# An Approach to Placement-Coupled Logic Replication

Miloš Hrkić, John Lillis, *Member, IEEE*, and Giancarlo Beraudo

***Abstract*—This paper presents a set of techniques for placement-coupled timing-driven logic replication. Two components are at the core of the approach. First, is an algorithm for optimal timing-driven fanin tree embedding; the algorithm is very general in that it can easily incorporate complex objective functions (e.g., placement costs) and can perform embedding on any graph-based target. Second, the replication tree is introduced, which allows to induce large fanin trees from a given circuit, which can then be optimized by the embedder. The authors have built an optimization engine around these two ideas and report promising results for the field-programmable gate array (FPGA) domain including clock period reductions of up to 36% compared with a timing-driven placement from versatile place and route (VPR) (Marquardt *et al.*, 2000) and almost double the average improvement of local replication (Beraudo and Lillis, 2003). These results are achieved with modest area and runtime overhead. In addition, issues that arise due to reconvergence in the circuit specification are addressed. The authors build on the replication tree idea and enhance the timing-driven fanin tree embedding algorithm to optimize subcritical paths, yielding even better delay improvements.**

***Index Terms*—Logic replication, placement, programmable logic, timing optimization.**

## I. Introduction

THE IDEA of logic replication is to duplicate certain cells in a design so as to enable more effective optimization of one or more design objectives. Recently, a few papers including [1], [4], [5], and [8] have explored the idea of using replication to effectively deal with interconnect-dominated delay at the physical level. In these papers, it is observed that because replication effectively separates multiple signal paths, it becomes easier to "straighten" input-to-output [flip-flop (FF) to FF] paths at the physical design level. These paths might otherwise have been very circuitous (and therefore high delay).

A simple example from [1] reproduced in Figs. 1 and 2 illustrates the idea. Suppose that the terminals at a, b, d, and e are fixed. There are four distinct input-to-output paths; any movement of the central cell c from the shown location will degrade the delay of at least one of these paths (assume for the moment a linear delay model). Thus, in Fig. 1, we have no choice but to tolerate nonmonotone input-to-output paths. Now suppose that we replicate cell c as shown in Fig. 2 to form c′, computing the same function, but feeding only output b while c drives only d. If we produce such a logically equivalent netlist, all input-to-output paths become virtually monotone. Note that in this example the total wire length after replication remains almost the same as before.

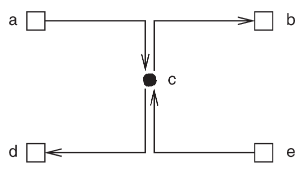

Fig. 1. Example with forced nonmonotone paths.

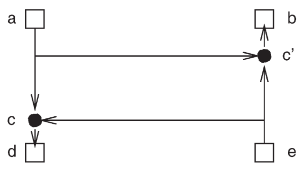

Fig. 2. Example illustrating path straightening by replication of cell *c*.

### A. Related Work

Logic replication has been applied in several different contexts in the past. Perhaps the most common application was in min-cut partitioning. Here, it is observed that by selectively replicating certain logic cells, the crossing count of a bipartition can often be reduced. Hwang and Gamal [11] proposed logic replication heuristics that reduces pin count and wiring density in partitioned logic networks. Kring and Newton [13] adapted the Fiduccia–Mattheyses partitioning algorithm to support replication. Liu *et al.* [15] devised the replication graph for finding a min-cut separating a source from a sink when replication is allowed; the replication graph was also used as the basis of a balanced bipartitioner implicitly allowing replication. Mak and Wong [17] presented a network-flow-based algorithm for the same unbalanced min-cut problem, but ensured minimum replication to achieve the min-cut. In addition, the algorithm in [17] can be generalized to separate two given subsets of nodes and determine an optimum partitioning of the minimum possible cut size using the least amount of replication. Neumann *et al.* [20] integrated cell replication into timing-driven partition-based placement. Schabas and Brown [21] presented a placement algorithm for field-programmable gate arrays (FPGAs), which uses logic replication to achieve performance gains. Their method consists of a packing algorithm that

Manuscript received May 4, 2005; revised August 15, 2005 and November 8, 2005. This work was supported in part by the IBM Ph.D. Fellowship. This paper was recommended by Associate Editor S. Sapatnekar.

M. Hrkić is with the IBM Corporation, East Fishkill, NY 12533 USA (e-mail: milosh@us.ibm.com).

J. Lillis is with the Department of Computer Science, University of Illinois, Chicago, IL 60607 USA (e-mail: jlillis@cs.uic.edu).

G. Beraudo is with Bain & Company Inc., 20122 Milan, Italy (e-mail: giancarlo.beraudo@bain.it).

Digital Object Identifier 10.1109/TCAD.2006.871624

leaves empty slots in timing critical clusters and a placement algorithm that permits logic replication to reduce the critical path length.

Replication is also being exploited to enhance the performance of high-fanout logic cells. It has been observed that by replicating a high-fanout cell, one copy can drive noncritical sinks (and thus can be smaller in area) and the other copy can drive critical sinks. This approach can improve overall area and/or performance. Lillis *et al.* [14] proposed a solution to the unconstrained two-way fanout partitioning problem under delay constraints as well as a solution to layout constrained multiway gate replication under timing constraints. Further, Lillis *et al.* [14] proposed that gate replication can be applied after gate sizing to further optimize performance and area or that these two operations may be interleaved in alternating replication and sizing steps. Srivastava *et al.* [23] derived complexity results on some replication-based optimization problems and presented a heuristic for delay optimization by gate replication under a load-dependent delay model.

Recently, a few papers have explored the idea of using replication to improve interconnect-dominated delay (e.g., [1], [4], and [5]). By replicating a cell, one can possibly improve the delay of the path passing through it, as shown in the example in Figs. 1 and 2.

The work by Beraudo and Lillis [1] exploits this phenomenon and is most closely related to this paper, so we briefly review its contributions. First, Beraudo and Lillis [1] made a compelling case for the potential of replication by observing not only that typical placements contain critical paths that are highly nonmonotone but also that the number of cells that have near-critical paths flowing through them is relatively small (thus, one may conjecture that a small amount of replication may be sufficient). Then, an incremental replication procedure was proposed and evaluated experimentally with promising results. Roughly speaking, the algorithm examined the current critical path and looked for cells to replicate; for such cells, it placed the duplicate, performed fanout partitioning, and then legalized the placement. The criteria for selecting a cell were based on the goal of inducing local monotonicity. Local monotonicity was defined by a sequence of three cells on a path $v_1$, $v_2$, $v_3$. Let $d(u, v)$ be the rectilinear distance between cells $u$ and $v$. Then, we say that the path from $v_1$ to $v_3$ is nonmonotone if $d(v_1, v_3) < d(v_1, v_2) + d(v_2, v_3)$ (i.e., traveling to $v_2$ creates a detour). In such a case, $v_2$ is a good candidate for replication so as to straighten this path without disturbing other paths passing through $v_2$.

While this strategy proved effective in reducing clock period, we now observe that a technique based on local monotonicity has limitations. Fig. 3 demonstrates this limitation. In the figure, we see a critical path (`s`, `a`, `b`, `t`) (dashed lines indicate other signal paths that may be near critical). Clearly, this path is nonmonotone and yet all subpaths (of length 3) are locally monotone. In this case (which is not unusual), the approach is unable to improve the delay.

In addition to previous work on replication, tree embedding/placement algorithms are also related to the current work. A well-studied problem is the optimal linear placement of tree circuits. Several papers (e.g., [22] and [24]) have shown polynomial time algorithms under various objectives. These algorithms have also been generalized to perform simultaneous technology mapping and linear placement [16]. While these works are interesting and often present elegant solutions, the current work requires a more general tree embedding formulation—i.e., the placement target is not the line, but is an embedding graph or the two-dimensional (2-D) plane.[1]

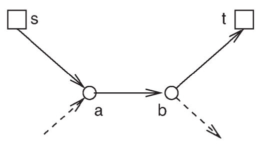

Fig. 3. Limitation of local monotonicity. Cells $a$ and $b$ are locally monotone yet the $s$-to-$t$ path is not.

## B. Contributions

With this in mind, we have developed a more robust and general replication strategy. There are two key elements around which the approach has been built.

First, we study the optimal fanin tree[2] embedding problem. In this problem, we are given the root of a fanin tree (e.g., an FF input), arrival times at the inputs (leaves) of the tree, and the physical locations of the root and inputs. Our goal then is to embed the internal nodes of the tree so as to obtain a tradeoff between the cost of the embedding (which can be quite general) and the arrival time at the root (sink) of the tree. Our solution has evolved from the closely related problem of embedding a fanout tree in buffer tree synthesis [6].

While this is an interesting result in its own right, unfortunately, most circuits, because of reconvergence, do not contain large subcircuits, which are fanin trees. This brings us to the second item at the core of the approach—the replication tree. The replication tree gives us a systematic way of taking a set of edges in a circuit forming a directed tree (e.g., with the root being the input of an FF), and, using replication, induce a genuine fanin tree that can, in turn, be optimized by the fanin tree embedder. For timing optimization as in our case, a natural selection for such a tree is a slowest paths tree (SPT) derived from static timing analysis. At this point, the embedder's ability to handle general cost functions becomes important—in particular, we are able to naturally encode the cost/benefit of replicating a cell in the "placement cost" component of the cost function.

Around these two main ideas—fanin tree embedding and the replication tree—we have built an optimization engine for the FPGA domain. Additional components of interest include a timing-driven legalizer and a set of postprocessing enhancement techniques.

Section II describes a solution to the timing-driven fanin tree embedding problem. We introduce the replication tree in Section III. Section IV gives a top-level view of our approach. Section V reveals some of the implementation details,

[1]In general, we will not be able to guarantee nonoverlapping embedded solutions. However, the effects are mitigated in a number of ways in our flow.

[2]Fanin trees are referred to by some authors as fan-out-free circuits or Leaf-DAG circuits [2]; we can handle either such structure.

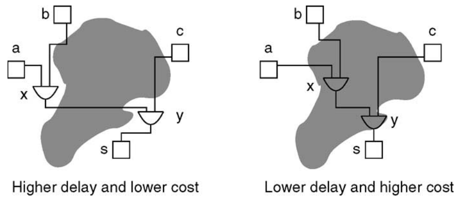

Fig. 4. Example of fanin tree embedding.

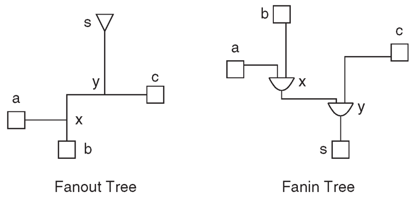

Fig. 5. Similarity between fanout and fanin tree.

including timing-driven legalization. Section VI describes the problem caused by optimal cost/max-arrival-time embedding of the replication tree that is linked to netlist reconvergence. We introduce a modified fanin tree embedding approach to address this issue. The experimental evaluation is presented in Section VII. We discuss potential remaining problems in Section VIII and conclude in Section IX.

## II. Fanin Tree Embedding

In the fanin tree embedding problem, we are given a fanin tree, placement of leaves (inputs) and root (sink), signal arrival times at the inputs, and a target placement region (in our case, this is encoded in an embedding graph). The goal is to place the internal tree nodes (gates) minimizing cost subject to an arrival time constraint at the root (typically, there is a tradeoff between cost and arrival time).

Fig. 4 illustrates two embeddings of the same fanin tree. Supposing that the shaded region in the middle has high placement cost, we can have a solution with smaller cost but larger delay (left part of the figure), or we can have a solution with better delay but larger cost (on the right).

We have observed that the problems of embedding fanin and fanout trees are very similar. An example is given in Fig. 5. On the left, we have a fanout tree with source s and sinks a, b, and c (signal flow is from top to bottom). In fanout tree embedding, we place Steiner nodes x and y, in addition to performing routing. In the fanin tree case, on the right, we have a sink s and inputs a, b, and c. Here, we place gates x and y. For this reason, we have been able to adapt the dynamic programming (DP) tree embedding algorithm from [6] and [7] to perform fanin tree embedding.

In Fig. 6, we show a general tree embedding algorithm. The input to the algorithm is a nonembedded tree topology and a target routing graph (both can be constructed by any means). In the DP approach, a candidate solution (embedding) for a subtree rooted at node $i$ in the tree with node $i$ placed at vertex $j$ in the embedding graph is represented by its signature. In general, this signature is a $k$-tuple that captures $k$ design criteria associated with the candidate solution. A partial order on such tuples determines if one candidate tuple dominates another (is superior in all dimensions). Because of this partial order, there is often not a single "best" solution for an $(i, j)$ (subtree/target graph vertex) pair, so we keep a list of all nondominated solutions. For example, if the signature is a cost/arrival time pair, we cannot tell which candidate solution is the "best" since one may be fast and expensive while the other is slow and cheap. In Fig. 6, $A^b[i][j]$ represents the set of nondominated embeddings of the subtree rooted at $i$, where $i$ is constrained to be placed exactly at vertex $j$ in the target graph. They are also called branching solutions since the actual routes branch at vertex $j$. In the same figure, $A[i][j]$ represents the set of nondominated embeddings of the subtree rooted at $i$ that is driven from the vertex $j$ in the target graph [i.e., there may be an extra wiring segment from vertex $j$ to the actual placement location of subtree root $i$ (a branching point); such a configuration is sometimes called a "single-stem" tree].

```
Subroutine: GenDijkstra($A^b, G, i$)
        $A^b$: Joined solutions; G: Target routing graph
        i: subtree root
d1      $A[i] \leftarrow \emptyset$
d2      for each solution in $A^b[i][j]$ where $j \in G$
d3          Insert $A^b[i][j]$ into Queue
d4      endfor
d5      while $Queue \neq \emptyset$
d6          $s_j \leftarrow$ Top(Queue)
d7          if $s_j$ is not suboptimal in $A[i][j]$
d8              $A[i][j] \leftarrow A[i][j] \cup s_j$
d9              for each edge $(j, x) \in G$ adjacent to $j$
d10                 $s_x \leftarrow$ Augment$(j, x)$
d11                 Insert $s_x$ into Queue
d12             endfor
d13         endif
d14     endwhile
d15     return $A[i]$
Subroutine: JoinTree($G, i, l, r$)
        G: Target routing graph; i: subtree root
        l: left subtree; r: right subtree;
c1      for each vertex $j \in G$
c2          $A^b[i][j] \leftarrow \mathbf{Join}(A[l][j], A[r][j])$
c3      endfor
c4      $A[i] \leftarrow \mathbf{GenDijkstra}(A^b[i], G, i)$
c5      return $A[i]$
Subroutine: ComputeSubTree(T,G,i)
        T: Topology subtree; G: Target routing graph
        i: subtree root
b1      if($leaf(T)$)
b2          $A[i] \leftarrow \mathbf{ComputeInitial}(G, T, i)$
b3      else
b4          $l = T.left.index$
b5          $r = T.right.index$
b6          $A[l] \leftarrow \mathbf{ComputeSubTree}(T.left, G, l)$
b7          $A[r] \leftarrow \mathbf{ComputeSubTree}(T.right, G, r)$
b8          $A[i] \leftarrow \mathbf{JoinTree}(G, i, l, r)$
b9      endif
b10     return $A[i]$
Algorithm: TreeEmbedding($T, G, s$)
        T: Topology; G: Target routing graph
        s: source node
a1      $A[s] \leftarrow \mathbf{ComputeSubTree}(T, G, s)$
a2      $Final \leftarrow \mathbf{AugmentRoot}(A[s])$
a3      return $Final$
```

Fig. 6. General tree embedding algorithm. $A^b[i][j]$ represents the set of nondominated embeddings of the subtree rooted at $i$, where $i$ is constrained to be placed exactly at vertex $j$ in the target graph. They are also called branching solutions since the actual routes branch at vertex $j$. $A[i][j]$ represents the set of nondominated embeddings of the subtree rooted at $i$ that is driven from the vertex $j$ in the target graph (i.e., there may be an extra wiring segment from vertex $j$ to the actual placement location of subtree root $i$).

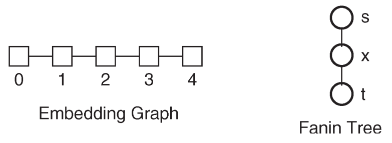

Fig. 7. Tree embedding algorithm example.

The algorithm proceeds in a bottom-up fashion starting from the leaves. First, it initializes candidate solutions at leaves (line `b2`). Then, at each internal node in bottom-up order, the algorithm joins candidate solutions for the left and right subtrees at each vertex $j$ in the embedding graph (line `c2`). Those candidate solutions are then propagated through the embedding graph using a generalization of Dijkstra's shortest paths algorithm as in [9] (in other words, a multisource wavefront expansion is performed through the embedding graph while disregarding inferior candidate solutions). Note that at line `d10` solutions are augmented for propagation cost, delay, or any other criteria of interest. In this way, we obtain a new generation of candidate solutions at the root of the current subtree. The process is repeated until we reach the root of the tree. Once at the root, we pick a final solution from the list of all candidate solutions (these nondominated solutions exhibit tradeoffs between the design criteria encoded in the signatures). From the chosen solution, the actual embedding is reconstructed in a top-down process by retracing the choices of subtree configurations that led to the chosen solution. This allows us to extract both routes and gate placements. Since we are interested in optimizing cost and delay of the tree, the DP for fanin tree embedding starts from inputs and propagates arrival time and cost toward the sink. Additional details on tree embedding algorithms can be found in [6] and [7].

Let us illustrate the embedding algorithm through a small example. In Fig. 7, we have an embedding graph on the left and a fanin tree on the right (signal flow is from `s` to `t`). Assume that `s` is placed at slot 0 and `t` at slot 4. Let the placement cost be equal to the slot index in the embedding graph and the wire cost be proportional to the wire length. Also, let the wire delay be quadratically proportional to the length and let the internal gate delay be 1 unit. Finally, assume that the placement cost of the source and target be zero, since those cells are already placed and not movable. We want to place the internal node `x` while considering both cost and delay. Assume that solution signatures are represented as cost/arrival time pairs. We start propagation through the topology from `s` to `t`. The initial solution for the source is given by $A^b[s][0] = \{(0,0)\}$. After wavefront expansion, we obtain $A[s][1] = \{(1,1)\}$, $A[s][2] = \{(2,4)\}$, $A[s][3] = \{(3,9)\}$, and $A[s][4] = \{(4,16)\}$. Now, for node `x`, we first have "branching" solutions $A^b[x][1] = \{(2,2)\}$, $A^b[x][2] = \{(4,5)\}$, $A^b[x][3] = \{(6,10)\}$. For example, $A^b[x][2]$ is obtained by augmenting solution $A[s][2]$ with placement cost and gate delay of node `x` at slot 2: $(2,4) + (2,1) = (4,5)$. Next, the wavefront expansion is run again and we obtain $A[x][1] = \{(2,2)\}$, $A[x][2] = \{(3,3)\}$, $A[x][3] = \{(4,6)\}$, and $A[x][4] = \{(5,11),(6,9)\}$. Here, we see that $A[x][2]$ is obtained not by placing node `x` at the second slot but rather by placing it at the first slot and then propagating wire from there to the second slot. Also, we can see that there are two candidate solutions for $A[x][4]$, one of cost 5 and delay 11 (node `x` placed at slot 1) and the other of cost 6 and delay 9 (node `x` placed at slot 2). Finally, at node `t`, we have $A[t][4] = A^b[t][4] = \{(5,12),(6,10)\}$. At the end, we have two solutions to choose from. Assuming that a lower bound on some global circuit delay is 15 units, we would rather choose solution (5, 12) (node `x` placed at slot 1), since it is the cheapest and is also fast enough, instead of the faster (6, 10) solution (node `x` placed at slot 2).

## A. *Cost Function*

In the general case, the cost function is extremely flexible and may include, in addition to wire length cost, "placement cost" in which a cost $p_{ij}$ is incurred when cell $i$ is placed at slot $j$. This is extremely useful in our application since it allows us to give a cost "discount" if a cell is placed "on-top" of a logically equivalent cell (and thus these two cells can be unified). Thus, the solutions to the embedding problem naturally capture replication overhead.[3]

First, we construct an embedding graph as a uniform grid of feasible placement locations. Then, we assign placement costs based on local placement congestion information. Note that we allow placing a cell from the critical tree on top of some other cell in the design. The idea behind this is that the critical tree should be able to get the best "real-estate" locations and other noncritical components should "move out." Of course, high cost is assigned to congested areas, so those areas are utilized only if needed for performance reasons (Section IV gives more details about overall optimization flow and timing-driven legalization step). Sometimes a designer may wish that certain areas of the design remain undisturbed. This is easily achieved by marking appropriate locations in the embedding graph as blocked.

In addition to this placement cost, we have another component that gives a cost "discount" to a cell at a corresponding tree node that is placed on top of its old location in the placement (or on top of a cell logically equivalent to it). In this way, we can bias the embedder to use placement locations that are less likely to require cell replication.

To each edge in the graph we assign wire cost. The ability to work on arbitrary graphs implicitly allows support of nonuniform target technology structures (e.g., consideration of routing blockages is straightforward as are nonuniform FPGA routing architectures).

At this point, it is worth mentioning one important difference between fanout and fanin tree embedding. While doing fanout tree embedding, usually, we do not care if Steiner points of the same tree overlap one another. In the fanin tree case, Steiner points translate to gates. During the embedding process, it is possible that gates on various levels of the tree hierarchy end up sharing the same placement location. We propose two different approaches to address this problem.

[3]Besides, a simple linear program as in [12] can solve special cases of the embedding problem but seems incapable of solving it in the generality we have described here.

The first approach is to add an extra bit to the solution signature. For each initial solution (obtained at line b2) and every candidate solution obtained through the join operation (line c2), we set that bit to 1 (i.e., "branching solutions"). For every candidate solution obtained in the wavefront propagation phase by augmenting (line d10), we set this bit to 0. Going back to the join operation (line c2), if the sum of these bits for candidate solutions that we are joining is 0, then the join operation (and placing parent gate at that location) will not produce an overlap. In other words, we should not try to join "branching" solutions. In this way, one can prevent overlap of parent and child cells in the tree hierarchy (though it cannot, in general, guarantee zero overlap). As a side note, one has to be careful about pruning suboptimal solutions since placement bits have to be considered as well (for details, see [19]). In hierarchical FPGAs with multiple look-up tables (LUTs) per each configurable logic block (CLB), the embedder assigns LUTs to CLBs. In this case, some "overlap" is acceptable and desirable (up to the limit of the CLB capacity). Assuming that CLB capacity is $n$ LUTs, and $m$ are already occupied, then join should not be allowed if the sum of these bits is greater than $n - m$.

The second approach is to allow overlap of internal tree nodes and then legalize the placement after the embedding process. In our flow, we already allow placement overlaps with other gates outside of the critical tree to avoid overconstraining the solution space. For example, if a cell on a critical path desires an already occupied location, it makes sense to place it there and let the legalizer handle the overlap rather than force it into a (possibly distant) empty slot. Since we are already running placement legalization in our current flow (see Section IV), this approach does not introduce any additional steps. In the experiments, we use the second approach. In our experience, the timing-driven legalizer introduces a very modest solution degradation.

## B. Delay Model

In our experimental setup, we use essentially the same placement-level delay estimator as used by VPR [1] and [18]. For the target FPGA architecture under consideration, all the switches are buffered and interconnect resources are uniform. As a result, RC effects are localized and thus the interconnect delay is reasonably approximated by a linear function of the Manhattan length of the interconnect. In other words, each edge in the embedding graph is annotated with propagation delay and each vertex with intrinsic delay.

## C. Solution Signature

Since we are simultaneously optimizing delay and cost under the linear delay model, it is sufficient that the solution signature in the resulting DP approach contains only two values: cost and delay. A candidate solution (embedding) for a subtree rooted at node $i$ in the tree with node $i$ placed at vertex $j$ in the embedding graph is represented by its signature $(c, t) \in A[i][j]$, indicating that this subsolution incurs cost $c$ and has the latest arrival time $t$ at $i$. Solutions at leaves are initialized to have a cost of zero and arrival times as specified by the problem instance [zero for primary inputs (PIs) and FFs, and the latest arrival time computed by static timing analysis for other leaves].

In the bottom-up DP procedure, we must combine candidate solutions from subtrees to form new candidate solutions (compute $A^b$ sets). At an internal node $i$ in the tree and vertex $j$ in the graph, we join (line c2) subtree solutions as

$$c = p_{i,j} + c_1 + c_2 + \cdots + c_k$$
$$t = \max(t_1, t_2, \ldots, t_k)$$

where $k$ is the number of inputs for gate at $i$ and $p_{i,j}$ is the placement cost. Note that the pseudo-code in Fig. 6 is shown for binary trees only. We have extended the algorithm to support arbitrary number of inputs.

At the root, we obtain a set of solutions with cost versus delay tradeoff. From the tradeoff curve, we pick the cheapest solution that is faster than the precomputed lower bound on the best possible worst delay of the circuit [which is in general limited by distance between PIs and primary outputs (POs) and number of logic blocks in between], i.e., the cheapest solution that is fast enough.

## D. Solution Signature for Nonlinear Delay Models

In principle, the embedding algorithm can use more general delay models (which may be useful in, for example, the ASIC domain). In [6] and [7], the tree embedding algorithm is applied to interconnect synthesis and presented for the Elmore delay model [3]. Same ideas can be applied in our case with some adjustments.

The DP approach for fanin tree embedding in the previous section is based on 2-D variant of the tree embedding problem presented in [6] and [7]. A candidate solution has signature $(c, t)$, indicating that this subsolution incurs cost $c$ and has latest arrival time $t$. The variant is called 2-D since the solution signature contains two parameters that we are simultaneously optimizing (in this case, cost and max arrival time). In [7], one can find a description of a more complex three-dimensional (3-D) case, i.e., simultaneously optimizing three parameters. The solution signature in the fanout case was a ($p$, $c$, $q$) triple, where $p$ represented solution cost, $c$ was downstream load, and $q$ was the required arrival time or slack.

In the fanin tree case, we propagate solutions from inputs toward the sink, so for the Elmore delay model, our solution signature would be represented as a ($c$, $r$, $t$) triple, where $c$ is cost, $r$ is upstream resistance (up to the driving gate and including the output resistance of that gate), and $t$ is the signal arrival time. In addition, each edge in the embedding graph is labeled with its resistance and capacitance.

Delay of a wire segment can be easily calculated as

$$d_{uv} = c_{uv}\left(R(u) + \frac{r_{uv}}{2}\right)$$

where $d_{uv}$ is the delay of the wire segment, $c_{uv}$ is the wire capacitance, $R(u)$ is the cumulative upstream resistance (up to the driving gate and including the output resistance of that gate), and $r_{uv}$ is the wire resistance.

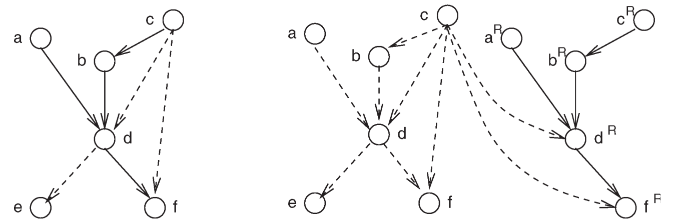

Fig. 8. Replication tree example.

The join operation for subtree solutions is shown as

$$c = p_{i,j} + c_1 + c_2 + \cdots + c_k$$
$$r = R_{out}$$
$$t = \max(t_1, t_2, \ldots, t_k).$$

where $R_{out}$ is the output resistance of the target gate.

The top-level algorithms for the 2-D and 3-D cases are almost the same, and the major difference is how some of the primitives are handled. To better understand the proposed solution, let us describe differences between the 2-D and 3-D algorithms. The main difference is in determining the solution dominance property. Given a solution signature, a solution is nondominated if no other solution is superior in all dimensions. Any dominated solution may be discarded. At line `d7` of the pseudo-code, in the 2-D case, if we keep solutions ordered in the list (costs increasing while arrival times are decreasing), the dominance test is trivial (check the tail of the list) and takes constant time. However, the 3-D case is more complex, so balanced binary search trees are needed for efficient dominance test. Another difference is in the implementation of the join primitive at line `c2`, which joins two sets of subsolutions. In the 2-D case, the join could be performed by linear traversal through ordered solution lists, while in the 3-D case we have to perform a cross product of all subsolutions in those sets. A more detailed description of these operations can be found in [7].

## III. Replication Tree

Since most circuits do not have large fanin trees due to reconvergence, we have devised the replication tree, which enables us to induce large fanin trees in a logically equivalent circuit. The idea is best illustrated by an example. In the left part of Fig. 8, we have a portion of a circuit with a set of edges in bold. These edges form a tree with all edges pointing toward the root (`f`). Note that in the left figure, this tree does not form a valid fanin tree due to reconvergence. To induce a fanin tree, we (temporarily) make a copy of each node in the tree (`f`, `d`, `a`, `b`, `c`). If the original cell is $v$ and the copy is $v^R$, we assign connections as follows. Let $u_1, \ldots, u_k$ be the inputs to $v$. If $(u_i, v)$ is a tree edge, then $v^R$ receives its $i$'th input from $u_i^R$; otherwise, it receives its $i$'th input from $u_i$.

This construction is applied to the circuit in the figure, resulting in the structure on the right and yielding a fanin tree subcircuit formed by replicated cells. Notice that cells $\mathtt{d}^R$ and $\mathtt{f}^R$ connect to `c` rather than $\mathtt{c}^R$—otherwise, the replicated cells would not form a proper fanin tree (i.e., it is a Leaf-DAG because, for example, the "leaf" node `c` connects to two cells in

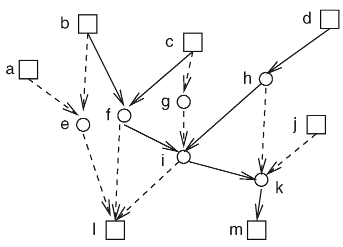

Fig. 9. Example of $\epsilon$-SPT.

the tree; however, since the timing properties of `c` are fixed and known, this does not complicate the embedding).

From the construction, we have two claims. First, if we modify the circuit in this way (again, temporarily), the result is functionally equivalent; this is clear from the construction. Second, the set of replicated nodes forms the internal vertices of a legitimate fanin tree that can be embedded.

The temporary nature of replication can now be tied to the placement cost that we have incorporated into the embedding formulation, and this point is crucial. We mentioned that placing a node coincidentally with a logically equivalent node receives a "discount." In the context of replication, this should now become clear—if the embedder places $v^R$ at the same location as $v$, there is no replication and, thus, we implicitly only replicate the cells that yield the most significant improvement. A special case may occur if node $v$ has a fanout of one. Then we still replicate, but all placement locations receive a discounted cost, since no actual replication will ever occur.

Furthermore, over the course of multiple optimizations, we may have more than two copies of a cell. Placement costs are assigned accordingly in such situations (i.e., placement with any logically equivalent cell receives a discounted cost, not only with the immediate source of the replication).

Clearly, there are many trees in a timing graph that we may use to generate a replication tree. For timing optimization, it is natural to focus on trees with slow paths. The SPT can be thought of as the result of finding a longest paths tree from the critical sink in the timing graph with the edges reversed (i.e., finding the shortest paths tree in the reversed graph with the delay values negated). Finding this tree is trivial once static timing analysis has completed.

Similarly, an $\epsilon$-SPT is a subset of the SPT that includes only cells with paths within $\epsilon$ of the current critical path delay. This allows us to focus on the most critical portions of the fanin cone of the critical sink. An example of $\epsilon$-SPT is given in Fig. 9. Circuit inputs are `a`, `b`, `c`, `d`, and `j`. Outputs are `l` and `m`. Sink `m` has been identified as critical. The edges of the $\epsilon$-SPT are shown with solid lines and dashed edges represent circuit connectivity. Note that `g` and `j` are not contained in the $\epsilon$-SPT.

## IV. Optimization Flow

Having introduced the core components of our approach, we now give an overview of how they can be put together into a complete optimization flow. As in [1], we begin from a valid timing-driven placement produced by VPR [18].

The top-level view of how our optimizer relates to VPR is given in Fig. 10: VPR is invoked to give an initial placement;

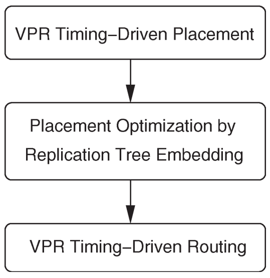
VPR Timing–Driven Placement
Placement Optimization by Replication Tree Embedding
VPR Timing–Driven Routing

Fig. 10. Optimization flow.

```
Algorithm: RT-Embedding(C,P)
       C: Circuit; P: Placement
a1     while(improvement)
a2         Static_Timing_Analysis$(C, P)$
a3         $T \leftarrow$ Extract_Replication_Tree$(C, P)$
a4         Embed_Tree$(T)$
a5         Post_Unify$(C, P, T)$
a6         Legalize$(C, P)$
a7     endwhile
```

Fig. 11. Replication tree embedding.

we optimize the placement by replication; we then pass it to the VPR detailed router to accurately assess the results.

Thus, our approach is not currently intended to replace any existing optimization steps in the flow but rather to complement them. The core replication procedure is focused on highly timing-critical subcircuits and thus, while the embedding algorithm is nontrivial, the runtime penalty for using such a sophisticated algorithm is very small in the scope of the entire flow (this has been verified experimentally).

Fig. 11 illustrates the pseudocode of our main optimization loop. In each iteration, we start with a static timing analysis to identify the most critical sink. From an $\epsilon$-SPT, we extract a replication tree with the critical sink at its root. Then, we pass the tree to the embedder, which produces a family of solutions with cost/delay tradeoff. After the embedding phase, we analyze the resulting configuration for possible postprocess unifications (see Section V-C) since it is possible to have equivalent cells that are not exactly coincident but close enough that unifying them would not harm timing. The chosen solution from the tradeoff curve will guide the solution extraction algorithm to determine which cells need to be replicated or just relocated if no replication is necessary. As a final step, we invoke placement legalizer to resolve any possible cell overlaps that may have occurred.

## V. Implementation Details

In this section, we present some of the algorithmic details such as the timing-driven legalizer that we implemented and the parameters that we used during replication tree construction. Also, we present some algorithmic enhancements such as postprocess cell unification and FF relocation.

### A. Timing-Driven Legalization

After embedding and replication, it is likely that some cells overlap in the placement. The purpose of the legalizer is to

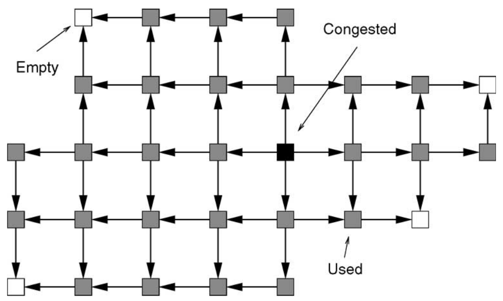

Fig. 12. Gain graph example. Slot marked Empty can accommodate an additional cell. Slot marked Used is completely full; no additional cells can fit in. Slot marked Congested contains more cells than its capacity.

resolve these overlaps by moving cells from congested to empty locations. We observe that by moving cells that are on a critical path one may degrade circuit performance. In order to minimize perturbations to the placement and limit degradation of the timing achieved in the embedding phase, we have adapted the ripple move strategy from [10]. The technique has been generalized to incorporate both timing and wiring information.

The legalizer is invoked after each embedding phase. During embedding, it is possible that we replicate and/or move multiple cells, so we may have more than one violation in the placement. If an overlap-free placement is achievable (i.e., there are enough free slots), the legalizer will resolve one overlap at a time until the entire placement is legal.

In the procedure, we first identify an overlap location. If we have more than one overlap, we pick the first one we encounter while we scan the placement for overlaps. We then identify up to four closest free slots (one slot in each quadrant, if they exist, assuming that the center is at the congested slot). Next, we identify which of those free slots will be used for legalization. To do this, we construct a gain graph (Fig. 12), which has monotone paths from a congested slot to free slots. Each edge is labeled by the "gain" that we get by moving a cell from that slot to the neighboring slot (in the direction toward the target free slot).

Gain is computed as the difference of costs of having a cell at the current and the neighboring slot. This cost has wire and timing components. Wire cost is the sum of the estimated wire lengths of the net for which the current cell is the driver and those nets that are inputs of the cell. Wire length estimation is given by the half-perimeter metric augmented by a net size coefficient from [18].

Timing cost is computed as the squared delay of the slowest path through the current cell if such delay approaches the current critical delay (within 40% in our experiments) and zero otherwise (in this way, moves that are likely to make a near-critical path worse are discouraged). The cost of a cell at a particular location is a composite of timing and wire cost, i.e.,

$$C = \alpha C_T + (1 - \alpha) C_W.$$

The gain of moving a cell from its current slot to a neighboring slot is

$$\text{Gain} = C_{\text{curr}} - C_{\text{new}}.$$

Once we have constructed the gain graph, we find the max-gain path in the graph and use the target slot with the highest gain for ripple move legalization. Note that to minimize perturbations of the placement we move cells at most one slot during a ripple move. Another motivation for this is that the embedder has a much stronger algorithm for optimizing cell locations, so we want to keep cells as close to those locations as possible. Note that the best gain value could still be negative (i.e., we may lose some quality/performance). Since the main goal of our optimization technique was to improve timing, the value of $\alpha$ that we used in our experiments was 0.95.

During ripple moves, it is possible that a cell may be moved to a slot that contains one of its logically equivalent cells. In that case, we unify them and stop the current pass of a single overlap legalization.

## B. $\epsilon$-SPT

As discussed previously, we use $\epsilon$-SPT to guide replication tree construction. The value of $\epsilon$ is initially set to zero and is dynamically updated in the main loop of optimization flow. Since our approach has no randomized components, when no improvement is found for a tree rooted at a particular critical sink, we would not be able to improve any further in subsequent iterations since the same sink will still be critical and the same tree will be selected. We address this problem by dynamically increasing the value of $\epsilon$ when nonimprovement occurs. The intuition is that as the extracted tree becomes larger, we have more freedom in tree embedding optimization.

## C. Postprocess Unification

Once we have an embedding of a replication tree, if we seek a timing superior solution, some cells may be placed close to logically equivalent cells but not quite on top of them. In this case, implicit cell unification will not occur. However, it is possible that some of the equivalent cells lie on noncritical paths and that their fanouts could receive the appropriate signal from the newly replicated cell without degrading their arrival time (sometimes delay can even improve).

As a postprocess, for each newly replicated cell, we examine all logically equivalent cells. If any fanout cell of those equivalent cells can improve its arrival time by taking the corresponding input from a newly replicated cell, we reassign it to the new replica. In this way, we can improve delay on paths that were not explicitly captured by the replication tree. It is possible that, in this process, some of the equivalent cells end up with no fanouts (i.e., no cell is using their output). In this case, such a cell is deleted as redundant. After deletion, we may have induced the same condition to its parent. Then the parent becomes redundant. This test is applied recursively.

An example of unification in practice is when we have a nontree structure (DAG) on one side of the FPGA. Then, in each iteration, a part of the DAG is extracted as a replication/fanin tree, optimized, and placed further away so that replication must occur. In consecutive iterations, the other parts of the DAG slowly migrate to the other side. Finally, the entire DAG can migrate to the other side, in which case replications, although necessary for intermediate solutions, are now completely redundant. Unification naturally handles this anomaly. Fig. 13 shows an example of unification. Before optimization, we had cell a and its replica $a^R$. Cell a gets relocated to the proximity of cell $a^R$. Timing analysis reveals that the fanins of $a^R$ can get signal from a without degrading the worst delay through it and so we perform a unification.

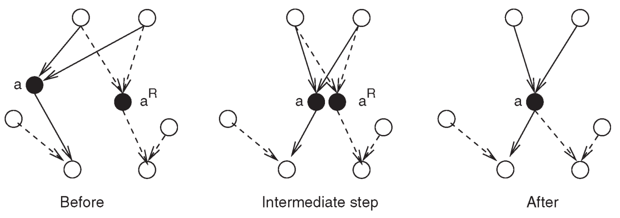

Fig. 13. Example of cell unification.

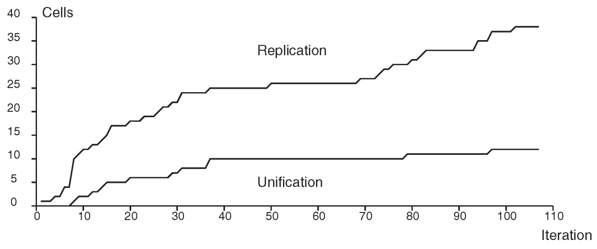

Fig. 14. Replication statistics for circuit `ex1010`.

Fig. 14 shows the relation between replicated and unified cells for circuit `ex1010`. The optimization took 106 iterations, and during that time 38 cells were replicated but 12 were unified giving total of 26 replications at the end.

## D. FF Relocation

If the circuit that we are optimizing contains FFs, it is possible that FF placement is the limiting factor for further optimization. To address this issue, we use a feature of the S-Tree algorithm [6], which performs simultaneous driver placement at no extra run-time overhead; in our case, this translates to simultaneous sink placement. Due to the deterministic nature of the approach, if the algorithm is not able to improve timing of a particular critical sink, that sink will be selected again as critical in the next iteration. If this occurs and the critical sink is an FF, then we allow it to move. We extract the tradeoff curve that is composed of solutions at all possible locations for the critical sink (not just the original location) and choose the solution minimizing the arrival time without introducing large delay penalty on other paths that touch that FF (for example, nets for which this critical FF is a source). However, we do allow some degradation of global delay in such a step with the idea being that the relocation will enable otherwise unachievable solution quality in subsequent iterations. Because of such intermediate degradation, we save the best solution seen until this point so that we can always report the best solution encountered.

## VI. Effects of Reconvergence

Although the presented approach was able to achieve larger improvement than [1] in minimizing critical delay, a significant

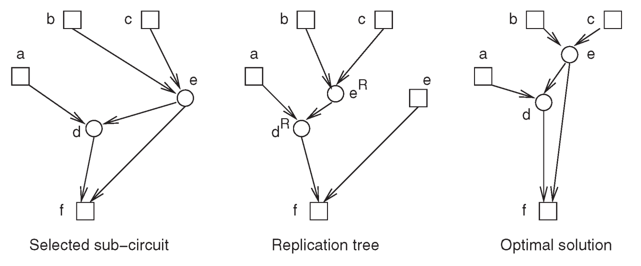

Fig. 15. Example of reconvergence effect on the replication tree.

number of circuits from the test suite have nonmonotone critical paths and thus additional improvement may be possible.

In this section, we analyze the effects caused by reconvergence in the netlist on replication-tree-based optimization. These effects prevent the fanin tree embedding algorithm from "straightening" some paths. This particular problem is related to the way in which reconvergence affects achievable maximum arrival time by replication tree embedding. While we still use the same replication tree construction algorithm, we propose a new fanin tree embedding objective, which has more success in addressing reconvergence issues by optimizing subcritical paths. Since the optimization framework is iterative, we optimize subcritical paths hoping to create better optimization choices for subsequent iterations of the flow.

Recall that the embedding algorithm as presented thus far derives optimal solutions with respect to the cost/max-arrival-time tradeoff. However, when reconvergence is present, near-critical paths can prevent the algorithm from improving any paths. Let us use an example to illustrate this issue. In Fig. 15 on the left, we have an extracted critical subcircuit. Inputs are nodes `a`, `b`, and `c`. Internal nodes are `d` and `e`, and the sink is node `f`. Nodes that are fixed are represented by squares and movable/replicable nodes are represented with circles. Let us assume that nodes `a`, `b`, and `c` have a signal arrival time of zero. Also, let the arrival time at `e` be 2 units, and 3 units at sink `f`, while the delay of the path from `a` to `d` is 1 unit. The replication tree construction procedure applied to the net on the left will yield the replication tree shown in the middle of Fig. 15. Node `d` becomes replicable $\mathtt{d}^R$, but since there is a reconvergence on node `e` we will have two nodes: movable/replicable $\mathtt{e}^R$ and fixed node `e`, where reconvergence "breaks." When we apply the optimal cost/max-arrival-time tree embedding algorithm to this replication tree, the fastest solution with smallest cost will have internal nodes placed at exactly the same location as they originally were (on the left in Fig. 15). The path from `a` to `f` is not critical. Since the path from `e` to `f` is monotone, it cannot be improved any further and the initial signal arrival time at `e` is set to 2 units, so the arrival time at `f` will remain 3 units. The embedder is not going to overoptimize the subcritical path that goes through node $\mathtt{e}^R$, and this will yield minimum placement cost for nodes $\mathtt{e}^R$ and $\mathtt{d}^R$, which is achievable by placing those nodes on top of `d` and `e`, respectively.[4] However, the solution on the right of Fig. 15 has clearly better delay (i.e., all monotone paths) and potentially uses less wiring resources.

[4]Note that there are some rare exceptions where savings in wiring resources could outweigh the replication cost for $\mathtt{e}^R$. In this case, the path could be straightened. In subsequent iterations, node `e` could move as well, and it may become possible to unify it with $\mathtt{e}^R$.

To address this problem, we propose a modified fanin tree embedding algorithm that overoptimizes subcritical paths.

## A. *Overoptimized Tree Embedding*

To address the optimality issue presented above, we propose a modification to the embedding problem formulation. The optimization criteria are to place internal tree nodes minimizing cost subject to the composite arrival time constraint at the root. Going back to the example in Fig. 15, if we could optimize delay of the second critical path, then reconvergence could be broken and in the next iteration we could potentially optimize the remaining path of the critical net and achieve global delay improvement.

In this scenario, we would like to optimize cost, critical delay, and chosen path delay. This suggests a triple $(c,\ t,\ t2)$ (cost, critical arrival time, and subcritical arrival time) as a solution signature. It seems that using the 3-D variant of the embedding tree algorithm (as explained in Section II-D) would solve the problem. Using 3-D embedding would certainly (and significantly) affect the run time of the proposed approach. Also, it would present a significant problem if we would try optimizing additional two or three critical paths, since we would have to increase the number of dimension in the embedding algorithm that would yield still higher complexity. Perhaps more importantly, intuitively, the relationship between the critical arrival time and the subcritical arrival time seems to not be symmetric. Consider two candidate solutions $a$ and $b$ with identical cost, but where $a$ has better max arrival time and $b$ has better subcritical arrival time; the general dominance property would retain both solutions, even though solution $a$ seems preferable. As a result, we propose the use of subcritical arrival times as tiebreakers in a lexicographic strategy described below.

Observe that subcritical delay cannot be larger than critical delay. This means that we can still order nondominated solutions by increasing cost and decreasing arrival times and use dominance test from the 2-D variant of the tree embedding algorithm. With this approach, we can extend the solution signature further to include more subcritical paths, as long as we treat delay as lexicographically ordered values.

As mentioned earlier, in the bottom-up DP procedure, we must combine candidate solutions from subtrees to form new candidate solutions. At internal node $i$ in the tree and vertex $j$ in the graph, we join subtree solutions as

$$\begin{aligned} c &= p_{i,j} + c_1 + c_2 + \cdots + c_k \\ t &= \max(t_1, t_2, \ldots, t_k) \\ t2 &= \max\left(\{t_1, t_2, \ldots, t_k\} \cup \{t2_1, t2_2, \ldots, t2_k\} \setminus \{t\}\right) \end{aligned}$$

where $k$ is the number of inputs for gate at $i$ and $p_{i,j}$ is the placement cost. We refer to this version as `Lex-2` (we have two lexicographically ordered delay values).

Having virtually no restrictions on the number of subcritical paths that we can overoptimize, we have implemented `Lex-3`, `Lex-4`, and `Lex-5` in a similar fashion. Due to multiple reconvergences in the circuit description, it is possible that

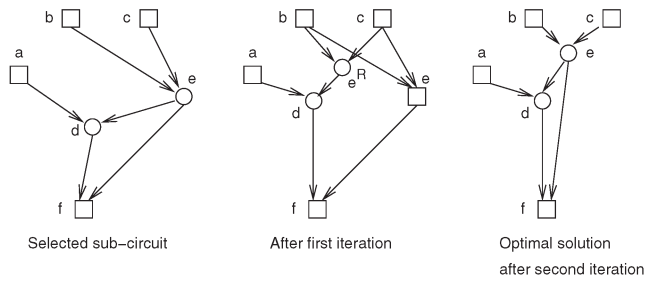

Fig. 16. Example of applying the Lex-3 version of the algorithm on the same subcircuit in two consecutive iterations. The subcircuit on the left is transformed to the one in the middle by overoptimizing subcritical paths and replication. In the second iteration, replicated cells get unified and an optimal configuration on the right is achieved.

many paths in the replication tree have exactly the same delay and considering more subcritical paths may be needed in order to resolve this issue. A subtree solution join for Lex-3 is shown as

$$\begin{aligned} c &= p_{i,j} + c_1 + c_2 + \cdots + c_k \\ t &= \max(t_1, t_2, \ldots, t_k) \\ t2 &= \max(\{t_1, t_2, \ldots, t_k\} \cup \{t2_1, t2_2, \ldots, t2_k\} \setminus \{t\}) \\ t3 &= \max(\{t_1, t_2, \ldots, t_k\} \cup \{t2_1, t2_2, \ldots, t2_k\} \\ &\qquad \cup \{t3_1, t3_2, \ldots, t3_k\} \setminus \{t\} \setminus \{t2\}). \end{aligned}$$

One other scheme of interest is to additionally optimize delay from some specific input of the replication tree. In the replication tree, the actual inputs are identified as leaves of the tree that have zero signal arrival time (in this way, we can distinguish them from the leaves that are created as reconvergence terminators). Among those inputs, we can identify the critical input as one with the largest downstream delay (which is easily obtained by performing static timing analysis). The resulting solution signature is ($c$, $t$, $tc$, $w$), where $tc$ is the delay from the critical input and $w$ is a weight factor that is not included in the dominance test. The weight factor is set to 1 for the critical branch and 0 otherwise. The join operation for Lex-mc (max and critical) is

$$\begin{aligned} c &= p_{i,j} + c_1 + c_2 + \cdots + c_k \\ t &= \max(t_1, t_2, \ldots, t_k) \\ tc &= tc_1 * w_1 + tc_2 * w_2 + \cdots + tc_k * w_k \\ w &= w_1 + w_2 + \cdots + w_k. \end{aligned}$$

Going back to the example in Fig. 15, we are able to achieve the solution on the far right by applying the Lex-3 variant of the embedding algorithm. In the first iteration, paths from b and c to f will get overoptimized with respect to the dominating arrival time from e to f and we will obtain a solution similar to the figure in the middle. Then, in the next iteration, since reconvergence is "broken," we will optimize paths from b and c through e to f. Due to cost minimization, e would most likely be placed on top of e$^R$ and thus unified with it, achieving the configuration on the right. Fig. 16 shows how, in two consecutive iterations in the optimization flow, the subcircuit on the left can improve delay by applying the Lex-3 version of the algorithm.

## VII. Experimental Evaluation

We have implemented the replication tree embedding algorithm and performed some initial experiments to evaluate its effectiveness. The experiments were conducted in a LINUX environment on a PC with an Intel PentiumM 1.3-GHz CPU and 256 MB of RAM. The main criteria of interest are the maximum delay of the circuit (i.e., clock period), wire length, and the amount of replication. We compared our approach with Timing Driven VPR [18] and with the local replication algorithm from [1]. Table I shows baseline results for 20 MCNC benchmark circuits. As mentioned earlier, circuits were placed and later routed using timing-driven VPR.

The circuits were placed on the minimum square FPGA able to contain the circuit. As in [18], we use the definition of low-stress routing as routing where FPGA has about 20% more routing resources available than the minimum required to successfully route the circuit. Also from [18], infinite-resource routing is when the FPGA has unbounded routing resources. It is argued in [18] that the former represents the situation of how FPGA will be routed in practice and the latter is a good placement evaluation metrics. For post-place-and-route experiments, we present both low-stress ($W_{\text{ls}}$) and infinite-resource ($W_\infty$) critical path delay numbers. Table I also contains total wire length after routing, number of LUTs, I/Os, and total number of cells. We also report the actual FPGA size we used for each circuit as well as the FPGA design density that represents the ratio of utilized LUTs and available FPGA area. Note that most of the circuits are of very high density. All but three of them have density above 95% and six of them above 99%.

### A. *Replication Tree Embedding Results*

Experimental data are shown in Table II. All values in the table are normalized to the corresponding VPR results. In the first data set, we optimized placement by local replication[5] algorithm. Since the local replication algorithm is randomized, we ran it three times and took the best result. In the second data set, we optimized the circuits using our approach (RT-Embedding).

We are able to improve the critical path delay over VPR for all circuits in the test suite. The best delay reduction of 36% was achieved for circuit pdc. The average delay reduction is 14.2%, which almost doubles the average delay improvement of the local replication algorithm. The largest improvement over local replication is almost 19% for circuit apex2, for which local replication was not able to improve the critical path delay at all. Observe that the wire length degradation of our approach is 8.4% on average, and the average number of newly introduced cells by replication is only 0.4% of the total number of cells. One may argue that the increase in wire length is not negligible. However, perhaps more important than wire length

[5]We have verified with Beraudo and Lillis [1] that the results reported in [1] were based on wire-length-driven VPR (by mistake), while it was reported that comparisons were based on timing-driven VPR.

TABLE I
INITIAL CIRCUIT DATA AND TIMING-DRIVEN VPR RESULTS

| Circuit | Timing Driven VPR | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | crit path [ns] | | wire | | | total | FPGA | design |
| | $W_\infty$ | $W_{ls}$ | length | LUTs | I/Os | blk | size | density |
| `ex5p` | 80.59 | 81.99 | 20020 | 1064 | 71 | 1135 | 33 x 33 | 0.977 |
| `tseng` | 50.54 | 53.65 | 10495 | 1047 | 174 | 1221 | 33 x 33 | 0.961 |
| `apex4` | 72.12 | 75.41 | 22332 | 1262 | 28 | 1290 | 36 x 36 | 0.974 |
| `misex3` | 64.44 | 65.87 | 21784 | 1397 | 28 | 1425 | 38 x 38 | 0.967 |
| `alu4` | 77.20 | 81.07 | 20796 | 1522 | 22 | 1544 | 40 x 40 | 0.951 |
| `diffeq` | 55.29 | 57.49 | 15560 | 1497 | 103 | 1600 | 39 x 39 | 0.984 |
| `dsip` | 65.38 | 67.21 | 17237 | 1370 | 426 | 1796 | 54 x 54 | 0.470 |
| `seq` | 76.93 | 77.82 | 28493 | 1750 | 76 | 1826 | 42 x 42 | 0.992 |
| `apex2` | 94.61 | 95.47 | 30998 | 1878 | 41 | 1919 | 44 x 44 | 0.970 |
| `s298` | 124.20 | 127.35 | 22762 | 1931 | 10 | 1941 | 44 x 44 | 0.997 |
| `des` | 90.44 | 91.31 | 27415 | 1591 | 501 | 2092 | 63 x 63 | 0.401 |
| `bigkey` | 59.69 | 60.65 | 21074 | 1707 | 426 | 2133 | 54 x 54 | 0.585 |
| `frisc` | 119.02 | 124.61 | 61109 | 3556 | 136 | 3692 | 60 x 60 | 0.988 |
| `spla` | 111.03 | 113.57 | 68308 | 3690 | 62 | 3752 | 61 x 61 | 0.992 |
| `elliptic` | 105.96 | 108.50 | 47456 | 3604 | 245 | 3849 | 61 x 61 | 0.969 |
| `ex1010` | 184.84 | 185.56 | 70300 | 4598 | 20 | 4618 | 68 x 68 | 0.994 |
| `pdc` | 167.81 | 169.33 | 105073 | 4575 | 56 | 4631 | 68 x 68 | 0.989 |
| `s38417` | 97.20 | 100.61 | 64490 | 6406 | 135 | 6541 | 81 x 81 | 0.976 |
| `s38584.1` | 99.74 | 102.10 | 58869 | 6447 | 342 | 6789 | 81 x 81 | 0.983 |
| `clma` | 211.78 | 217.24 | 145551 | 8383 | 144 | 8527 | 92 x 92 | 0.990 |

TABLE II
COMPARISON BETWEEN LOCAL REPLICATION, RT-EMBEDDING, AND `Lex-3`

| Circuit | local replication normalized to VPR | | | | RT-Embedding normalized to VPR | | | | Lex-3 normalized to VPR | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | crit path | | wire | | crit path | | wire | | crit path | | wire | |
| | $W_\infty$ | $W_{ls}$ | length | blk | $W_\infty$ | $W_{ls}$ | length | blk | $W_\infty$ | $W_{ls}$ | length | blk |
| `ex5p` | 0.792 | 0.806 | 1.027 | 1.004 | 0.764 | 0.774 | 1.090 | 1.011 | 0.764 | 0.783 | 1.110 | 1.019 |
| `tseng` | 0.987 | 0.955 | 1.012 | 1.004 | 0.987 | 0.978 | 1.060 | 1.002 | 0.970 | 0.933 | 1.068 | 1.010 |
| `apex4` | 0.912 | 0.913 | 1.042 | 1.012 | 0.888 | 0.913 | 1.107 | 1.011 | 0.854 | 0.871 | 1.193 | 1.024 |
| `misex3` | 0.914 | 0.937 | 1.013 | 1.007 | 0.852 | 0.891 | 1.148 | 1.010 | 0.835 | 0.872 | 1.273 | 1.021 |
| `alu4` | 0.987 | 0.963 | 1.004 | 1.000 | 0.922 | 0.925 | 1.053 | 1.002 | **0.860** | 0.945 | 1.197 | 1.013 |
| `diffeq` | 1.004 | 1.000 | 1.002 | 1.003 | 0.989 | 0.969 | 1.026 | 1.001 | 0.999 | 0.990 | 1.020 | 1.002 |
| `dsip` | 0.924 | 0.938 | 1.024 | 1.001 | 0.793 | 0.804 | 1.277 | 1.001 | **0.731** | 0.822 | 1.559 | 1.001 |
| `seq` | 0.939 | 0.969 | 1.011 | 1.002 | 0.870 | 0.885 | 1.048 | 1.003 | **0.818** | 0.859 | 1.100 | 1.008 |
| `apex2` | 1.000 | 1.000 | 1.000 | 1.000 | 0.811 | 0.838 | 1.120 | 1.010 | **0.755** | 0.799 | 1.262 | 1.016 |
| `s298` | 0.937 | 0.937 | 1.029 | 1.003 | 0.915 | 0.903 | 1.034 | 1.001 | 0.875 | 0.899 | 1.066 | 1.002 |
| `des` | 0.898 | 0.895 | 1.044 | 1.003 | 0.876 | 0.876 | 1.039 | 1.001 | 0.876 | 0.886 | 1.043 | 1.002 |
| `bigkey` | 1.000 | 1.000 | 1.000 | 1.000 | 0.855 | 0.892 | 1.190 | 1.000 | 0.801 | 0.901 | 1.328 | 1.000 |
| `frisc` | 1.007 | 0.997 | 1.007 | 1.001 | 0.999 | 0.983 | 1.018 | 1.001 | 0.958 | 0.917 | 1.069 | 1.007 |
| `spla` | 0.874 | 0.889 | 1.035 | 1.005 | 0.812 | 0.824 | 1.108 | 1.008 | 0.793 | 0.829 | 1.164 | 1.008 |
| `elliptic` | 0.926 | 0.934 | 1.040 | 1.003 | 0.853 | 0.838 | 1.030 | 1.001 | **0.780** | 0.792 | 1.132 | 1.009 |
| `ex1010` | 0.861 | 0.882 | 1.044 | 1.003 | 0.818 | 0.847 | 1.148 | 1.006 | 0.795 | 0.821 | 1.144 | 1.006 |
| `pdc` | 0.707 | 0.728 | 1.031 | 1.003 | 0.641 | 0.707 | 1.072 | 1.005 | **0.624** | 0.690 | 1.142 | 1.009 |
| `s38417` | 0.974 | 0.961 | 1.004 | 1.000 | 0.930 | 0.944 | 1.017 | 1.000 | **0.840** | 0.888 | 1.069 | 1.009 |
| `s38584.1` | 0.919 | 0.927 | 1.002 | 1.000 | 0.842 | 0.839 | 1.048 | 1.001 | 0.819 | 0.845 | 1.115 | 1.000 |
| `clma` | 0.926 | 0.915 | 1.021 | 1.003 | 0.746 | 0.745 | 1.053 | 1.005 | 0.708 | 0.707 | 1.100 | 1.006 |
| **average** | **0.925** | **0.927** | **1.020** | **1.003** | **0.858** | **0.869** | **1.084** | **1.004** | **0.823** | **0.853** | **1.158** | **1.009** |
| **small avg.** | **0.941** | **0.943** | **1.017** | **1.003** | **0.877** | **0.887** | **1.099** | **1.004** | **0.845** | **0.880** | **1.185** | **1.010** |
| **large avg.** | **0.899** | **0.904** | **1.023** | **1.002** | **0.830** | **0.841** | **1.062** | **1.003** | **0.790** | **0.811** | **1.117** | **1.007** |

is routability, and our designs were always successfully routed (this is most relevant in the case of $W_{ls}$).

The runtime overhead of our approach is very modest—under 5% of the VPR flow (place and route). Note that under low-stress routing, the critical path delay is slightly worse than the case with infinite routing resources. Degradation is consistent for all circuits in the test suites and correlates with low-stress routing behavior conclusions from [18].

## *B. Additional Reconvergence Experiments*

We have performed additional experiments assuming the optimization flow that addresses reconvergence effects (described in Section VI). We compared the following algorithm variants: `Lex-mc`, `Lex-2`, `Lex-3`, `Lex-4`, and `Lex-5`[6] with the timing-driven VPR [18] and the original RT-Embedding algorithm. Due to space constraints, we decided to include complete results only for the `Lex-3` version (third data set in Table II). In Table III, we report average values for the same set of 20 MCNC benchmark circuits for all compared algorithms (as in the bottom rows in Table II). In addition, we have divided circuits into small (< 3K cells) and large (≥ 3K cells) based on the number of cells they have.

[6]In fact, we have implemented a "`Lex-N`" version of the algorithm, but for values of $N$ above 5, we cannot claim modest runtime overhead any longer.

TABLE III
COMPARISON OF AVERAGE IMPROVEMENTS (RT-EMBEDDING, Lex-mc, Lex-2, Lex-3, Lex-4, AND Lex-5)

| Algorithm | Average normalized to VPR | | | | Average for small ckts normalized to VPR | | | | Average for large ckts normalized to VPR | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | crit path | | wire | | crit path | | wire | | crit path | | wire | |
| | $W_\infty$ | $W_{ls}$ | length | blk | $W_\infty$ | $W_{ls}$ | length | blk | $W_\infty$ | $W_{ls}$ | length | blk |
| RT-Embedding | 0.858 | 0.869 | 1.084 | 1.004 | 0.877 | 0.887 | 1.099 | 1.004 | 0.830 | 0.841 | 1.062 | 1.003 |
| Lex-mc | 0.841 | 0.925 | 1.168 | 1.013 | 0.852 | 0.951 | 1.197 | 1.014 | 0.824 | 0.886 | 1.124 | 1.010 |
| Lex-2 | 0.827 | 0.869 | 1.157 | 1.008 | 0.850 | 0.889 | 1.185 | 1.010 | 0.794 | 0.838 | 1.114 | 1.006 |
| Lex-3 | 0.823 | 0.853 | 1.158 | 1.009 | 0.845 | 0.880 | 1.185 | 1.010 | 0.790 | 0.811 | 1.117 | 1.007 |
| Lex-4 | 0.825 | 0.857 | 1.152 | 1.008 | 0.848 | 0.889 | 1.175 | 1.009 | 0.790 | 0.809 | 1.117 | 1.006 |
| Lex-5 | 0.827 | 0.869 | 1.150 | 1.008 | 0.849 | 0.901 | 1.168 | 1.008 | 0.795 | 0.823 | 1.124 | 1.008 |

From Table III, we can see that Lex-3 has the best average delay improvement among all the proposed algorithms. We also observed that the average improvement for larger circuits is better than for smaller ones, which is encouraging since majority of the circuits today belong to the large group. The average improvement over RT-Embedding for large circuits is 4% of the initial (unoptimized) delay. Indeed, this may look as a small improvement, but it is averaged over 20 circuits and not all could be equally improved. The best improvement is 9% for circuit s38417, improving from 0.93 to 0.84 (see Table II). For elliptic we have 7% improvement, and about 6% for alu4, dsip, seq, and apex2. It should be pointed out that for circuits misex3, diffeq, dsip, des, bigkey, and s38584.1, we have reached a theoretical lower bound, i.e., all FF to FF paths are monotone (assuming fixed FF locations). Most of the circuits in this test suite have very high density; all but three of them have density above 95% and six of them above 99%. For circuits ex5p, apex4, seq, spla, and ex1010, we ran out of free slots for replication and thus had to terminate early (for some of them, we were able to replicate under 1% of the total number of cells).

The best improvement over VPR is almost 38% for circuit pdc, and we have nine circuits with improvement larger than 20%. The number of cells introduced by replication is on the average only 0.9% of the total number of cells. However, usage of wiring resources increased to 15.8% on the average compared to the 8.4% increase of the RT-Embedding approach. It looks a bit surprising that such a small amount of replication can yield a wiring overhead of this magnitude. Since our assumption was that we are optimizing circuits of high cell density, postprocess unification was designed to be very aggressive in attempts to unify replicated cells as long as they do not violate current critical delay, hoping to clean-up intermediate replications as much as possible (Section V-C presents some statistics on unification). The biggest wiring overhead is almost 56% for circuit dsip. It turns out that a cell density of dsip is only 47%, and it is not congested at all. Similarly, bigkey has the second largest wiring overhead of almost 33%, and its density is also low, only 58.5%. This suggests that we have to revisit the unification strategy for circuits of smaller cell density.

The runtime overhead is still under 5% of the time of VPR flow (place and route) and comparable to RT-Embedding.

## VIII. DISCUSSION

We have shown that the reconvergence in circuit description presents an obstacle for further timing optimization by RT-Embedding. The proposed enhancements by optimizing subcritical paths do seem to address the reconvergence issue, but still need further improvements.

The issue of most concern is the potential overuse of wiring resources. We believe that the aggressive unification strategy creates excessive wire usage problems, which may lead to routing problems and degradation of circuit performance due to routing congestion (i.e., routes need to take detours around congested regions, thus increasing interconnect delay). To further analyze this problem, we have examined the sizes of trees that were selected for embedding. They range from a couple of cells up to almost a thousand cells per tree. We found no clear pattern that could connect tree sizes with the increase of wiring resources.

The tree embedding algorithm itself can be used to better address the routing issue, since to a certain extent it does perform global routing. We could run the routing algorithm on unoptimized circuit, and then use the actual channel occupancy to assign wire costs in the embedding graph, which is then used by the tree embedder. In this way, the embedder is biased to place cells in regions with smaller wire utilization.

Another anomaly that we have observed is that some replication trees are constructed from subcircuits that do not reconverge. In those cases, optimizing subcritical paths seems unnecessary. This may explain why Lex-5, which performs more powerful optimization, does worse than Lex-3 on the average, since it may overoptimize too many subcritical paths that turned out to really be noncritical, thus consuming more free cell space and routing resources. A possible solution would be to invoke different versions of the tree embedding algorithm based on the amount of reconvergence that we encounter while constructing the replication tree.

Despite these issues, the experiments do demonstrate significant potential of the overall approach with respect to clock period reduction.

## IX. CONCLUSION

We have presented a general and robust approach to timing-driven placement-coupled replication. Two items form the core of the approach. First, we presented an efficient algorithm for optimal fanin tree embedding under a general cost model. Second, we proposed the replication tree for inducing large subcircuits, which can be optimized by the embedder. The approach has a number of interesting properties including implicit unification of logically equivalent cells.

In addition, we address issues that arise due to reconvergence in the circuit specification. We build on the replication tree idea and enhance the timing-driven fanin tree embedding algorithm to optimize subcritical paths.

Around these core ideas, we have built an optimization engine for the FPGA domain and demonstrated very promising preliminary experimental results.

Finally, we argue that the general ideas in this work hold great promise beyond the context studied here. We suggest that the techniques can provide useful bridges between placement, routing, and logic (re)synthesis. Further, the graph-based modeling of the placement target would seem well suited to many practical problems (e.g., placement in the context of heterogeneous FPGA routing architectures).

## REFERENCES

[1] G. Beraudo and J. Lillis, "Timing optimization of FPGA placements by logic replication," in *Proc. 40th DAC*, Anaheim, CA, Jun. 2003, pp. 196–201.

[2] S. Devadas, A. Ghosh, and K. Keutzer, *Logic Synthesis*. New York: McGraw-Hill, 1994.

[3] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *J. Appl. Phys.*, vol. 19, no. 1, pp. 55–63, Jan. 1948.

[4] W. Gosti, S. P. Khatri, and A. L. Sangiovanni-Vincentelli, "Addressing the timing closure problem by integrating logic optimization and placement," in *Proc. ICCAD*, San Jose, CA, Nov. 2001, pp. 224–231.

[5] W. Gosti, A. Narayan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Wireplanning in logic synthesis," in *Proc. ICCAD*, San Jose, CA, Nov. 1998, pp. 26–33.

[6] M. Hrkić and J. Lillis, "S-Tree: A technique for buffered routing tree synthesis," in *Proc. 39th DAC*, New Orleans, LA, Jun. 2002, pp. 578–583.

[7] ——, "Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost, congestion and blockages," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 4, pp. 481–491, Apr. 2003.

[8] ——, "An approach to placement-coupled logic replication," in *Proc. 41th DAC*, San Diego, CA, Jun. 2004, pp. 711–716.

[9] S. W. Hur, A. Jagannathan, and J. Lillis, "Timing-driven maze routing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 2, pp. 234–241, Feb. 2000.

[10] S. W. Hur and J. Lillis, "Mongrel: Hybrid techniques for standard cell placement," in *Proc. ICCAD*, San Jose, CA, Nov. 2000, pp. 165–170.

[11] J. Hwang and A. E. Gamal, "Optimal replications for min-cut partitioning," in *Proc. ICCAD*, Santa Clara, CA, Nov. 1992, pp. 432–435.

[12] M. Jackson and E. Kuh, "Performance-driven placement of cell based IC's," in *Proc. 26th DAC*, Las Vegas, NV, Jun. 1989, pp. 370–375.

[13] C. Kring and A. R. Newton, "A cell-replicating approach to mincut-based circuit partitioning," in *Proc. ICCAD*, Santa Clara, CA, Nov. 1991, pp. 2–5.

[14] J. Lillis, C. K. Cheng, and T. T. Y. Lin, "Algorithms for optimal introduction of redundant logic for timing and area optimization," in *Proc. IEEE ISCS*, Atlanta, GA, May 1996, pp. 452–455.

[15] L. T. Liu, M. T. Kuo, C. K. Cheng, and T. C. Hu, "A replication cut for two-way partitioning," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 5, pp. 623–630, May 1995.

[16] J. Lou, A. H. Salek, and M. Pedram, "An exact solution to simultaneous technology mapping and linear placement problem," in *Proc. ICCAD*, San Jose, CA, Nov. 1997, pp. 671–675.

[17] W. K. Mak and D. F. Wong, "Minimum replication min-cut partitioning," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 16, no. 10, pp. 1221–1227, Oct. 1997.

[18] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *Proc. Int. Symp. Field Program. Gate Arrays*, Monterey, CA, Feb. 2000, pp. 203–213.

[19] R. Murgai, "Layout-driven area-constrained timing optimization by net buffering," in *Proc. ICCAD*, San Jose, CA, Nov. 2000, pp. 379–386.

[20] I. Neumann, D. Stoffel, H. Hartje, and W. Kunz, "Cell replication and redundancy elimination during placement for cycle time optimization," in *Proc. ICCAD*, San Jose, CA, Nov. 1999, pp. 25–30.

[21] K. Schabas and S. D. Brown, "Using logic duplication to improve performance in FPGAs," in *Proc. Int. Symp. Field Program. Gate Arrays*, Monterey, CA, Feb. 2003, pp. 136–142.

[22] Y. Shiloach, "A minimum linear arrangement algorithm for undirected trees," *SIAM J. Comput.*, vol. 8, no. 1, pp. 15–32, Feb. 1979.

[23] A. Srivastava, R. Kastner, and M. Sarrafzadeh, "Timing driven gate duplication: Complexity issues and algorithms," in *Proc. ICCAD*, San Jose, CA, Nov. 2000, pp. 447–450.

[24] M. Yannakakis, "A polynomial algorithm for the min-cut linear arrangement of trees," *J. ACM*, vol. 32, no. 4, pp. 950–988, Oct. 1985.

**Miloš Hrkić** received the B.S. and Ph.D. degrees in computer science from the University of Illinois, Chicago, in 2000 and 2004, respectively.

In 2001, 2002, and 2003, he was an Intern at IBM Austin Research Laboratory. In 2004, he joined IBM Corporation, East Fishkill, NY. His research interests include combinatorial optimizations, very large scale integration design automation, in particular routing, interconnect synthesis and buffering, and logic synthesis and resynthesis.

Dr. Hrkić received the University Fellowship in 2002 and the IBM Ph.D. Fellowship in 2003.

**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 Post-Doctoral Researcher at the University of California at Berkeley, supported in part by the NSF CISE program. In 1997, he joined the EECS Department, University of Illinois, Chicago, where he is currently an Associate Professor in computer science. His interests include design automation for very large scale integration (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.

**Giancarlo Beraudo** received the M.S. degree in electrical and computer engineering from the University of Illinois, Chicago, in 2002 and the "Laurea in Ingegneria Informatica" degree from Politecnico di Torino, Torino, Italy, in 2003.

In 2003 and 2004, he was with Telecom Italia Lab and Siemens. In 2004, he joined Bain & Company, Inc., Milan, Italy, where he is currently an Associate Consultant. His research interests include very large scale integration design automation, in particular physical design and timing optimization.