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

Milos Hrkic and John Lillis, *Member, IEEE*

***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, congestion, 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.**

***Index Terms*—Buffer insertion, routing, Steiner trees, timing optimization.**

I. Introduction

In THE DEEP submicrometer 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 have 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 [13] 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 [8], [15]. In the area of routing topology construction, there are several works of particular relevance to this paper. These include the *P-Tree*-based methods [11], methods for timing driven routing of two-pin nets (including blockages) [7], [15], and methods which combine buffer insertion and topology construction [4], [10], [12], [14].

In this paper, we address the interconnect synthesis problem and introduce S-Tree and SP-Tree algorithms with a number of criteria and objectives in mind which we believe to be of practical importance. An overview of the issues emphasized follows.

A. 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 [4], [10], those methods do not meet some of our other objectives (e.g., scalability and critical sink isolation).

B. 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.

C. 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 [11], 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. See Section II-A and Section II-B for more discussion of this issue. A similar phenomenon occurs when sinks have specified signal polarity requirements (some sinks expecting an inverted signal). 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. We argue that notions of temporal and physical locality should be separated if robust and predictable behavior is to be expected. The algorithms presented herein capture physical locality through topology constraints and temporal locality through sink partitioning. Together these two items create a generalized topology space capturing both requirements. Additionally, simultaneous exploration of embedding and buffering was performed.

D. Ability to Handle Relatively High Fan-Out Nets

It is often the case that in a typical modern design flow we see some signal nets with relatively high fan-out (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

Manuscript received May 31, 2002; revised September 6, 2002. This work was supported in part by the National Science Foundation CAREER Award CCR-9875945 and in part by the Scientific Research Council under Contract 2001-TJ-914. This paper was recommended by Guest Editor S. S. Sapatnekar.

The authors are with the Department of Computer Science, University of Illinois, Chicago, IL 60607 USA (e-mail: mhrkic@cs.uic.edu; jlillis@cs.uic.edu).

Digital Object Identifier 10.1109/TCAD.2003.809648

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., [4], and [10] which are high-degree polynomial and exponential respectively). In our approach, we use a set of tools to address this problem. The SP-Tree algorithm is able to generate high quality solutions for small and medium size nets. The S-Tree algorithm is able to efficiently solve high fan-out nets.

E. Cost/Performance Tradeoffs and Congestion

We recognize that while our algorithms focus on the one-net problem, a real computer-aided design (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, for example, to 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 700 K buffers will be required simply to buffer global or near-global interconnects [3], it becomes clear that the overuse of buffering resources will have dramatic consequences.

Having these issues in mind, we have designed our algorithms in a way that naturally exploits three degrees of freedom: topology, embedding (placement of Steiner nodes in a target graph), and buffer placement. Different utilization of these degrees of freedom provides a unique framework with a tradeoff between solution space coverage and runtime. An overview of how these degrees of freedom are exploited is given in the Section II, then we sketch some of the implementation details in the Section III.

II. Solution Space Description

The motivation behind topology spaces of the S-Tree and the SP-Tree is that they are compatible with bottom-up dynamic programming framework. We will first examine topology space of the S-Tree. To understand the SP-Tree topology space, it is also instructive to examine the P-Tree space.

A. S-Tree Space

We begin with the problem of mapping the Steiner nodes of a given topology to the vertices of a target graph.

The S-Tree algorithm uses a given topology to capture physical locality of sinks (this topology can be generated by any means). 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).

S-Tree generalizes this notion to provide more flexibility in the topology space (in fact, exponential flexibility). This is done through a sink partition (in general, this sink partition may be arbitrary, but a heuristic partition, likely near critical sinks from noncritical sinks, is useful in practice).

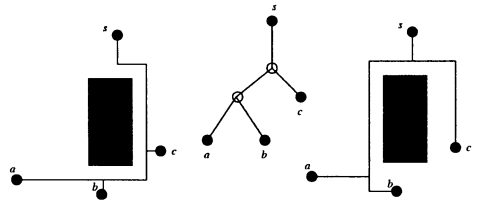

Fig. 1. Topology for a three-sink net and two physical embeddings of that topology. Pin s is the driver.

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 with left and right subtrees $l(u)$ and $r(u)$. Some sinks in the subtree belong to S_1, 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 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 nonempty 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. If u is a sink, then it does not have subtrees. 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 $P_2(u) = \emptyset$. The 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 six-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 relabeled (b and e now being closer to the root and c and f farther away).

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 extra work versus the totally fixed topology case. Second, it is well suited to timing-related issues where it is often desirable for groups of critical or noncritical sinks to be in the same subtree. If S_1 and S_2 are critical and noncritical sinks (or are so believed), then the stitching operation enables this quite naturally.

To illustrate the second point, consider Fig. 3. Topology and sink partition are given; sink b is critical. It is likely that we desire a direct path from s to b which decouples all off-path capacitance with one or more buffers. For the given topology, the

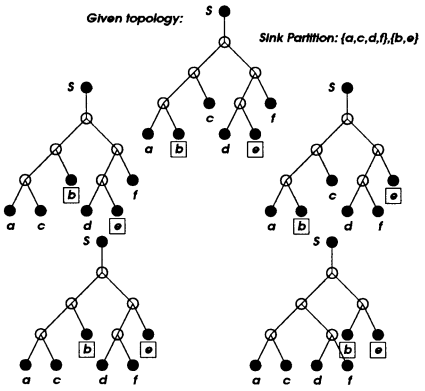

Fig. 2. S-Tree topology solution space example.

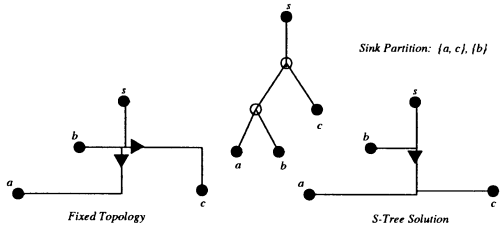

Fig. 3. Illustration of critical sink isolation and buffer savings in the S-Tree.

best we can do is illustrated on the lower-left. In the S-Tree, 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 one buffer. We expect that such savings will be increasingly important as more buffers are needed in typical designs and conservation of buffering resources becomes crucial.

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, and its embedding and buffer assignments which minimize cost (e.g., some function of buffers, area, wire length) subject to timing constraints.

B. P-Tree Space

The P-Tree algorithm [11] 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 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/noncritical 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 decouple the noncritical 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.

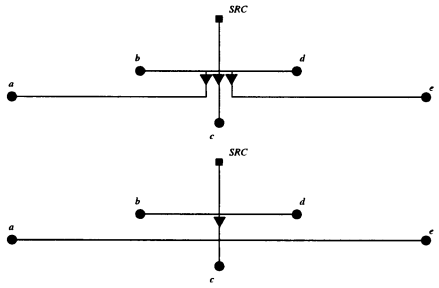

Fig. 4. P-Tree's inability to capture solution of smaller cost. Critical sinks are b and d. Sink permutation is $\{a, b, c, d, e\}$, sink partition is $\{a, c, e\}$, $\{b, d\}$; proposed approach captures solution on bottom.

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 in that we enable the partitioning of sinks into four sets (if appropriate) to consider sink polarity requirements: 1) critical/positive; 2) critical/negative; 3) noncritical/positive; and 4) noncritical/negative. This generalization has been applied to the S-Tree framework as well.

Before discussing algorithmic details, we clarify the objectives of SP-Tree versus those of the S-Tree. The main advantage S-Tree has is scalability, while the advantage of SP-Tree is solution space coverage. The goal 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 ten or 12 sinks). On the other hand, S-Tree (or some hybrid approach) may be used for larger fan-out nets (e.g., up to 20 sinks).

III. Algorithmic Overview

This section sketches details of the proposed algorithms. 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). We then show how to join branching solutions followed by methods to eliminate suboptimal solutions as early as possible. Finally, we discuss how the stitching idea can be incorporated into this framework.

A. Target Graph Construction

As a preprocessing step we construct a graph on which an abstract topology will be embedded. We extend grid lines from the driver and every sink. Intersections of those grid lines become candidates for branch points (Fig. 5). To allow simultaneous routing blockage avoidance, we remove all vertices (and edges adjacent to them) which are covered by routing blockages. Then, we include additional gridlines which are adjacent to boundaries of those blockages (shifted in appropriate direction by value

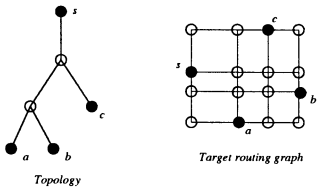

Fig. 5. Illustration of abstract topology and target routing graph.

determined from technology parameters). Intersection of those new gridlines with existing ones also gives candidates for branch points. Routing blockages that are not of rectangular shape can be represented as a union of rectangular shaped blockages.

Another type of blockage is buffer blockage—i.e., the region in which the buffer insertion is not allowed, but through which wires can pass. To handle those, we mark all vertices in routing graph that are covered by buffer blockages to prevent buffer insertion (but still consider them as branching candidates). Also, we add extra gridlines around buffer blockages to allow detours for possible buffer pickups.

It is worth mentioning that our algorithms are designed to work on general graphs (which can represent multiple layers, diagonal wires, etc.) and that this is just one of the ways to construct a target routing graph (we used this construction in our experiments).

To capture congestion, we can assign some cost (different from default values) to each edge and vertex in the graph. Vertex cost should reflect placement congestion information. If the region is congested, then it is "expensive" to insert a buffer at that location and it would be "cheaper" to insert a buffer in some less congested area. In this way, cost of inserting a buffer is context dependent. If a buffer is not inserted at some vertex, there is no contribution of that vertex to the total solution cost. In the similar way, edge cost should reflect routing congestion information, i.e., routing over regions with smaller routing congestion is "cheaper" then routing over highly congested areas. Considering both routing and placement congestion, noncritical nets (and noncritical subtrees of a net) are stimulated to use less expensive resources, leaving scarce resources in congested regions available for critical nets.

B. Solutions for the Nonstitching Cases

For ease of presentation, we first describe the dynamic programming decomposition used in the S-Tree and the SP-Tree without stitching (equation with one set) which is similar in flavor to several past works [9], [11]. 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 (e.g., subsolution's composite wire and buffer cost), has upward capacitance c, and has required arrival time q at its root. Given this notion of a signature, a subsolution is *nondominated* if no other solution is superior in *all three* dimensions. Any dominated solution may be discarded. Note that while, as many papers have noticed [4], 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. Since c represents the load of a solution, once we insert a buffer, c becomes equal to the input capacitance of that buffer loosing any downstream information. In a case without buffer insertion, cost of a wire segment is not necessarily proportional to the capacitance of that segment due to different routing congestion induced costs. To capture cost of a solution we need a separate parameter p. On the other hand, we need to keep the actual downstream load c for delay computation purposes. This increases run-time complexity, but we believe it is truly necessary for practical solutions; experimental evidence supports this belief.

Subroutine: JoinSTree(T, G)
T: Topology subtree; G: Target routing graph
c1 **for** each vertex $v \in G$
c2 $A^b[v] \leftarrow$ **Join**($A.left[v], A.right[v]$)
c3 **endfor**
c4 $A \leftarrow$ **GenDijkstra**(A^b, G)
c5 **return** A

Subroutine: ComputeSTree(T,G)
T: Topology subtree; G: Target routing graph
b1 **if**($leaf(T)$)
b2 $A(T) \leftarrow$ **ComputeInitial**(G, T)
b3 **else**
b4 $A(T.left) \leftarrow$ **ComputeSTree**($T.left, G$)
b5 $A(T.right) \leftarrow$ **ComputeSTree**($T.right, G$)
b6 $A(T) \leftarrow$ **JoinSTree**(T, G)
b7 **endif**
b8 **return** $A(T)$

Algorithm: S-Tree(T, G, s)
T: Topology; G: Target routing graph
s: source node
a1 $A(T) \leftarrow$ **ComputeSTree**(T, G)
a2 $Final \leftarrow$ **AugmentForDriver**($A(T), s$)
a3 **return** $Final$

Fig. 6. Basic S-Tree algorithm.

Let u be a vertex in the given topology T and v be a vertex in the target routing graph G (Fig. 5). In the S-Tree, we define $A(u, v)$ to be the set of nondominated solutions for subtree rooted at u in given topology T, with root placed at v in G, connecting all sinks in that subtree. In the SP-Tree, we define $A(i, j, v)$ to be the set of nondominated 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 basic S-Tree algorithm appears in Fig. 6 and for the more complex SP-Tree case in Fig. 7).

When all sets are established, the candidate solutions in $A(u_s, v_s)$ (where u_s is the root of the topology and v_s is the location of the driver in the target graph) for the S-Tree case and $A(1, n, v_s)$ in the SP-Tree case are augmented to consider the effect of the driver (additional load-dependent delay) which then results in the overall set of nondominated solutions with respect to cost p and required-time (slack) 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 the S-Tree variant and $A^b(i, j, v)$ in the SP-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 SP-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 Fig. 6 for S-Tree and Fig. 7 for SP-Tree).

```
Subroutine: JoinSPTree(i, j, G)
      i,j: Sub-solution covering sinks from i to j
      G: Target routing graph
f1    A^b ← ∅
f2    for each vertex v ∈ G
f3      for k = i to j-1
f4        A^b[i][j][v] ← A^b[i][j][v] ∪ Join(A[i][k][v], A[k+1][j][v])
f5      endfor
f6    endfor
f7    A[i][j] ← GenDijkstra(A^b[i][j], G)
f8    return A
Algorithm: SP-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] ← Initial(P, G, i)
e2    for gap = 1 to n-1
e3      for i = 1 to n-gap
e4        A[i][i+gap] ← JoinSPTree(i, i+gap, G)
e5      endfor
e6    endfor
e7    Final ← AugmentForDriver(A[1][n], s)
e8    return Final
```

Fig. 7. Basic SP-Tree algorithm.

```
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      v1 ← Top Queue
d7      if Solution(v1) is not suboptimal
d8        A(v1) ← A(v1) ∪ Solution(v1)
d9        for each edge (v1, v2) adjacent to v1
d10         v2 ← Augment(v1, v2)
d11         Insert v2 into Queue
d12       endfor
d13     endif
d14   endwhile
d15   return A
```

Fig. 8. Generalized Dijskstra's algorithm.

As an example, in the S-Tree case, proceeding bottom-up in the topology, suppose we are at some vertex u; we visit each graph vertex v and first compute $A^b(u,v)$ by joining solutions from the appropriate subtrees (for instance, a solution in $A^b(u,v)$ are formed by joining a solution in $A^b\big(l(u),v\big)$ with one from $A^b\big(r(u),v\big)$ where $l(u)$ and $r(u)$ are u's left and right children in T).

It is worth mentioning that *join* step (line f4 in Fig. 7) is trivial in a two-dimensional case when we keep only c and q and can be done in time proportionally linear to the sizes of the joining sets. In case where we have (p, c, q) triples, to guarantee optimality we have to construct a cross product of all costs p and join sets in respect of that cross product (see Section III-D). Also, pruning of dominated solutions is no longer trivial. Methods and modified data-structures from [9] have been used (see Section III-E).

C. Generalized Dijkstra's Algorithm

We have defined what the solution sets $A(\cdot)$ and $A^b(\cdot)$ mean and have given an idea of how to compute $A^b(\cdot)$. What remains is the computation of $A(\cdot)$. We propose that the problem can be solved efficiently with a generalized version of Dijkstra's shortest path algorithm (Fig. 8). 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 [7] and [15].

We start by initializing the priority queue with $A^b(\cdot)$ solutions. These solutions represent the nondominated solutions that are constrained to be a branching solution at particular vertex v in G. These solutions are candidates for the $A(\cdot)$ solutions and not all of them will be chosen at the end.

Once the queue is initialized, we start removing candidate solutions from the queue and perform a suboptimality test (i.e., compare it with already stored solutions). If a solution is not suboptimal, we store it in a permanent data structure. Then, for each neighbor in the target graph, we augment its cost, load, and slack for the wire segment it traverses and buffers that we try to insert. Those augmented solutions are tested for suboptimality again and if they pass the test they are inserted back into the queue. This additional suboptimality test is not required for the optimality of the algorithm, but we noticed a reduction in run-time that we could not ignore due to a smaller queue size. We repeat this until the queue becomes empty. Instead of a single "best" solution, we generate a set of nondominated solutions and instead of a single label we have a list of nondominated labels at each vertex v in G.

Solution candidates in the queue (which is implemented as a binary heap) are ordered by best cost first, second, by slack, and third, by downstream load. The main reason for this ordering is that pruning of suboptimal solutions (see Section III-E) can be done more efficiently. If we know that next expanded candidate solution has higher cost than the previous one we can exploit this property and design our dominance query as two-dimension using only (c, q). Further, for expanded candidate solutions to be monotone, we have to order them secondarily by slack because downstream load value is not monotone (high load of some candidate solution is reduced to a buffer input capacitance after a buffer is inserted). Another reason for this ordering is that it allows an efficient insertion of accepted solutions into our data structure, what is done in constant time (insert always at the tail of the list).

D. Join Operation

Since our $A(\cdot)$ sets contain (p, c, q) triples, when we join two sets $A_l(\cdot)$ with $A_r(\cdot)$ we have to join every triple from one set with every triple from the other. We then prune suboptimal solutions. Joining two triples, (p_{li}, c_{li}, q_{li}) and (p_{rj}, c_{rj}, q_{rj}) into (p, c, q) is done by adding up costs and loads and taking the worst slack

$$\begin{aligned} p &= p_{li} + p_{rj} \\ c &= c_{li} + c_{rj} \\ q &= MIN\,(q_{li}, q_{rj})\,. \end{aligned}$$

To efficiently join two sets, we use methods from [10]. In each set $A(\cdot)$, we can store (p, c, q) triples distributed into linked lists based on cost p, i.e., all triples of the same cost are in the same list. Then we use "*join_list*" primitive [10] to join these lists. We repeat this for every possible pair of costs p (basically a cross product). It is worth mentioning that *join_list* step can be done in time proportionally linear to the sizes of the joining sets [10].

case	condition	action
(1)	$c < t.c$ and $q \leq t.q_{l\_max}$	$t \leftarrow t.left$
(2)	$c \leq t.c$ and $q > t.q_{l\_max}$	**return** "non-dominated"
(3)	$c \geq t.c$ and $q \leq t.q_{l\_max}$	**return** "dominated"
(4)	$c > t.c$ and $q > t.q_{l\_max}$	$t \leftarrow t.right$

Fig. 9. Recursive rules for dominance testing.

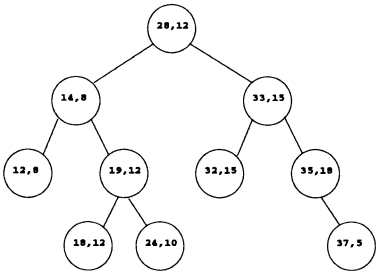

Fig. 10. Augmented binary search tree for detecting dominance property. Vertex labels are c (capacitance) and $q_{l\_\max}$ (maximum slack in left subtree).

E. Dominance Property and Pruning

An important concept through this paper is the notion of *non-dominated* solution. Since a single solution is characterized by a triplet (p, c, q) (cost, capacitance, and slack), there is no total order on solutions in one particular $S(\cdot)$ set.

Consider two solutions (p, c, q) and (p, c', q') that have the same cost. We say that (p, c, q) *dominates* (p, c', q') if ($c < c'$ and $q \geq q'$) or ($c = c'$ and $q > q'$). If the first solution has smaller load and equal or better slack (or equal load and better slack), then we do not want to keep the other solution.

To determine dominance in general case, a solution (p, c, q) is nondominated if no other solution is better in all three dimensions. Given a nondominated set of solutions A and a candidate solution (p, c, q), we can say that

$$\begin{aligned}&(p,c,q) \text{ is dominated by } A\\ &\Leftrightarrow \exists (p',c',q') \in A, p \geq p'\\ &\qquad \text{and } c \geq c' \text{and } q \leq q'.\end{aligned}$$

Because the solutions are expanded from the queue in nondecreasing order of cost p, we observe that the first inequality ($p \geq p'$) holds for every member of A. Thus, the problem reduces to determine if some member of A is at least as good as (p, c, q) in the c and q dimension. The existence of such a member of A can, of course, be determined through a linear scan of A. However, with use of appropriate data structure the problem can be solved in $O\big(\log(|A|)\big)$ time.

Such a data structure is given in [9] (for the optimization of static routing topologies) and is adopted here with some modifications. We store the members of A in a binary search tree ordered by c (p is ignored due to the reasoning above). At each node we store $t.c$ (the associated c value) and $t.q_{l\_\max}$ (the largest q value in the left subtree, including the current value of q). The recursive rules for traversing the tree are shown in Fig. 9 (in the base case where the tree is empty, the solution is clearly nondominated).

An example of this data structure appears in Fig. 10. Keep in mind that underlying shown values in the tree are p values, so values in the tree are not in general nondominated among themselves, but the corresponding (p, c, q) triples are. Consider a query in which we want to determine if (29,7) is dominated. We apply case 3 at the root since some solution in the left subtree dominates (has smaller cost and load and better slack). Suppose instead the query is (10,7); in this case, we traverse the tree all the way to the vertex (12,8) and, finally, a NULL subtree to conclude that it is nondominated. If the query is (10,15), we apply case 2 at the root and declare it nondominated.

The idea is that instead of using a linked list to store values of A, we use the augmented binary tree. The strategy is implemented such that queries and insertions are always logarithmic in the size of the tree by applying a balanced scheme such as red–black trees. Our observation is that updating the augmenting values during the rotation operations is not necessary. In fact, $q_{l\_\max}$ value does not have to be strict in the sense that as long as $q_{l\_\max}$ at some node is not smaller then the max q from the left subtree (including the current node) and not greater then the max q of all elements in the tree that have smaller c, the dominance rules will hold. Since rotation operations for balancing red–black trees do not violate these constraints if the elements are inserted in nondominated order (what is our case), we do not need to perform updates.

F. Incorporating "Stitching"

In addition to the ability to deal with blockages, one of the contributions of this paper is the notion of using sink partitions to capture *temporal* and/or *polarity* locality and expand the solution space accordingly. We refer to the way in which the various notions locality interact as "stitching."

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 nondominated 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 connect sinks in $S_1 \cup S_2$ contained in $i..j$ (in this case *all* sinks in $i..j$). Note that the set of topologies covered by this set is a strict generalization of the usual P-Tree space in that topologies are built up additionally using the $A_1(\cdot)$ and $A_2(\cdot)$ solutions.

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 nondominated solutions in the resulting union are retained. In this way, we separate critical/noncritical subtrees (and similarly when considering sink polarities). Given the pseudocode for modified **JoinSTree** subroutine that incorporates stitching (Fig. 11), we can extend **JoinSPTree** to capture the stitching idea. Modified **JoinSPTree** subroutine is given in Fig. 12.

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 expo-

```
Subroutine: JoinSTree(T, G)
      T: Topology subtree; G: Target routing graph
c1    for each vertex v ∈ G
c2      A_1^b[v] ← Join(A_1.left[v], A_1.right[v])
c3      A_2^b[v] ← Join(A_2.left[v], A_2.right[v])
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[v] ← Join(A_12.left[v], A_12.right[v])
c9      A_12^b[v] ← A_12^b[v] ∪ Join(A_1[v], A_2[v])
c10   endfor
c11   A_12 ← GenDijkstra(A_12^b, G)
c12   return A
```

Fig. 11. S-Tree Join with two sink partitions.

```
Subroutine: JoinSPTree(i, j, G)
      i,j: sinks i to j to be connected by sub-solution
      G: Target routing graph
f1    A^b ← ∅
f2    for each vertex v ∈ G
f3      for k = i to j-1
f4        A_1^b[i][j][v] ← A_1^b[i][j][v] ∪ Join(A_1[i][k][v], A_1[k+1][j][v])
f5        A_2^b[i][j][v] ← A_2^b[i][j][v] ∪ Join(A_2[i][k][v], A_2[k+1][j][v])
f6      endfor
f7    endfor
f8    A_1[i][j] ← GenDijkstra(A_1^b[i][j], G)
f9    A_2[i][j] ← GenDijkstra(A_2^b[i][j], G)
f10   for each vertex v ∈ G
f11     for k = i to j-1
f12       tmp ← Join(A_12[i][k][v], A_12[k+1][j][v])
f13       A_12^b[i][j][v] ← A_12^b[i][j][v] ∪ tmp
f14     endfor
f15     A_12^b[i][j][v] ← A_12^b[i][j][v] ∪ Join(A_1[i][j][v], A_2[i][j][v])
f16   endfor
f17   A_12[i][j] ← GenDijkstra(A_12^b[i][j], G)
f18   return A
```

Fig. 12. SP-Tree Join with two sink partitions.

nential 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 noncritical "not sure," if critical and highly critical. We then define the following sets: A_1, A_2, A_3, A_{12}, A_{23}, and A_{123}. Observe that we do not join solutions that spawn highly critical and noncritical sinks (A_{13}) for obvious reasons.

The other case of practical interest is with four partitions: (S_1) positive polarity constraints and critical, (S_2) positive and noncritical, (S_3) negative and noncritical, and (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}, and A_{1234} in a similar way. Observe that, in "intermediate" sets, we join all critical sets, all noncritical sets, all positive polarity sets and all negative polarity sets. In this way, we are able to simultaneously capture *physical*, *temporal*, and *polarity* locality.

IV. Discussion

A. Implementation Issues

Some important implementation details have not been mentioned in the interest of space. In balanced binary tree data structure for efficient determination of the dominance property, we do not need to keep elements with identical c values. When a new element is inserted and its c value is already in the tree, that means that the element is nondominated and it must have its q value larger than the existing value. So, we only need to update $t.q_{l\_\max}$ to $MAX(q, t.q_{l\_\max})$.

In subroutine **Join**, since we have to compute a cross product of joining sets, we first compute the new joined costs and sort them. Then, we compute the actual joins in the min-cost first order. This allows us not only to have solutions expanded in min-cost order what is required for efficient pruning, but also to perform list pruning for some fixed cost (which is done in $O(1)$ time; just check the tail of the list) and then only for those elements that "survived" we check the general dominance through balanced binary tree data structure.

Sometimes a subtree's sinks are from only one set. Careful bookkeeping exploits this and avoids redundant computations.

B. Initial Topology

Although initial topology is a parameter to S-Tree algorithm, it has to be generated somehow. We used the minimum spanning tree (MST)-based approach and a P-TreeA topology, both discussed in [11].

C. Speedup Techniques

It is often the case that nondominated candidate solution can be discarded by computing lower-bounds or putting a threshold on the global cost. We have incorporated such strategies and achieved substantial speedups. Also, a buffer library pruning [2] (discarding buffers which are dominated by other buffers) as a preprocessing step improves run-time.

D. Solution Quality

Careful choice of buffer candidate locations may improve solution quality. For example, segmenting long wires and/or inserting additional grid lines into graph may yield better solutions. These simple modifications do have impact on run-time but do not affect overall algorithm complexity.

E. Sink Partitioning

Currently, we use simple heuristics to partition critical from noncritical 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.

F. Modifications

There are some trivial modifications to the algorithm which are important in practice. Inverter handling is done through previously studied techniques in buffer insertion. Buffer insertion step is done in generalized Dijkstra's algorithm in a way similar to [8], implicitly allowing buffer cascading.

G. Solution Space

It is instructive to compare solution space coverage of our algorithms and other known techniques (Fig. 13). If we start with the van Ginneken-style buffer insertion algorithm [13] on fixed and embedded topology, we can generalize it by relaxing the embedding constraint and allowing simultaneous topology embedding and buffer insertion on fixed (nonembedded) topology. By relaxing topology constraint further we can take

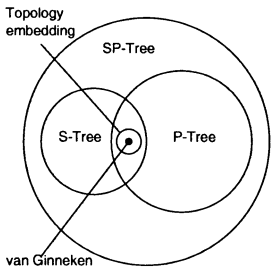

Fig. 13. Illustration of solution space coverage.

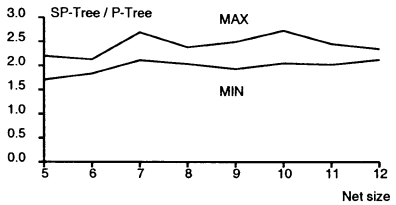

Fig. 14. Run-time of SP-Tree scaled to run-time of P-Tree. Curves show minimal and maximal ratio.

two directions. Allowing topology to break using stitching, we get the S-Tree algorithm, or by considering all topologies that satisfy sink permutation we get the P-Tree algorithm. If we start from the same topology, solution spaces of S-Tree and P-Tree do intersect, but one is not contained in the other (also the fixed topology mentioned above must be consistent with S-Tree and P-Tree spaces). In the SP-Tree algorithm, stitching allows topologies that break sink permutation constraint, so both S-Tree and P-Tree spaces are completely contained in SP-Tree's space together with some other topologies that are not considered by either S-Tree or P-Tree.

H. Algorithm Complexity

Let us first consider the case without *stitching*, which is an extended version of P-Tree algorithm with the ability to handle blockages. In the literature, P-Tree algorithm is mentioned in "various running modes" (P-TreeA, P-TreeAT, and P-Tree). We have extended the P-Tree algorithm from [10], since P-TreeA and P-TreeAT do not perform buffer insertion (P-TreeA constructs a timing-driven min-cost unbuffered tree and P-TreeAT constructs a set of nondominated unbuffered trees with cost/slack tradeoff). As stated in [10], evaluating the algorithm in terms of the number of primitives it executes, we have an $O(n^5)$ algorithm. This can be seen in the computation of A^b. There are $O(n^4)$ sets $A^b(v,i,j)$ (note that there are $O(n^2)$ vertices in target routing graph) and for each such set we execute $O(n)$ primitives. The primitives are not constant time operations. However, it can be argued that the size of these sets is polynomially bounded in the parameters of the problem instance and, thus, the algorithm is pseudopolynomial overall. In further discussion, we will use notation $P(n)$ for nonconstant time operations that are polynomially bounded in the parameters of the problem instance.

Our first extension of P-Tree [10] includes the ability to handle blockages. We have done that through general graph model and modified Dijkstra's algorithm (Section III-C) without affecting the overall algorithm complexity. The overall complexity is still dominated by computing $A^b(\cdot)$ sets. In P-Tree [10], computation of sets $A(\cdot)$ was performed by four sweeps of the grid graph ($O(n^2)$ size) and that was invoked $O(n^2)$ times yielding overall complexity of $O(n^4)$. With our modifications we have $O(n^2)$ runs of modified Dijkstra's algorithm yielding $O(n^2) \times O\Big(n^2 \log\big(P(n)\big)\Big)$, what is $O\Big(n^4 \log\big(P(n)\big)\Big)$, so computation of A^b still dominates overall complexity.

Our second extension incorporates *stitching*. Let us examine the case with two sink partitions (cases with more sink partitions are similar). Instead of computing sets $A(\cdot)$ and $A^b(\cdot)$ we now have to compute sets $A_{12}(\cdot)$, $A_1(\cdot)$, $A_2(\cdot)$, $A^b_{12}(\cdot)$, $A^b_1(\cdot)$, and $A^b_2(\cdot)$. In other words, we have about three times more work. In practice, due to pruning suboptimal solutions and due to the fact that sinks in some subtree may be from one partition only, we have a slowdown smaller than a factor of three. In Fig. 14, we have compared a slowdown factor of SP-Tree to P-Tree on various nets. We have plotted maximal and minimal slowdown factors per net size that we encountered in our experiments. By *stitching*, we are able to expand solution space exponentially (in general case) and pay only constant overhead in run-time.

As for the S-Tree, we have $O(n^3)$ sets $A^b(u,v)$. In computing A^b sets we execute $O(n^3)$ primitives. For computation of A we have $O(n)$ calls of modified Dijkstra's algorithm what yields a complexity of $O\Big(n^3 \log\big(P(n)\big)\Big)$.

V. Experiments

We have implemented S-Tree and SP-Tree algorithms and performed some initial experiments to evaluate their effectiveness. The experiments were conducted in Solaris environment on a Sun Ultra 1 workstation with a 200-MHz CPU and 384 MB of RAM. [1] 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 [10] (modified to handle blockages) which is known to produce high quality solutions particularly for uniform required times and recursively merging and pruning (RMP) [4]. For RMP, we also reported results for "Quick" running mode where heuristics are used to reduce CPU time. Wire length is reported in micrometers, slack in picoseconds and execution time in seconds of CPU time. Technology parameters are representative of 0.25-μm technology. For each net three solutions are shown: minimum cost, minimum cost feasible (cheapest with positive slack), and maximum slack. Runs that failed to report result in 30 min were terminated. The maximum amount of memory used by S/P-Tree algorithms was 83 MB. Solution cost in the S/P-Tree algorithms is defined as a function of wire capacitance (which is proportional to wire length) and input buffer capacitance ($WireCap + \beta \cdot InBufCap$). This enables arbitrary kind of cost normalization (e.g., bias toward minimizing buffers or wire length). Buffer input capacitance is not a good measure of buffer cost since buffer libraries contain cascaded buffers which do have small input capacitance but large area. In

[1] For reference, an 800-MHz Celeron is roughly 3.5 times faster than the machine that was used for compatibility with the RMP executable

TABLE I
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	
net1-06	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
net1-08	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	677
	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
net1-10	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
net1-12	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 II
6-12 PIN NETS, NONUNIFORM 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	
net2-06	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
net2-08	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
net2-10	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
net2-12	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

absence of "real world" buffer cost data we used this cost, but, in general, it is given to our algorithms as a parameter.

In the first set of experiments, we used randomly generated nets with small variations in sink required arrival time (Table I). Also, we made buffers inexpensive ($\beta = 2$). This experiment shows that when there are no variations in *temporal* or *polarity* locality between sinks; an 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. Also, it is clear that in complex designs, algorithm that maximizes slack without considering solution cost is not of much use for interconnect construction (it could be used for delay estimation).

In the second set of experiments, we allowed large variations in sink-required arrival times (Table II). We made buffers very expensive ($\beta = 200$). [2] This allows more wire detours in searching for a 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 the third set of experiments, we included sink input polarity constraints (Table III). 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 the stitching idea are even larger for those "difficult" nets. On the cost/performance tradeoff curve (Fig. 15), 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.

In the fourth set of experiments (Table IV), we demonstrate the ability of S-Tree to handle relatively high fan-out nets. In cost function, buffer cost dominates wire cost. Other algorithms were not able to produce output within the given time limit.

[2]Considering the overhead of adding a buffer (e.g., introduction of vias, local routing overhead, power consumption, and possibility of needing to do an ECO on the placement), this high buffer cost seems justified in some scenarios

TABLE III
6-10 PIN NETS, NONUNIFORM 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	
net3-06	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
net3-08	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
net3-10	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

TABLE IV
15-21 PIN NETS (HIGH FANOUT NETS), BUFFER-DOMINATED COST

Net	Alg.	min cost				min cost feasible				max slack				cpu
		wl	slk	buf	cost	wl	slk	buf	cost	wl	slk	buf	cost	
net4-15	S-Tree	49424	-3451	0	49	39501	209	5	5049	46802	385	8	8049	18.9
net4-18	S-Tree	35325	-3784	0	35	43696	62	5	5043	49776	450	11	11049	42.8
net4-21	S-Tree	45708	-2836	0	45	45480	409	3	3045	47421	1137	7	7047	115.5

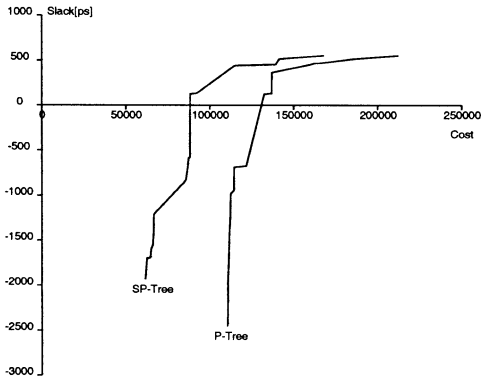

Fig. 15. Cost/performance tradeoff curve for P-Tree and SP-Tree on net3–10.

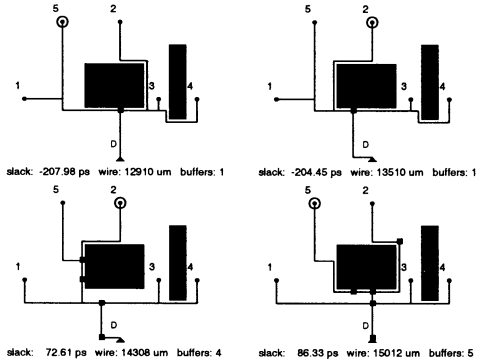

Fig. 16. Example topology.

VI. EXAMPLE

To demonstrate how *stitching* works in practice, we constructed a simple example instance and ran S-Tree algorithm on it. Some of the solution topologies are shown in Fig. 16. 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, the total wire length in micrometers, and the 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 the tradeoff curve toward more expensive (and faster) ones, it can be clearly seen how that initial topology "breaks" and how, 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.

VII. CONCLUSION

We have presented the S-Tree and SP-Tree algorithms for synthesis of buffered interconnects. The approach incorporates a unique combination of real-world issues (handling of routing and buffer blockages, cost minimization, congestion awareness, 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 versus previous approaches has been experimentally verified.

ACKNOWLEDGMENT

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

REFERENCES

[1] C. Alpert *et al.*, "Buffered steiner trees for difficult instances," in *Proc. Int. Symp. Physical Design*, 2001, pp. 4–9.
[2] ——, "Buffer library selection," in *Proc. Int. Conf. Comput. Design* , 2000, pp. 221–226.
[3] J. Cong. (1997) Challenges and opportunities for design innovations in nanometer technologies. Frontiers in Semiconductor Research: A Collection of SRC Working Papers. [Online]. Available: www.src.org/prg_mgmt/frontier.dgw
[4] J. Cong and X. Yuan, "Routing tree construction under fixed buffer locations," in *Proc. Design Automation Conf.* , 2000, pp. 368–373.
[5] M. Hrkić and J. Lillis, "S-Tree: A technique for buffered routing tree synthesis," in *Proc. Design Automation Conf.*, 2002, pp. 578–583.
[6] ——, "Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost and blockages," in *Proc. Int. Symp. Phyiscal Design*, 2002, pp. 98–103.

[7] S.-W. Hur, A. Jagannathan, and J. Lillis, "Timing-driven maze routing," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 234–241, Feb. 2000.
[8] A. Jagannathan, S.-W. Hur, and J. Lillis, "A fast algorithm for context-aware buffer insertion," in *Proc. Design Automation Conf.*, 2000, pp. 368–373.
[9] J. Lillis, C.-K. Cheng, and T.-T. Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE J. Solid-State Circuits*, vol. 31, pp. 437–447, Mar. 1996.
[10] ——, "Simultaneous routing and buffer insertion for high performance interconnect," in *Proc. 6th IEEE Great Lakes Symp. VLSI*, Ames, IA, Mar. 1996, pp. 148–153.
[11] ——, "New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing," in *Proc. Design Automation Conf.*, 1996, pp. 395–400.
[12] T. Okamoto and J. Cong, "Buffered steiner tree construction with wire sizing for interconnect layout optimization," in *Proc. Int. Conf. Computer-Aided Design*, 1996, pp. 44–49.
[13] L. P. P. P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal elmore delay," in *Proc. Int. Symp. Circuits Syst.*, 1990, pp. 865–868.
[14] A. H. Salek, J. Lou, and M. Pedram, "MERLIN: Semi-order-independent hierarchical buffered routing tree generation using local neighborhood search," in *Proc. Design Automation Conf.*, 1999, pp. 472–478.
[15] H. Zhou, D. F. Wong, I. M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," in *Proc. Design Automation Conf.*, 1999, pp. 96–99.

Milos Hrkic received the B.S. degree in computer science from the University of Illinois, Chicago, where he is currently pursuing the Ph.D. degree in computer science.

He was an Intern with IBM Austin Research Laboratory, Austin, TX, in 2001 and 2002. His research interests include combinatorial optimizations, VLSI design automation, in particular interconnect synthesis and buffering.

Mr. Hrkic received an award at the International Conference on Computer-Aided Design CADAthlon 2002 Programming contest.

John Lillis (M'01) received the M.S. and Ph.D. degrees in computer science from the University of California, San Diego, in 1993 and 1996, respectively.

From 1996 to 1997 he was a Postdoctoral Researcher at the University of California, Berkeley, supported in part by the National Science Foundation Computer and Information Science and Engineering (CISE) Program. He joined the Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, in 1997, where he is currently an Assistant Professor in computer science. His interests include design automation for VLSI, particularly physical design and timing optimization and combinatorial optimization.

Dr. Lillis was awarded an NSF Career award in 1999 to pursue research in search mechanisms for design automation for VLSI.