ . Now consider a subtree in  $T$  rooted at vertex  $u$  and with left and right subtrees  $l(u)$  and  $r(u)$ . Some sinks in the subtree belong to  $S_1$  and others to  $S_2$ . If we have two sets of topologies  $L$  and  $R$ , then  $L \times R$  is the set of all topologies where a root has a member of  $L$  as its left subtree and a member of  $R$  as its right subtree (basically a cross-product; if one of the sets is empty, the output is the other 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

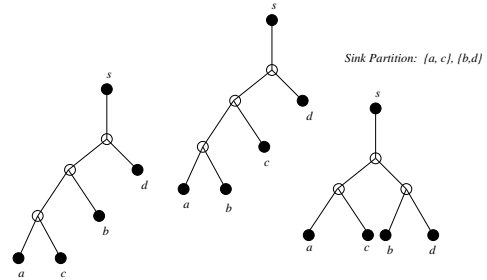


Figure 2: S-Tree topology solution space example.

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:

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

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 4-sink topology with a given sink partitioning. Naturally, the given topology is included in the space in addition to the other two shown. Note that while the left topology is isomorphic to the given topology, some of the sinks have been re-labeled ( $b$  now being closer to the root and  $c$  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. Topology and a sink partition is given and sink  $b$  is critical. It is likely that we desire a direct path from  $s$  to  $b$  which decouples all off-path capacitance. For the given topology, the best we can do is illustrated on the lower-left. In S-Tree, the solution on the lower-right becomes possible. Besides having lower delay to sink  $b$ , it also uses just one buffer.

Given this notion of the S-Tree space, the algorithm optimally solves the following problem: Given technology parameters, 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., some function of buffers, area, wire length) subject to timing constraints being met.

### 2.2 The P-Tree Space

The P-Tree algorithm [10] achieves a high degree of flexibility (enabling an exponential number of topologies) by constraining the topology to be induced by a sink permutation. This sink permutation is constructed in a way that captures physical locality between sinks; thus consecutive

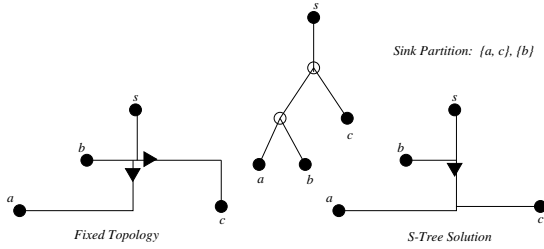


Figure 3: Illustration of critical sink isolation and buffer savings in S-Tree.

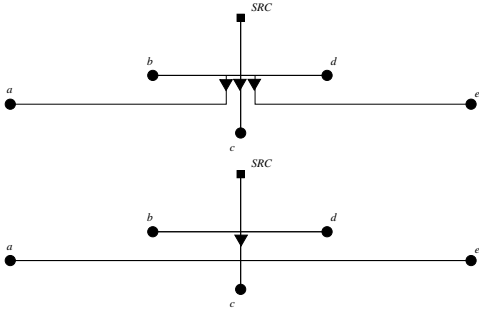


Figure 4: P-Tree's inability to capture solution of smaller cost. Critical sinks are  $b$  and  $d$ . Sink permutation is  $\{a, b, c, d, e\}$ , sink partition is  $\{a, c, e\}, \{b, d\}$ ; proposed approach captures solution on bottom.

subsequences in the permutation are likely to make good candidate subtrees.

Since the permutation is determined solely by physical locality, we observe similar phenomenon as in the case of fixed topology embedding: the ability to effectively isolate critical/non-critical groups of sinks is limited (similarly, the ability to separate sets of sinks with differing polarity requirements) is poor. For example, consider Fig 4. We see that the critical sinks are “pinched” by the given sink permutation and it requires three buffers to de-couple the non-critical sinks. On the other hand a different topology outside the P-Tree space would enable a single buffer to give equal or better performance as shown in the lower solution.

At first glance, it is appealing to adopt a different similarity metric (as in [1]) as a guide for constructing the sink permutation to alleviate this problem. However, our experience indicates that the heuristic measures one might attempt lack the predictability necessary for a robust approach. Instead, we have adopted the stitching idea of S-Tree whereby sink criticality (and now polarity requirements) are captured through the orthogonal notion of sink partitions, while the sink permutation continues to capture the physical sink locality. The resulting generalization of P-Tree has been dubbed SP-Tree.

Our current framework is more general than that of S-Tree in that we enable the partitioning of sinks into four sets (if appropriate) to consider sink polarity requirements: **critical/positive**, **critical/negative**, **non-critical/positive**, **non-critical/negative**. This generalization has been applied to the S-Tree framework as well.

Before discussing algorithmic details, we clarify the objectives of SP-Tree vs. those of S-Tree. The main advantage S-Tree has is scalability while the main advantage of SP-

Tree is solution space coverage. The goal then of SP-Tree is to provide excellent solution quality for modest sized nets which represent a large percentage of those in practice (e.g., up to 10 or 12 sinks). On the other hand, S-Tree (or some hybrid approach) may be used for larger fanout nets.

### 3. ALGORITHMIC OVERVIEW

We first discuss the construction of the target routing graphs to capture blockages; for clarity we then present the algorithms *in the absence of the stitching operation* (“without sets”); an overview of a generalization of Dijkstra’s algorithm is discussed (this enables the natural handling of general graph); finally we discuss how the stitching idea can be incorporated into this framework. For details see [5].

#### 3.1 Target Graph Construction

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. In the similar way we introduce grid lines that follow the contours of routing and buffer blockages. Then we remove vertices and edges covered by routing blockages and mark vertices covered by buffer blockages as infeasible for buffer insertion.

#### 3.2 Solutions for the Non-Stitching Cases

For ease of presentation, we first describe the dynamic programming decomposition used in S-Tree and P-Tree without stitching which is similar in flavor to several past works ([8], [10]). 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 only  $c$  and  $q$  are necessary to assure a maximum  $q$  solution ([3]), 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; experimental evidence supports this belief.

In S-Tree we define  $A(u, v)$  to be the set of non-dominated solutions for subtree rooted at  $u$  in given topology  $T$ , with root placed at  $v$  in  $G$ , connecting all sinks in that subtree. In P-Tree we define  $A(i, j, v)$  to be the set of non-dominated solutions over all permutation induced routing topologies driving sinks  $i$  through  $j$  and rooted at  $v$ .

We apply dynamic programming techniques to compute these sets in bottom-up fashion (an overview of the algorithm for the more complex P-Tree case appears in Fig. 5).

When all sets are established, the candidate solutions in  $A(u_s, v_s)$  for the S-Tree case and  $A(1, n, v_s)$  in the P-Tree case 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 related group of intermediate sets in which the vertex  $v$  in the target graph is constrained to be a branching point (basically, a Steiner). Let these sets be  $A^b(u, v)$  in S-Tree vari-

```

Subroutine: JoinPTree( $i, j, G$ )
   $i, j$ : Subsolution covering sinks from  $i$  to  $j$ 
   $G$ : Target routing graph
f1  $A^b \leftarrow \emptyset$ 
f2 for each vertex  $v \in G$ 
f3   for  $k = i$  to  $j-1$ 
f4      $A^b[i][j][v] \leftarrow A^b[i][j][v] \cup \text{Join}(A[i][k][v], A[k+1][j][v])$ 
f5   endfor
f6 endfor
f7  $A[i][j] \leftarrow \text{GenDijkstra}(A^b[i][j], G)$ 
f8 return  $A$ 

Algorithm: P-Tree( $P, G, s, n$ )
   $P$ : Sink permutation;  $G$ : Target routing graph
   $s$ : source node;  $n$ : number of sinks
e1 for  $i = 1$  to  $n$  do  $A[i][i] \leftarrow \text{Initial}(P, G, i)$ 
e2 for  $\text{gap} = 1$  to  $n-1$ 
e3   for  $i = 1$  to  $n-\text{gap}$ 
e4      $A[i][i+\text{gap}] \leftarrow \text{JoinPTree}(i, i+\text{gap}, G)$ 
e5   endfor
e6 endfor
e7  $\text{Final} \leftarrow \text{AugmentForDriver}(A[1][n], s)$ 
e8 return  $\text{Final}$ 

```

Figure 5: Basic P-Tree Algorithm.

ant and  $A^b(i, j, v)$  in P-Tree variant. Subsequently we will refer to these sets simply as  $A(\cdot)$  and  $A^b(\cdot)$  when the discussion applies to both the S-Tree and P-Tree cases. These branching solutions  $A^b(\cdot)$  are computed from the appropriate previously computed single-stem solutions ( $A(\cdot)$ ) via a *Join* operation as illustrated in the figure for the P-Tree.

It is worth mentioning that in *Join* step (line f4 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.

### 3.3 Generalized Dijkstra’s Algorithm

In the previous P-Tree work, the fact that the target graph did not contain blockages was exploited and the single-stem solutions ( $A(\cdot)$ ) could be computed in a sequence of four sweeps of the grid to augment the branching solutions.

In the presence of blockages, the problem becomes more difficult. We propose that it can be solved efficiently through what can be viewed as a generalized version of Dijkstra’s algorithm. We let the branching point solutions form the initial wavefront (note that the wavefront entries are triples  $(p, c, q)$  not scalars as in traditional shortest paths) and expand in a manner similar to [13] and [6]. To do that we introduce a virtual vertex  $x$  in target graph and add directed edges from  $x$  to every other vertex labeling them with appropriate  $A^b(\cdot)$  solutions. 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. Instead of a single label we maintain a list of non-dominated labels.

### 3.4 Incorporating “Stitching”

As stated earlier, in addition to the ability to deal with blockages, one of the contributions of this work is the notion of using sink partitions to capture *temporal* and/or *polarity* locality and expand the solution space accordingly.

Suppose we have two sink partitions. We then define the following generalizations of the  $A(\cdot)$  sets. We state the sets for the SP-Tree case; the S-Tree sets are analogous.

$A_1(i, j, v)$ : the set of non-dominated solutions with root embedded at  $v$  in  $G$  and connecting *only* sinks in the intersection of  $S_1$  and sinks  $i..j$ .

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

$A_{12}(i, j, v)$ : Similarly defined, but topologies must con-

```

Subroutine: JoinSPTree( $i, j, G$ )
   $i, j$ : sinks  $i$  to  $j$  to be connected by subsolution
   $G$ : Target routing graph
f1  $A^b \leftarrow \emptyset$ 
f2 for each vertex  $v \in G$ 
f3   for  $k = i$  to  $j-1$ 
f4      $A_1^b[i][j][v] \leftarrow A_1^b[i][j][v] \cup \text{Join}(A_1[i][k][v], A_1[k+1][j][v])$ 
f5      $A_2^b[i][j][v] \leftarrow A_2^b[i][j][v] \cup \text{Join}(A_2[i][k][v], A_2[k+1][j][v])$ 
f6   endfor
f7 endfor
f8  $A_1[i][j] \leftarrow \text{GenDijkstra}(A_1^b[i][j], G)$ 
f9  $A_2[i][j] \leftarrow \text{GenDijkstra}(A_2^b[i][j], G)$ 
f10 for each vertex  $v \in G$ 
f11   for  $k = i$  to  $j-1$ 
f12      $\text{tmp} \leftarrow \text{Join}(A_{12}[i][k][v], A_{12}[k+1][j][v])$ 
f13      $A_{12}^b[i][j][v] \leftarrow A_{12}^b[i][j][v] \cup \text{tmp}$ 
f14   endfor
f15  $A_{12}^b[i][j][v] \leftarrow A_{12}^b[i][j][v] \cup \text{Join}(A_1[i][j][v], A_2[i][j][v])$ 
f16 endfor
f17  $A_{12}[i][j] \leftarrow \text{GenDijkstra}(A_{12}^b[i][j], G)$ 
f18 return  $A$ 

```

Figure 6: SP-Tree Join with two sink partitions

nect sinks in  $S_1 \cup S_2$  contained in  $i..j$  which in this case represent *all* sinks in the  $i..j$ .

To compute these sets we also define respective branching solutions  $A_1^b(\cdot)$ ,  $A_2^b(\cdot)$  and  $A_{12}^b(\cdot)$ . Proceeding bottom-up we compute sets  $A_1(\cdot)$ ,  $A_2(\cdot)$ , and  $A_{12}(\cdot)$  in the same way as explained previously with one difference in computing sets  $A_{12}(i, j, v)$  (this is where the stitching comes in). Solutions in  $A_{12}^b(i, j, v)$  may be constructed first by *joining*  $A_{12}(i, k, v)$  with  $A_{12}(k+1, j, v)$  for some  $k$  or from joining  $A_1(i, j, v)$  with  $A_2(i, j, v)$  (these are the “stitched” solutions). The non-dominated solutions in the resulting union are retained. In this way we separate critical/non-critical sub-trees (and similarly when considering sink polarities). Pseudocode for modified **JoinSPTree** subroutine is given in Fig. 6.

#### 3.4.1 Generalizations

We have presented algorithms 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 permutation becomes irrelevant and we obtain optimality.

We found two more set partitioning schemes to be of particular interest. We can partition sinks as non-critical, “not sure” if critical, and highly critical. We then define the following sets:  $A_1, A_2, A_3, A_{12}, A_{23}, A_{123}$ . Observe that we do not join solutions that spawn highly critical and non-critical sinks ( $A_{13}$ ) for obvious reasons.

The other case of practical interest is with 4 partitions: ( $S_1$ ) positive polarity constraints and critical, ( $S_2$ ) positive and non-critical, ( $S_3$ ) negative and non-critical, ( $S_4$ ) negative and critical. Here we define following nine sets:  $A_1, A_2, A_3, A_4, A_{12}, A_{23}, A_{34}, A_{14}, A_{1234}$  in the similar way. Observe that in “intermediate” sets we join all critical sets, all non-critical sets, all positive polarity sets, and all negative polarity sets. In this way we are able to simultaneously capture *physical*, *temporal* and *polarity* locality.

## 4. DISCUSSION

### Implementation Issues

Some important implementation details have not been mentioned in the interest of space. Efficient determination of the dominance property can be non-trivial and the method

and data-structures of [8] have been employed. Sometimes a subtree’s sinks are from only one set. Careful bookkeeping can exploit this and avoid redundant computations.

### 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.

### Modifications

There are a number of trivial modifications to the algorithm which are important in practice. Inverter handling is done through previously studied techniques. Buffer insertion step is done in generalized Dijkstra’s algorithm in a way similar to [7], implicitly allowing buffer cascading.

## 5. EXPERIMENTS

We have implemented the SP-Tree algorithm and performed some initial experiments in Solaris environment on a Sun Ultra 1 workstation with a 200MHz CPU and 384MB of RAM<sup>1</sup>. The main criteria of interest are solution quality in terms of both slack and cost (wire length and buffer usage) and run-time. Our experiments are compared with a P-Tree based approach [9] which is known to produce high quality solutions particularly for uniform required times, S-Tree [4] and RMP [3]. For RMP we also reported results for “Quick” running mode where heuristics are used to reduce CPU time. Wire length is reported in microns, slack in pico seconds, and execution time in seconds of CPU time. Technology parameters are representative of 0.25 $\mu$ m technology. For each net three solutions are shown: minimum cost solution, minimum cost feasible solution (cheapest with positive slack), and maximum slack solution. Runs that failed to report result in 30 minutes were terminated. Maximum amount of memory used by S/P-Tree algorithms was 83MB. Solution cost in S-Tree, P-Tree and SP-Tree algorithms is defined as a function of wire capacitance (which is proportional to wire length) and input buffer capacitance (in absence of actual buffer cost) ( $WireCap + \beta \cdot InBufCap$ ). This enables arbitrary kind of cost normalization (e.g., bias toward minimizing buffers or wire length).

In first set of experiments we used randomly generated nets with small variations in sink required arrival time (Tab. 1). Also we used small value for  $\beta$  ( $\beta = 2$ ), what means that buffers are relatively inexpensive. This experimental setup shows that when there are no variations in *temporal* or *polarity* locality between sinks, algorithm that considers only *physical* locality (P-Tree) is sufficient if the goal is to achieve high quality solution (i.e. good slack and small cost). If the goal is to achieve good solution quickly then S-Tree is a good choice, since its smaller solutions space is compensated by “stitching” to fix initial topology “bad” decisions if any. Also it is clear that in any complex design, algorithm that tries to maximize slack without paying any attention to solution cost is not of much use for interconnect construction.

In second set of experiments we allowed large variations in sink required arrival times (Tab. 2). We made buffers very expensive ( $\beta = 200$ ). This allows more wire detours

<sup>1</sup>For reference, an 800MHz Celeron is roughly 3.5 times faster than this machine which was used for compatibility with the RMP executable

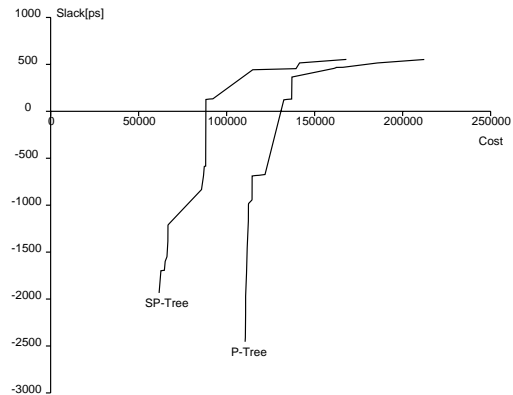


Figure 7: Cost/performance trade-off curve for P-Tree and SP-Tree on net3-10.

in searching for better solution. Algorithms that consider both *temporal* and *spatial* locality give cheaper solutions of the same slack. Although S-Tree is significantly faster, SP-Tree’s larger solution space makes it more robust. Again, algorithms that do not consider solution cost produce solutions that are not very likely to be used in practice.

To demonstrate the full power of SP-Tree, in third set of experiments we included sink input polarity constraints (Tab. 3). Buffer library now contains buffers and inverters. Data for RMP is not included since the implementation we were able to obtain does not support inverters and varying sink polarity constraints. Benefits of stitching idea are even larger for those “difficult” nets. On cost/performance trade-off curve (Fig. 7) it could be seen that SP-Tree gives much cheaper solutions than P-Tree for fixed slack, although solutions do converge to the same max slack solution.

## 6. CONCLUSIONS

We have presented the SP-Tree algorithm for synthesis of buffered interconnects. The approach incorporates a unique combination real-world issues (routing and buffer blockages, cost minimization, critical sink isolation, sink polarities), while appearing to provide predictably good solutions.

The ability to handle blockages in a uniform way and the ability to meet timing and polarity requirements while conserving buffering and wiring resources are the most important contributions of the work. Its effectiveness vs. previous approaches has been experimentally verified.

## 7. ACKNOWLEDGMENTS

The authors wish to thank X. Yuan and J. Cong for providing the RMP executable.

## 8. 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] M. Hrkic, J. Lillis, “S-Tree: A Technique for Buffered Routing Tree Synthesis” SASIMI-2001, pp. 242-249.

**Table 1: 6-12 pin nets, uniform required arrival time**

Net	Alg.	min cost				min cost feasible				max slack				cpu
		wl	slk	buf	cost	wl	slk	buf	cost	wl	slk	buf	cost	
net 1-06	RMP Quick	20766	12	9	23160	20766	12	9	23160	20766	12	9	23160	0.1
	RMP	22067	97	10	24727	22067	97	10	24727	22067	97	10	24727	3.9
	S-Tree	17991	-2493	0	17991	17991	16	5	19321	24495	120	7	26357	0.3
	P-Tree	17991	-2493	0	17991	17991	16	5	19321	22128	123	7	23990	0.6
	SP-Tree	17991	-2493	0	17991	17991	16	5	19321	22128	123	7	23990	1.1
net 1-08	RMP Quick	30054	483	17	34576	30054	483	17	34576	33244	510	17	37766	1.4
	RMP	32183	573	18	36971	32183	537	18	36971	32183	537	18	36971	6.77
	S-Tree	22894	-2551	0	22894	22894	67	4	23958	30276	537	12	33468	1.7
	P-Tree	21956	-4989	0	21956	22495	162	4	23559	29745	541	12	32937	7.1
	SP-Tree	21956	-4989	0	21956	22495	162	4	23559	29745	541	12	32937	16.9
net 1-10	RMP Quick	33448	353	23	39566	33448	353	23	39566	33448	353	23	39566	104
	RMP	-	-	-	-	-	-	-	-	-	-	-	-	-
	S-Tree	25691	-4314	0	25691	25911	2	5	27241	26860	418	15	30850	5.7
	P-Tree	25340	-4090	0	25340	25340	47	5	26670	27415	426	14	31139	40
	SP-Tree	25340	-4090	0	25340	25340	47	5	26670	27415	426	14	31139	109
net 1-12	RMP Quick	50741	555	32	59253	50741	555	32	59253	50741	555	32	59253	1096
	RMP	-	-	-	-	-	-	-	-	-	-	-	-	-
	S-Tree	25739	-5799	0	25739	25739	69	7	27601	37611	645	16	41867	24
	P-Tree	24970	-5650	0	24970	25445	118	5	26775	39870	648	20	45190	295
	SP-Tree	24970	-5650	0	24970	25445	118	5	26775	39870	648	20	45190	687

**Table 2: 6-12 pin nets, non-uniform required arrival time, buffer-biased cost**

Net	Alg.	min cost				min cost feasible				max slack				cpu
		wl	slk	buf	cost	wl	slk	buf	cost	wl	slk	buf	cost	
net 2-06	RMP Quick	18288	-176	8	231088	-	-	-	-	22664	-148	10	288664	0.1
	RMP	23001	-6	12	342201	-	-	-	-	23001	-6	12	342201	5.4
	S-Tree	16177	-2402	0	16177	22478	12	8	235278	22478	12	8	235278	0.3
	P-Tree	16177	-2402	0	16177	17867	11	7	204067	22478	19	8	235278	0.8
	SP-Tree	16177	-2402	0	16177	24168	3	6	183768	24168	19	7	210368	1.7
net 2-08	RMP Quick	24459	825	13	370259	24459	825	13	370259	24938	838	14	397338	1.1
	RMP	27328	907	15	426328	27328	907	15	426328	26808	925	16	452408	742
	S-Tree	20857	-1461	0	20857	25748	125	1	52348	29719	944	6	189319	2.7
	P-Tree	20340	-1413	0	20340	21617	13	1	48217	23279	944	7	209479	30
	SP-Tree	20340	-1413	0	20340	21617	13	1	48217	27884	944	6	187484	61
net 2-10	RMP Quick	27840	307	19	533240	27840	307	19	533240	27840	307	19	533240	48
	RMP	-	-	-	-	-	-	-	-	-	-	-	-	-
	S-Tree	18896	-1490	0	18896	19738	156	2	72938	21523	376	7	207723	28
	P-Tree	18428	-2114	0	18428	18428	160	3	98228	20998	381	8	233798	145
	SP-Tree	18428	-2114	0	18428	19355	156	2	72555	20998	381	8	233798	297
net 2-12	RMP Quick	35743	1630	23	647543	35743	1630	23	647543	36782	1636	25	701782	1801
	RMP	-	-	-	-	-	-	-	-	-	-	-	-	-
	S-Tree	23213	-1684	0	23213	23213	207	1	49813	33150	1704	11	325750	106
	P-Tree	22585	-1529	0	22585	22585	224	1	49185	32028	1711	11	324628	674
	SP-Tree	22585	-1529	0	22585	22585	224	1	49185	36149	1712	9	275549	1579

**Table 3: 6-10 pin nets, non-uniform required arrival time, polarity and buffer-biased cost**

Net	Alg.	min cost				min cost feasible				max slack				cpu
		wl	slk	buf	cost	wl	slk	buf	cost	wl	slk	buf	cost	
net 3-06	S-Tree	24289	-1037	1	50899	18047	160	3	97847	18127	650	9	257537	0.5
	P-Tree	16253	-1059	2	69453	22026	13	3	101826	20082	660	8	232882	2.4
	SP-Tree	24289	-1037	1	50889	25855	11	2	80485	20082	660	8	232882	4.1
net 3-08	S-Tree	31864	-590	1	58464	31864	469	2	85064	32743	1419	14	405143	3.3
	P-Tree	26953	-791	2	80153	22962	382	3	102762	31474	1422	14	403874	23
	SP-Tree	31864	-590	1	58464	31864	469	2	85064	31474	1422	14	403874	62
net 3-10	S-Tree	35030	-1935	1	61630	35030	126	2	88230	35030	553	5	168030	24
	P-Tree	30701	-2455	3	110501	26099	123	4	132499	26099	552	7	212299	147
	SP-Tree	35030	-1935	1	61630	35030	126	2	88230	35030	553	5	168030	322

[5] M. Hrkić, J. Lillis, TR# UIC-CS-02-1, [http://www.cs.uic.edu/~jlillis/sptree\\_tech.pdf](http://www.cs.uic.edu/~jlillis/sptree_tech.pdf)

[6] 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.

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

[8] 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.

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

[10] 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.

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

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

[13] 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.