Optimizing Non-Monotonic Interconnect using Functional Simulation and Logic Restructuring

Stephen M. Plaza, Igor L. Markov, and Valeria M. Bertacco
The University of Michigan, Department of EECS
2260 Hayward Ave., Ann Arbor, MI 48109-2121
{splaza,imarkov,valeria}@umich.edu

***Abstract*— The relatively poor scaling of interconnect in modern digital circuits necessitates a number of design optimizations, which must typically be iterated several times to meet the specified performance objectives. Such iterations are often due to the difficulty of early delay estimation, especially before placement. Therefore, effective logic restructuring to reduce interconnect delay has been a major challenge in physical synthesis, a phase during which more accurate delay estimates can be finally gathered. In this work, we develop a new approach that enhances modern high-performance logic synthesis techniques with flexibility and accuracy in the physical domain. This approach is based on (1) a novel criterion based on path monotonicity, that identifies those interconnects amenable to optimization through logic restructuring and (2) a synthesis algorithm relying on logic simulation and placement information to identify placed subcircuits that hold promise for interconnect reduction. Experiments indicate that our techniques find optimization opportunities and improve interconnect delay by** 11.7% **on average at less than** 2% **wirelength and area overhead.**

I. Introduction

As interconnect contributes an increasingly significant fraction of overall circuit delay, the focus of design methodology is shifting from logic optimization to interconnect optimization. While this transition has been occurring for over a decade, meeting performance objectives is becoming more and more difficult. In recent years, a few successful methodologies achieved timing closure by combining netlist level minimization in logic synthesis with post-placement physical optimizations. This family of solutions is known as physical synthesis. Related strategies, including interconnect buffering [21], gate sizing [18], and relocation [1], successfully improved delay. In [8], [11], [32], [17], post-placement resynthesis achieved delay improvement with limited placement perturbation, but these techniques are limited to simple signal substitution transformations. As the major portion of the critical delay is shifting into interconnect [38], poor design choices during synthesis cannot be easily corrected by limited scale post-placement optimizations. Therefore, more accurate delay models have been developed to guide logic synthesis. *Wire-load* models that estimate delay by considering the capacitive load of each net were effective until wire capacitance and resistance became predominant. Further knowledge of the impact of placement on wirelength was consequently needed by synthesis algorithms.

To meet the challenge of performance optimization at the 130nm technology node and beyond, the traditional design flow transformed from several discrete optimization phases (such as logic synthesis followed by place-and-route) into a more holistic strategy. In [14] wirelength estimation was incorporated in logic synthesis by constructing a highly placeable netlist with the goal of reducing wire detours. In addition, topographical information has been used to guide current synthesis tools [40]. Due to the importance and inherent difficulty of estimating the impact of placement and routing on interconnect, researchers suggested the idea of maintaining a companion placement estimate throughout the logic synthesis process [10], [15], [26]. However, interconnect-aware logic transformations are still limited by the accuracy of the estimates available. Furthermore, guiding logic synthesis by conservative delay estimates, as in [14], can lead to transformations that do not improve critical path delay but increase area and power consumption.

While performing aggressive logic restructuring using global information is desired to exploit better estimates later in the design flow, such accounts have eluded published literature. One particular complication is that the limited amount of flexibility found in combinational circuits must be combined with physical aspects of performance optimization. In this paper, we introduce a post-placement solution that enables aggressive optimization while minimizing changes to the physical netlist. We consider a wide range of changes to the circuit structure while also tracking their impact on physical parameters of the circuit.

Our contributions are as follows:

1) A novel metric for efficiently identifying *non-monotonic* paths in the circuit, that locates regions where restructuring provides the greatest gains. This metric generalizes the metric in [4] and considers longer paths.
2) A generic and powerful technique for discovering logic transformations using functional simulation, which also facilitates fast re-evaluation of physical parameters. Our technique does not require local equivalence between the optimized subcircuit and the original, but uses simulation and satisfiability to ensure that the circuit's functionality is unmodified.
3) A suite of powerful algorithms that efficiently exploit a circuit's don't-cares and avoid heavy-weight techniques traditionally used in logic synthesis, while allowing tighter integration with placement and a more realistic delay calculation.

In our methodology, we first identify detoured wires that lie on critical paths. The example in Figure 1 shows a long critical path with several wire detours. As mentioned in [4], many critical paths cannot be improved through cell relocation and better timing-driven placement. Furthermore, the inaccuracy of timing estimates before detailed placement limits the effectiveness of techniques from [14] in eliminating path non-monotonicity. We target these non-monotonic paths for resynthesis by generating different logic topologies that

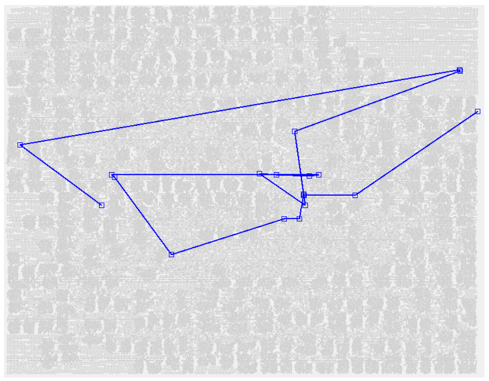


Fig. 1. An example from [8] of a non-monotone critical path after placement for circuit DES.

improve circuit delay. To efficiently find these topologies, we abstract away circuit complexity using logic simulation. Through logic simulation, we partially characterize the behavior of each node in the subcircuit with a *signature* [8], [7], [28]. We then use these signatures to determine whether a logic transformation generating the desired topology is possible.

In the example of Figure 2, we show that by applying our technique, a subcircuit with a long critical path can be transformed to a functionally-equivalent subcircuit with smaller critical path delay. Unlike most techniques from logic synthesis, our circuit restructuring can work directly on mapped circuits with complex standard cells. Another novel feature is our extensive use of circuit flexibility due to signal masking by downstream logic, also known as observability don't-cares (ODC). Additionally, our approach uses controllability don't-cares (CDC), *i.e.*, circuit flexibility due to upstream logic. Compared to work in [34], our approach exploits global don't-cares to enhance logic restructuring. In [17], redundancy addition and removal (RAR) are used to improve circuit timing. However, these rewiring techniques consider only a subset of our transformations, where we use redundancy and physical information in conjunction to directly guide the resynthesis of subcircuits containing multiple cells.

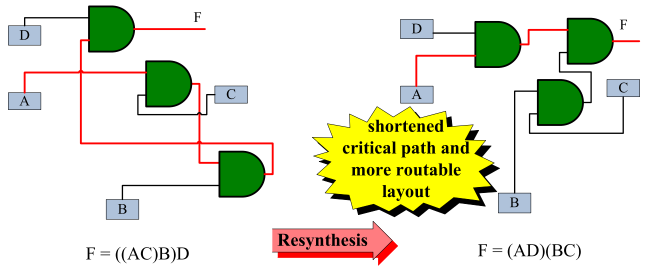


Fig. 2. The resynthesis of a non-monotone path can produce much shorter critical paths and improve routability.

Our experiments indicate that large circuits often contain many long critical paths that can be effectively targeted with our restructuring. Improving these paths results in consistent delay improvements of 11.7% on average with minimal degradation to other performance parameters. Furthermore, we achieve almost twice the delay improvement of that achieved by RAR-based timing optimizations. Our techniques are fast and scale to large designs, whereas completely characterizing node functionality with BDDs would require a prohibitive memory footprint.

In Section II, we review the use of simulation to guide logic optimization and summarize state-of-the-art synthesis strategies. In Section III, we introduce our interconnect optimization strategy. In Section IV, we propose a metric for finding circuit paths that require restructuring. Section V introduces a novel physically-aware synthesis approach that uses simulation. Empirical validation is presented in Section VII, and we summarize our work in Section VIII.

II. Background

This section describes how functional simulation can be used to characterize the behavior of internal nodes in the circuit and guide logic optimization. We then discuss a state-of-the-art approach for logic synthesis, currently limited to the logic domain that provides essential components for our physical synthesis algorithms.

A. Simulation and Satisfiability

A node F in a Boolean network can be viewed as a function of primary inputs. Such a node can be characterized by its *signature*, S_F, for K input vectors $X_1 \cdots X_K$.

Definition 1: $S_F = \{F(X_1), \ldots, F(X_K)\}$ where $F(X_i) = \{0, 1\}$ indicates the output of F for a given input vector.

A carefully-designed testbench or constrained-random simulator can be used to generate vectors X_i and derive signatures for each node in a circuit. For a network with n nodes, the time complexity of generating signatures for the whole network is $O(K * n)$. The functional non-equivalence of two nodes can be determined by the following: $S_F \neq S_G \Rightarrow F \neq G$.

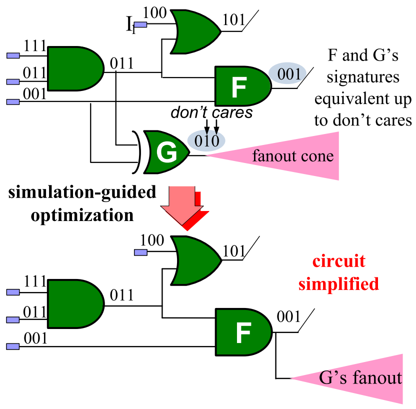


Fig. 3. Optimization by merging equivalent nodes in the presence of don't-cares. 3-bit signatures are shown at the output of each gate.

Signatures can be efficiently created and manipulated by taking advantage of bit-parallel operations. Therefore, equal signatures can be used to efficiently identify potential node equivalences in a circuit by deriving a hash index for each signature [19]. Since $S_F = S_G$ does not imply that $F = G$, this potential equivalence must be verified, *e.g.*, using a SAT solver, as explained below.

The signature is a *partial* characterization of a node's functionality. Furthermore, the signature encodes all of the node's CDCs under the input vectors applied. The signature's partial characterization enables fast and aggressive optimizations without requiring a fully specified truth table. However, unlike traditional, correct-by-construction optimizations, these speculative transformations must be validated by a formal proof mechanism. Hence, the efficiency of [19], [23] depends on the underlying engines which formally verify the equivalence of nodes with identical signatures.

Recent advances in SAT solvers, *e.g.*, learning, non-chronological backtracking, and watched literals [24], [30] have made SAT a more scalable alternative to BDDs for equivalence checking. The equivalence of two nodes, F and G, in a network can be determined by constructing an XOR-based miter [5] between them and asserting the output to 1 as shown in the following formula:

$$(F = G) \Leftrightarrow (\forall i \; F(X_i) \oplus G(X_i) \neq 1) \qquad (1)$$

where $\bigcup_i X_i$ is the set of all possible input vectors.

In [19], input vectors are generated dynamically from counter-examples returned by SAT checks proving $F \neq G$. The dynamic input vectors improve the quality of the signatures by limiting situations where $S_F = S_G$ despite $F \neq G$.

B. Logic Optimizations with Signatures

Simulation is an effective means for quickly identifying candidates for optimization. In [28], [37], signatures were used to additionally encode ODCs to enable circuit simplification and optimization by merging equivalent nodes. Consider the example in Figure 3 which shows a circuit where logic simulation produces the signatures shown. Notice that through efficient don't-care computation using a fast linear-time simulation [28] of downstream nodes, don't-care values can be determined for some of the signature's positions. In the example, these don't-cares suggest a potential circuit simplification by merging two nodes. The optimization will need to be verified by a formal proof engine.

Despite these advantages, signature-based optimizations are limited, and general synthesis algorithms have not been developed. A key contribution of this work is the application of signatures to enable logic restructuring while relying on available don't-care computation algorithms.

C. Logic Rewriting

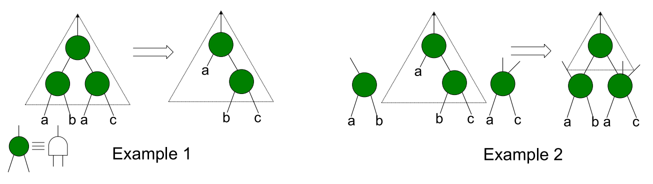


Fig. 4. Two examples of AIG rewriting. With structural hashing, it is possible in the second example to reuse external nodes and minimize the subgraph.

Performing scalable logic optimization requires efficient netlist manipulation, typically involving only a small set of gate primitives. Given a set of Boolean expressions that describe a circuit, the goal of synthesis optimization is to minimize the number of literals in the expressions along with the number of logic levels. Several drawbacks of these techniques are discussed in [22], including limited scalability. To this end, the authors of [22] introduced an efficient synthesis strategy called *rewriting*. Logic rewriting is performed over a netlist representation called an And-Inverter Graph (AIG) [19], where each node represents an AND gate and complemented (dotted) edges represent inverters. In logic rewriting, the quality of different functionally-equivalent implementations for a small logic block in a circuit is assessed. In Figure 4, the left transformation leads to a reduction in area. By using a technique called *structural hashing* [19], nodes in other parts of the circuit can be reused. In the right example, there is a global reduction in area by reusing available gates. However, structural hashing requires that the circuit be represented as an AIG and is not viable on mapped circuits.

The increasing significance of wire delay is addressed by providing more accurate delay models to logic synthesis, from using wire-load models to maintaining companion placements [10]. The delay model is used to modify the literal reduction objective so that transformations or rewrites that improve the delay according to the model are favored. However, delay estimation is becoming more inaccurate before detailed placement and routing as the actual interconnect routes become more important. This trend suggests that new synthesis algorithms should be applied after placement and routing because speculative optimizations can actually increase delay while negatively impacting other performance metrics like area.

III. Our Approach

In this paper, we introduce a new synthesis approach that accounts for physical aspects of performance optimization. We illustrate our approach in Figure 5. Starting from a fully placed circuit, we identify critical paths using static timing analysis. We then apply a novel metric introduced in Section IV that finds subcircuits for which restructuring could provide the greatest improvements. Next, we perform logic simulation using an even distribution of input vectors and generate signatures that encode don't-cares to obtain a partial characterization of the functional behavior of the circuit. Using this functional information encoded in signatures along with the physical constraints, we efficiently derive a topology that is logically equivalent to the original subcircuit but exhibits better performance. Finally, we legalize the altered placement and update the timing information in the circuit. As a result, we tailor our path-monotonicity metric to find portions of the critical path resulting in the greatest delay improvements. In addition, our techniques can target other objectives as well.

Using signatures for restructuring is advantageous because logic simulation provides a more scalable functional representation than BDDs. Furthermore, signatures can characterize internal nodes for netlists mapped to standard cells as well as for technology-independent netlists. In contrast, the logic rewriting strategy in [22] does not operate on technology-mapped circuits and does not take physical information into account. We also improve solution quality by considering more don't-cares while being directly guided by physical constraints.

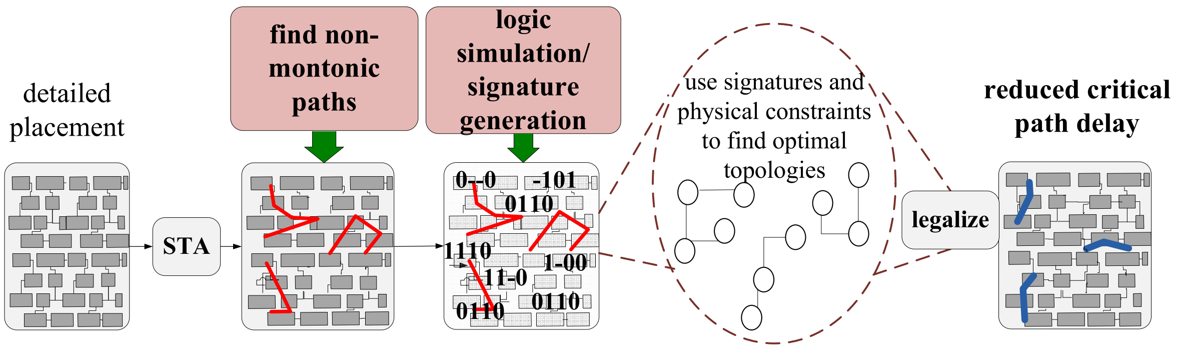

detailed placement
STA
find non-montonic paths
logic simulation/ signature generation
0--0
-101
0110
1110
1-00
11-0
0110
0110
use signatures and physical constraints to find optimal topologies
legalize
reduced critical path delay

Fig. 5. Our approach to optimizing interconnect. First, we identify non-monotonic critical path interconnect, and then we restructure these paths to improve delay. Such netlist transformations include gate cloning, but are substantially more general. They do not require that the subcircuits in question be equivalent. Instead, they use simulation and satisfiability to ensure that the entire circuit remains equivalent to the original.

IV. Identifying Non-monotone Paths

To maximize the effectiveness of our post-placement optimizations, we target timing critical parts of the design that are amenable to restructuring. In this section, we introduce our fast Dynamic Programming (DP) algorithm for finding paths in logic that are *non-monotone*, or paths that are not optimally short. Unlike the work in [4], we consider paths of arbitrary lengths and scale to many more segments in practice. We propose two models for computing path monotonicity: (1) wirelength-based and (2) delay-based. Non-monotonic paths indicate regions where interconnect and/or delay may be reduced by post-placement optimization.

A. Path Monotonicity

First, static timing analysis is performed to enable our delay-based monotonicity calculation and identify critical and near-critical paths. We use a timing analyzer whose interconnect delay calculation is based on Steiner-tree topologies produced by FLUTE [12][1] and the D2M delay metric [2] that is known to be more accurate than Elmore delay. Before focusing on critical paths, we will describe a general approach that examines the monotonicity of every path. We define the **non-monotone factor (NMF)** for the path $\{x_1, ..., x_k\}$ with respect to a given cost metric (such as wirelength or delay) as follows:

$$NMF = \frac{1}{c_{ideal}(x_1, x_k)} \sum_{n=1}^{k-1} c(x_n, x_{n+1}) \qquad (2)$$

where $c(a, b)$ defines the actual *cost* between a and b and c_{ideal} defines an optimal cost. When $NMF = 1$, the path is monotone under the cost metric. We explore two definitions for cost, one based on rectilinear distance and another on delay.

For the rectilinear case, $c(a, b)$ is the rectilinear distance between cell a and b while $c_{ideal}(a, b)$ is the optimal rectilinear distance assuming a monotonic path. For the delay-based definition, $c(a, b)$ is the $AT(b) - AT(a)$, where AT is arrival time. We define c_{ideal} as the delay of an optimally buffered path between a and b as described by [25] and given by the following formula:

$$c_{ideal}(a, b) = dist(a, b)(R_{buf}C + RC_{buf} + \sqrt{2R_{buf}C_{buf}RC}) \qquad (3)$$

[1]Timing-driven Steiner trees can be easily used instead [3].

where R and C are the wire resistance and capacitance respectively and R_{buf} and C_{buf} are the intrinsic resistance and input capacitance of the buffers. $dist(a, b)$ is the rectilinear distance between a and b. Unlike the distance calculation where the ideal path length between a and b can be equal to the actual path length, the optimal buffered wire between a and b has delay $\leq AT(b) - AT(a)$. We only attempt to optimize paths with large NMFs.

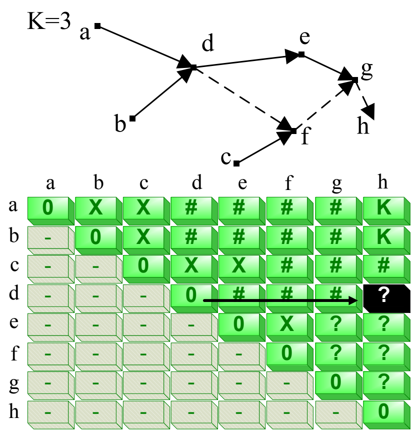
B. Calculating Non-monotone Factors

We now present our algorithm for calculating the NMF of all k-hop paths in a circuit, for a given $k \geq 2$. Our experiments indicate that the greatest NMFs are often observed on relatively short paths, and optimizing such paths brings greatest benefits.

```
inputs
  Nodes nodes: netlist
  Dist K: length of paths considered
output
  NMF: NMF between each node
void gen_NMF(Nodes nodes, Dist K) {
    levelize(nodes);
    for_each node1 in nodes
      for_each node2 in range(node1+1, node1+K)
        c_ideal_array[node1,node2] = c_ideal(node1, node2);
    for_each node1 in nodes
      subtot[] = 0;
      for_each node2 in range(node1_succ, node1_succ+K)
        subtot[node1,node2] = max(subtot[node1,node2_pred]
            + c(node2_pred, node2));
        factor = subtot / c_ideal_array[node1,node2];
        NMF[node1,node2] = factor;
}
```

Fig. 6. Generating non-monotone factors for a k-hop paths.

The non-monotone factor can be efficiently computed for every path using a $O(K * n)$-time algorithm for n nodes in the circuit, as shown in Figure 6. First, the circuit is levelized. Then, c_{ideal} is computed for node pairings with a connecting path of $\leq k$ hops, and the values are stored in `c_ideal_array`. All pairs are traversed again, and the `subtot` is generated by computing the maximum cost from *node1* to *node2* through a recurrence relation. The NMF is computed for the subpath, $\{node1,\ node2\}$, by dividing the



	a	b	c	d	e	f	g	h
a	0	X	X	#	#	#	#	K
b	-	0	X	#	#	#	#	K
c	-	-	0	X	X	#	#	#
d	-	-	-	0	#	#	#	?
e	-	-	-	-	0	X	?	?
f	-	-	-	-	-	0	?	?
g	-	-	-	-	-	-	0	?
h	-	-	-	-	-	-	-	0

Fig. 7. Calculating the non-monotone factor for path $\{d, h\}$. The matrix shows computations performed while executing the algorithm in Figure 6.

total cost, `subtot`, by `c_ideal[`$node1, node2$`]`. In Figure 7, we illustrate a subcircuit being traversed using the `gen_NMF` function where $k = 3$ and the current $node1$ is d. The matrix indicates the NMFs already computed with #, and nodes not lying on the same path with X. Because we traverse the graph in levelized order, a, b, c have already been examined. Notice, that nodes that are farther than k hops away are not examined (indicated by K in the matrix). For node d, the non-monotone factor is computed for path d, h by determining all the incoming sub-paths to h first.

V. Physically-aware Logic Restructuring

We optimize the subcircuits that are identified by the path monotonicity metric as illustrated in Figure 8. We first select a region of logic determined by the non-monotonic path for resynthesis. We then use signatures to find an alternative implementation with a topology that improves physical parameters and that is logically equivalent to the original implementation (up to the signatures). This implementation is then formally verified by performing SAT-based equivalence checking between the original and new netlists.

Previous work on improving path monotonicity used logic replication [16]. However, the technique is restricted to the topology of the extracted subcircuit, and its optimization are independent of the subcircuit's functionality. Furthermore, as observed in [16], gate relocation sometimes cannot improve path monotonicity. In the following, we introduce the theoretical framework to resynthesize a subcircuit given a set of inputs and a target output by introducing a concept called *logical feasibility*. We then introduce an algorithm for constructing subcircuits using signatures and physical constraints to optimize the interconnect.

A. Determining Logical Feasibility with Signatures

We introduce a goal-driven synthesis strategy that efficiently finds a logic implementation for a given physical topology. The major thrust of previous efforts in post-placement logic optimization involves the efficient encoding of logic functionality and, in particular, circuit don't-cares. In [20], the authors proposed a technique to enumerate through the decompositions of a particular node using BDDs. By encoding the ways of decomposing a node with BDDs, the authors provide an algorithm for resynthesizing logic that can work on mapped netlists using different standard-cell libraries. No strategy is considered for exploiting global circuit don't-cares which could be used to enhance the quality of the decompositions considered. In [35], *sets of pairs of functions to be distinguished* (SPFDs) are introduced as a way of representing a node's functionality which can be used to exploit circuit flexibility in logic optimization. In [31], the authors propose a technique that uses SPFDs to find a logic implementation given a topological constraint, but their resynthesis approach does not incorporate physical parameters such as timing and is limited to only a few neighboring levels of logic to reduce the memory and computational requirements of SPFDs. In an alternative strategy to reduce the memory requirements of SPFDs, the authors in [36] choose a subset of SPFDs for a node using simulation and compatibility don't-cares in a logic rewriting application.

In our work, we use logic signatures to expose circuit functionality. Our approach is advantageous because the data structures involved in our technique do not need to represent an exponential amount of information. This is generally the case regardless of the underlying functionality. Through logic simulation we can encode global circuit don't-cares which are not limited by levels of logic or required to be compatible. Furthermore, our approach encodes the distinguishing bits in a compact data structure with logic signatures so that these operations can be performed with bitwise parallelism. This is particularly beneficial in our development of a novel goal-driven synthesis technique where fast evaluation of topological constraints while exploiting don't-cares is essential to tightly couple physical optimization and logic synthesis.

Given an extracted subcircuit with χ inputs, $\{a_1, a_2, ..., a_\chi\}$ and output F to resynthesize, we express a candidate restructuring as a directed graph T_F with χ incoming edges, one outgoing edge F, and n internal vertices. We would like to determine whether there is a labeling, G^*, of n vertices with gates $g \in G$ such that F is logically equivalent to the subcircuit T_{F^c} that implements T_F with respect to the outputs of the circuit. We define the **logical feasibility** of T_F as:

Definition 2: T_F is logically feasible iff

$$\exists_{G^*} onset(T_{F^c}) = onset(F)$$

where $onset$ represents where the subcircuit produces 1 for an input combination. This definition can be relaxed by considering its relation within the care-set which could be considerably smaller than 2^χ due to circuit don't-cares.

Definition 3: T_F is logically feasible up to circuit don't-cares iff $\exists_{G^*} onset(T_{F^c}) \cup dc(F) = onset(F) \cup dc(F)$ where dc is the don't-care set.

A naive algorithm for determining the logical feasibility of T_F requires that every possible labeling G^* is tried. For n vertices, this requires checking $|G|^n$ mappings. If the set of 2-input logic functions is considered, there are 5^n mappings.[2] Furthermore, performing equivalence checking between T_{F^c} and F is an NP-complete problem. Below, we discuss how

[2]Although there are 16 different two-input Boolean functions, the tautology and two one variable transfer functions along with the negated form of each function do not need to be explicitly considered

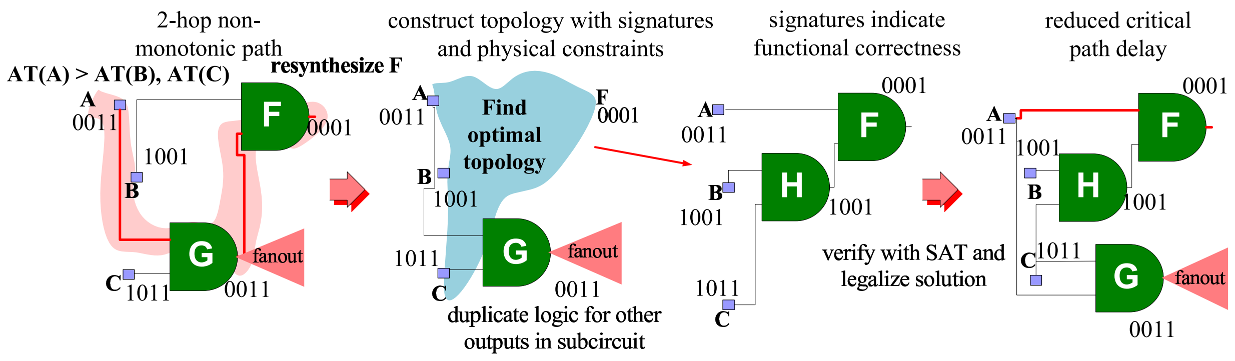

2-hop non-monotonic path
AT(A) > AT(B), AT(C)
resynthesize F
A
0011
1001
B
F
0001
G
fanout
C
1011
0011
construct topology with signatures and physical constraints
Find optimal topology
duplicate logic for other outputs in subcircuit
signatures indicate functional correctness
H
verify with SAT and legalize solution
reduced critical path delay

Fig. 8. Our flow for restructuring non-monotonic interconnect. We extract a subcircuit determined by our non-monotonic metric and find topologies that are logically equivalent using simulation. This new implementation is then verified by equivalence checking with an incremental SAT solver.

signatures can be used to determine a set of inputs that implements a given function and how to quickly determine logical feasibility up to the signature approximation.

Pairs of bits to be distinguished: A function F is said to be dependent on an input a_i if and only if:

$$F_{a_i=0} \oplus F_{a_i=1} \neq 0 \tag{4}$$

A similar relationship between a signature S_f and input signatures $S_1, ..., S_\chi$ can be established. In [7], it was observed that a set of input signatures can implement a target signature, if and only if, every pair of different bits in S_f is *distinguished* by at least one S_χ.

Definition 4: A pair of bits to be distinguished (PBD) is a pair $\{i, j\}$ such that $S_f(i) \neq S_f(j)$.

Definition 5: A candidate signature, S_χ distinguishes a PBD in S_f if $S_x(i) \neq S_x(j)$ where $\{i, j\} \in S_f^{PBD}$ where S_f^{PBD} is f's set of PBDs.

Example 1. Assume a target signal $S_f = \{0, 0, 1, 1\}$ and candidates $S_1 = \{0, 0, 0, 1\}$, $S_2 = \{0, 1, 0, 1\}$, and $S_3 = \{0, 1, 1, 1\}$. The PBDs of S_f are $\{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}$ that need to be distinguished. Note that S_1 and S_2 together cannot implement S_f because they do not distinguish $\{0, 2\}$. However, if all S_χ are used, there exists a function that gives S_f. In this example $S_f = S_3 \cdot (S_1 \bar{\oplus} S_2)$. □

Essential PBDs: Input signtures form an irredundant cover of S_f's PBDs when 1) every PBD is covered by at least one S_i and 2) removing one S_i results in at least one uncovered PBD. The resulting S_is form the support of the function to be resynthesized. We define a PBD that is distinguished by only one S_i as an *essential* PBD for S_i. According to the definition of an irredundant cover and PBDs, each S_i must have at least one essential PBD (or else that input can be discarded). Because there is at least one essential PBD for each input, S_f is dependent on S_i independent of the specific implementation, if the following condition holds:

$$S_{f(S_i=0)} \oplus S_{f(S_i=1)} = 1 \tag{5}$$

In the case of the resynthesis of a function $F(a_1, ..., a_\chi)$, we note that the cardinality of the irredundant cover can be less than χ because F may be independent of an a_i up to don't-cares and the signature abstraction might not expose enough essential PBDs. Furthermore, several irredundant covers are possible. In this paper, we greedily determine irredundant covers by first selecting signatures that cover the most PBDs and continuing until all PBDs are covered.

Determining logical feasibility with essential PBDs: We now describe how the logical feasibility of a given topology can be determined using signatures. Later, we explain how to create such topologies and how to verify the signature-based abstraction. Our strategy considers the set of available gates G as implementing all the 2-input logic functions, so that each node n has exactly two input edges. In general, we do not restrict our topologies to be *fanout-free* trees, where a topology is fanout-free if each node n in T_F has only one outgoing edge.

However, fanout-free topologies (where we make the additional constraint that each primary input has only one outgoing edge) form a critical aspect of our goal-driven synthesis strategy because, under a couple of assumptions, they produce circuits with optimal area and timing if such a fanout-free circuit exists. First, we assume that the area associated with each node/gate in the topology is equal (since the implementation of the topology is unknown). Second, the delay through the subcircuit is determined by its path length through the topology, where we assume that each wire corresponding to an edge is optimally buffered. Therefore, fanout-free topologies have smaller area than their non-fanout free counterparts when implementing a single-output function because they have fewer internal nodes ($\chi - 1$ nodes). Furthermore, fanout-free topologies have the same or smaller delay as non-fanout free trees. The proof of this is straightforward because if a reconvergent topology has optimal delay based on path length, converting this topology to a fanout-free tree by removing edges and nodes will not increase path length.

In the next few paragraphs, we introduce an algorithm for determining logical feasibility on fanout-free circuits where each primary input has only one outgoing edge F, which can be performed with a $O(|S_F^{PBD}| * \chi)$-time algorithm using signatures. Because logical feasibility is not always possible for a fanout-free tree that optimizes a particular performance criterion, we extend our synthesis techniques to handle arbitrary non-tree topologies.

First, we associate a signature to each input χ of T_F. These signatures implicitly handle controllability don't-cares as impossible input combinations which will never occur in the signatures. By simulating downstream nodes as in [28], observability don't-cares are derived and S_f is reduced to include only care values. If we assume that each S_i under simulation distinguishes at least one essential PBD, we note the following for each 2-input gate in a fanout-free topology:

Theorem 1: For input signatures S_1 and S_2 and the 2-input function, Φ, the signature $S_{1,2} = \Phi(S_1, S_2)$ has S_1 and S_2 essential PBDs.

Proof. Any cut through T_F gives a set of inputs that implements F. Therefore, the S_F^{PBD} must be distinguished by each cut in T_F^c for a feasible topology. Since in a fanout-free topology, S_1 and S_2 do not reoccur in the topology, the output of the node combining S_1 and S_2, S_{12}, must contain their essential PBDs to distinguish S_F. □

As a direct consequence, each 2-input transformation preserves at least two essential PBDs. Furthermore, PBDs that only occur in both S_1 and S_2 must also be preserved to uphold the invariant that every cut through the topology forms an input support. In a similar manner, the work in [31] upholds this invariant in constructing a subcircuit but considers SPFDs instead. We note the following:

Theorem 2: There are at most two 2-input Boolean functions (ignoring negated version of these functions) that can preserve all the essential PBDs of the input signatures.

Proof. A 2-input Boolean function has 4 row truth table with output 0 or 1. One essential PBD adds the following constraint:

$$[\Phi(a,b) = z] \wedge [\Phi(a',b) = z'] \tag{6}$$

where a, b, and z are variables with value 0 or 1. In other words, two different rows of the truth table must have different values. For a given a and b where an essential PBD is defined, there are only 2 such assignments to z that satisfy this constraint. The remaining 2 rows in the truth table can have any of 4 possible output combinations. Therefore, there is a total of 8 different functions that satisfy this constraint. We ignore negated versions of the Boolean function since that negation can be propagated to the inputs of later gates. Given this, there are 4 distinct functions that can preserve one essential PBD. However, since two essential PBDs must be preserved, the following constraint needs to be satisfied:

$$[\Phi(a,b) = z] \wedge [\Phi(a',b) = z'] \wedge [\Phi(d,e) = y] \wedge [\Phi(d,e') = y'] \tag{7}$$

If $\{(a,b),(a',b)\}$ is disjoint from $\{(d,e),(d,e')\}$, there are only 4 possible output combinations of z and y that satisfy the constraints, where 2 of them are the negated form. This is also the case if $\{(a,b),(a',b)\}$ is not disjoint from $\{(d,e),(d,e')\}$ (it is impossible for two different functions to have essential PBDs on the same two rows). Therefore, there are at most only 2 distinct Boolean functions that can preserve the essential PBDs of its inputs. □

If the fanout-free tree is traversed in topological order, a choice between two different 2-input gates is available for each node. In the worst case, all possible combinations must be tried to preserve all the essential PBDs giving $O(|S_F^{PBD}|2^{\chi})$-time complexity (there are $\chi - 1$ nodes). For the typically small topologies that are considered for resynthesizing portions of the critical paths, this results in significant practical runtime improvement over trying all possible gate combinations without considering PBDs. However, we note that in many cases the runtime complexity is linear.

Theorem 3: The logical feasibility of an χ-input fanout-free T_F can be determined in $O(S_F^{PBD} * \chi)$ time when K simulation vectors completely specify the functionality of F.

Proof. A fanout-free topology specifies a disjoint partition of the inputs. If an implementation exists with a disjoint partitioning of inputs, each internal node corresponds to a function that is specified independently of the rest of the implementation. Therefore, when the signatures completely specify F (a complete truth table), each internal node is also completely specified. Because of this, each two-input operation must preserve at least 3 essential PBDs (the minimal number of distinguishing bits a 2-input function can have) and therefore only one function satisfies this relation. Because there is only one such candidate function, the complexity of finding an implementation is $O(S_F^{PBD} * \chi)$. □

Although we often resynthesize functions with small supports and therefore small truth tables, a logic signature does not always completely specify a function's behavior resulting in a reduction in the number of bits that need to be distinguished. Also, the ability of simulation to quickly identify circuit don't-cares further reduces the number of bits that need to be distinguished. By not having a completely specified function, we facilitate multiple feasible implementations. Despite the advantages of this flexibility in determining a feasible implementation, an internal 2-input operation may only need to preserve 2 essential PBDs rather than 3, which can increase the runtime of finding an implementation. However, in practice, this runtime penalty is minor because the topologies are typically small. Also, in many cases logical feasibility can still be determined in $O(S_F^{PBD} * \chi)$ time depending on which bits need to be distinguished.

Although in this paper we use a functionally complete set of 2-input gates, our approach extends to other standard-cell libraries. We now explain how to accomodate larger cells. First, we allow topologies where each node can have more than two incoming edges. Then, each node with more than two incoming edges is decomposed into nodes that represent 2-input gates. Finally, this implementation is mapped to a set of library cells using structural matching.

In some cases, a topology optimizing a certain performance objective may be logically infeasible. Furthermore, some functions, *e.g.*, $z = a'b + ac$ (a multiplexor), cannot be implemented using a fanout-free topology. Therefore, a viable technique must handle a broader family of topologies. In the case of the multiplexor, notice that only signal a has fanout, while b and c only occur once. We now describe how essential PBDs can be used to guide synthesis for non-tree topologies where each operation preserves at least one of its inputs' essential PBDs. This facilitates reconvergence and the implementation of useful functions including multiplexors, as shown below.

Theorem 4: The logical feasibility of an n-node topology T_F can be determined in $O(|S_F|^{PBD} * 3^{\chi})$ time for K simulation vectors under the following conditions:

1) At least one input to each node does not fanout to another node at the same or greater logic level

2) Only implementations are considered where the signatures along each cut through the topology form an irredundant cover.[3]

The logic level of a node is determined by the path from the node to the primary inputs with the greatest number of edges. **Proof.** By traversing the graph in topological order, note that at least one essential PBD is transferred to the output. Also, when those implementations are considered where the signatures along each cut of the topology form an irredundant cover, each signature along the cut has at least one essential PBD. The constraints in Equation 6 suggest that there are 4 distinct 2-input functions that preserve one essential PBD. However, one of these functions will correspond to the 1 input identity function, *i.e.*, a buffer (or inverter in the negated case). Ignoring this case, there are 3 distinct functions can be tried at each node, which requires no more than 3^n total gate combinations to determine logic feasibility. □

Handling arbitrary topologies with no implementation constraints requires more computation where 5^n gate combinations are examined. However, in practice, our approach is faster than the naive enumeration described at the beginning of the section because the operations are performed on the signatures, not over the whole truth table. Also, essential PBDs can still significantly prune the search space. Each cut still must cover all of the PBDs. If an edge from internal node or primary input does not appear past a certain logic level in the topology, its signature's essential PBDs must be preserved across that level.

B. Sub-circuit Extraction

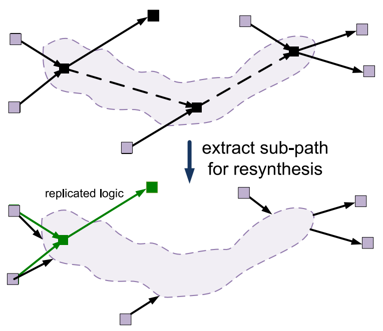


Fig. 9. Extracting a subcircuit for resynthesis from a non-monotonic path.

After identifying the most non-monotonic path, we extract a subcircuit as shown in Figure 9, where the inputs of the subcircuit are the incoming edges to the path and the output has outgoing edges from the end of the path. The inputs and fanout of the subcircuit are treated as fixed cells, which form the physical constraints. As shown in the figure, if there are outgoing edges at intermediate nodes in the path, this logic is duplicated. In practice, we experience minimal cell area increase because few cells are duplicated, and the resynthesized circuit is sometimes smaller than the original.

[3]In general, a topology may have an implementation with redundant covers. However, we focus on implementations that do not use this redundancy to improve the efficiency of our approach.

C. Physically-guided Topology Construction

In addition to efficiently *determining* the logical feasibility of various topologies, we propose an algorithm that uses PBDs and physical constraints to efficiently *construct* logically feasible topologies. In this paper, we guide our approach using delay and physical proximity. In the example shown in Figure 10, we try to find an optimal restructuring to implement the target function F with the inputs a, b, and c, using signatures. The functionality of the original circuit is represented by signatures, and a table is associated with each signal showing the PBDs that are distinguished. The non-essential PBDs for each input signature have light-gray background.

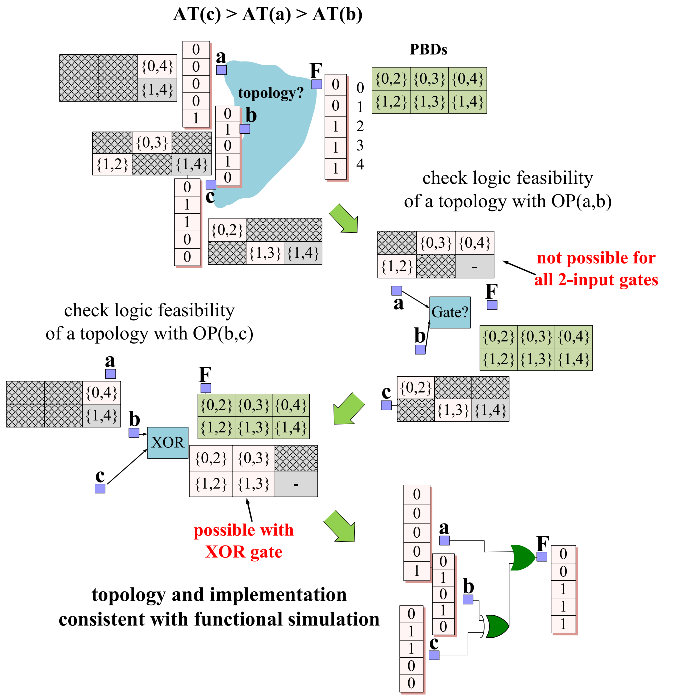


Fig. 10. Signatures and topology constraints guide logic restructuring to improve critical path delay. The figure shows the signatures for the inputs of the topology to be derived along with the output. Each table represents the PBDs of the output F that are distinguished. The topology that applies a and b is infeasible because it does not preserve essential PBDs of a and b. A feasible topology uses b and c, followed by a.

The example shows that the arrival time for c is the greatest, followed by a, then b. Therefore, we first consider a topology where c's value is required later. We also consider the proximity of the signals and therefore examine a topology where an operation between a and b is performed. Notice that if all possible 2-input operations are tried, the essential PBDs are not preserved and hence this is not a feasible topology. We then consider another topology where a can be consumed later because no topology exists where c is consumed last. For this topology, we see that an XOR-gate will preserve the essential PBDs. We then easily determine that an OR gate is needed to implement F.

Algorithm: We introduce the pseudo-code of our algorithm for restructuring non-monotonic interconnect in Figure 11. After identifying the non-monotonic paths, `Optimize_Interconnect` restructures a portion of the critical path. We first simplify the signatures by simplify_signatures by noting that the size of the signature $|S_F|$ can be reduced to the number of different input combinations that occur across $\{S_1, ...S_i\}$. Thus, only a subset of the signature is needed for restructuring because the small subcircuits considered have a maximum of 2^i possible different input combinations, smaller than the number of simulation vectors applied.[4]

```
void Optimize_circuit(){
    gen_NMF();
    num_tries = X;
    while(worst_nmf > 1)
      if(nckt == Optimize_Interconnect(worst_nmf))
          if(!check_equiv(nckt))
              refine_signatures();
              continue;
          update_netlist();
          legalize_placement();
          update_NMF();
}
Subckt* Optimize_Interconnect(Subckt F){
    simplify_signatures(F);
    Constraints constrs;
    while(find_opt_topology(constrs))
        if(nckt == check_logical_feasibility())
            (*nckt).opt_place();
            return nckt;
        constrs.add(nckt);
}
```

Fig. 11. Restructuring non-monotonic interconnect.

We then add any timing or physical constraints, such as locations of the inputs and outputs of the subcircuit being restructured. In find_opt_topology, we find a topology that satisfies all the physical constraints and optimizes delay. The topology is created by a greedy algorithm which derives a fanout-free topology from the current input wires. We examine each pair of wires, apply an arbitrary cell, and estimate the delay to the output of the subcircuit. The topology is then greedily constructed so that wire pairs that produce earlier arrival times are consumed farther from the output of the topology. We will later discuss how to construct arbitrary non-tree topologies. From this topology, we can get an upper-bound for the best implementation possible that contains the examined combination. If a topology can't be found that satisfies the constraints, the function returns.

We then check the logical feasibility using PBDs and signatures in check_logical_feasibility. If the topology is feasible, we associate the appropriate gate with each vertex and place the subcircuit. Our placement routine considers only the legality of the subcircuit (we call a placement legalizer later for the entire design). In our approach, we determine a location for each gate by placing it at the center of gravity of its inputs and outputs and then sifting the gate to different nearby locations. This sifting is done over all the gates over several passes until a locally optimal solution is achieved, which results in no overlaps. For the typically small subcircuits considered, this requires little computational effort.

If the topology is not logically feasible, we add a *functional* constraint that will prevent the construction of similar topologies. The constraint states which wire pairs should not be combined again. For instance, for the multiplexor, $z = a'b+ac$, there is no implementation for a fanout-free topology with inputs $\{a, b, c\}$. If a and b form a wire pair, we see that no implementation preserves its essential PBDs. However, we can exploit Theorem 4 and consider implementations that can eliminate one of the inputs. In this case, if the implementation $a'b$ is tried, the wire b does not need to reappear in the topology. Therefore, a constraint is added so that the inputs to the topology are now $\{a'b, a, c\}$. With these inputs, a fanout-free tree does exist which is logically feasible.

If Optimize_Interconnect returns a subcircuit, we check the equivalence of the entire circuit using a SAT engine. In the case where our candidate produces a functionally different circuit (which is rare as shown in Section VII), we use the counter-example generated by SAT to refine our simulation hence improving the signatures' quality. If the resulting subcircuit passes verification, we update the netlist and legalize the placement. We update the timing information and the NMFs if a new critical path is found, in which case we select with the next highest NMF and restructure it.

D. *Efficient Subcircuit Verification*

Because we use signatures to limit verification of optimization candidates that are most likely correct, equivalence checking typically confirms the transformation. As in [9], we refine simulation using counterexamples found by failed equivalence checks, so as to reduces additional failed checks. We also minimize the verification time due to equivalence checking by considering only the portions of logic that contributes to the don't-cares used in the transformation. As explained in [28], several don't-cares can exist within a few levels of logic. We invoke a SAT engine so that it considers only these necessary levels of downstream logic. Additionally, we could restrict the equivalence checking to a window around the optimization location to further reduce verification time while still utilizing CDCs and ODCs in the circuit.

However, in practice, we observe that the SAT-based equivalence checking requires a small percentage of runtime compared to constructing optimal topologies even for our larger circuit examples. This small runtime can be attributed to the locality of most of our structural transformations. Because the structure of the original and modified circuits are similar, the SAT instance can be greatly reduced in size and complexity. This limits the complexity of our approach, which tends not to grow with the size of the overall circuit.

VI. Enhancing Resynthesis through Global Signature Matching

Our resynthesis strategy considers the inputs to a non-monotonic path for resynthesis. This strategy is convenient because 1) the set of inputs can always implement the target output and 2) the inputs tend to be physically close to the target output. However, local manipulations can be enhanced by incorporating global information, as in logic rewriting which uses structural hashing [22]. In this section, we explain how

[4]In our experiments, we apply 2048 input vectors and restructure subcircuits with < 10 inputs.

to exploit the same advantages as structural hashing, by performing matching up to the signature abstraction. Furthermore, our approach is more powerful than logic rewriting because the signatures are matched up to global don't-cares, and our initial physically-guided local rewriting over signatures already exploits don't-cares. We observe that our enhancement is no worse than the algorithm from [27] but appears more robust and predictable.

Algorithm: We now outline how signature matching is used in the resynthesis of non-monotonic paths:

1) Find a set of candidate wires within a certain distance of the output wire to be resynthesized.
2) Check whether any of these wires' signatures is equal to the output signature up to don't-cares. If a match is found and the timing improves, replace the output wire with the corresponding candidate wire.
3) While checking logic feasibility in topological order, check whether any of the internal wires of the topology can be reimplemented by a candidate wire with a matching signature so that the timing is improved.

The candidate wires are chosen by proximity to the output wire being resynthesized as determined by its HPWL. Any wire that has arrival time after the current output wire's arrival time is not considered. Unlike the resynthesis algorithm that uses a simplified signature, for signature matching, we consider the whole signature except for the don't-cares. In this case, a single comparison between signatures can be performed quickly and is more efficient than finding a common set of inputs to both wires and then reducing the signatures to the number of simulated different input combinations. Notice our algorithm enhances the previous resynthesis strategy and improves the timing of an *implementation*, whereas topology construction only considers the inputs to the subcircuit.

VII. Experiments

We implemented and tested our algorithms with circuits from the IWLS 2005 benchmark suite [39], with design utilization set to 70% to match recent practices in the industry. Our wire and gate characterizations are based on a 0.18μm technology library. We perform static timing analysis using the D2M delay metric [2] on Rectilinear Steiner Minimal Trees (RSMTs) produced by FLUTE [12]; here FLUTE can be easily replaced by a timing-driven subroutine, but we do not expect the overall trends in our experiments to change significantly. Our netlist transformations are verified using a modified version of MiniSAT [13] and placed using Capo 10 [6]. We have considered several different initial placements for each circuit by varying a random seed in Capo and report results as averages over these placements. Our netlist transformations are legalized using the legalizer provided by GSRC Bookshelf [41].

Our delay improvements are achieved by executing the algorithm in Figure 11. We applied 2048 random simulation patterns initially to generate the signatures. We considered paths of less than or equal to 4 hops (5 nodes) using our delay-based metric which allowed us to find many non-monotonic paths while minimizing the size of the transformations considered. We conducted several optimization passes until no more gains were achieved.

A. Prevalence of Non-monotonic Interconnect

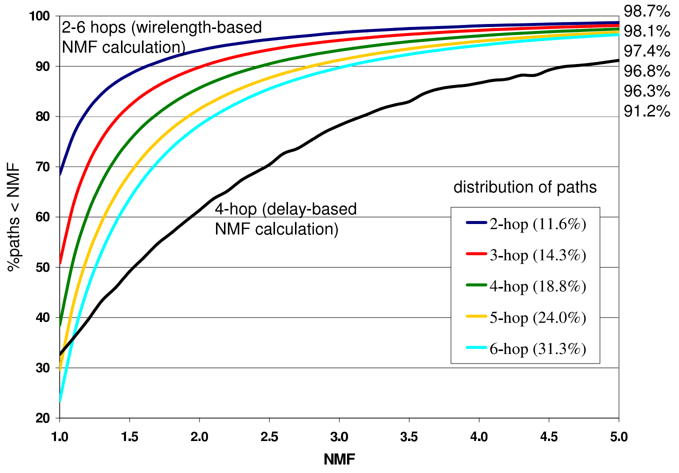


Fig. 12. The above graph shows the percentage of paths whose NMF is below a given value on the x-axis. Notice that longer paths tend to be non-monotonic and at least 1% of paths are > 5 times the ideal minimal length.

Our experiments indicate that circuits often contain many non-monotonic paths. In Figure 12, we illustrate a cumulative distribution of the percentage of paths whose NMFs is below the corresponding value on the x-axis. We generated these averages over all the circuits in Table I. Each line represents a different path-length examined, where we considered paths from 2 hops to 6 using the wirelength-based NMF metric. We also show the cumulative distribution for the 4-hop delay-based NMF calculation used to guide our delay-based restructuring. Of particular interest is the percentage of monotonic paths, *i.e.*, paths with NMF = 1.

Notice that smaller paths of 2-hops are mostly monotone, whereas the percentage of monotone paths decreases to 23% when paths with more logic levels are considered (6-hop paths). This indicates that focusing optimizations on small paths only, as in [4], can miss several optimization opportunities. It is also interesting to note that there are paths with considerably worse monotonicity having NMFs > 5, indicating regions where interconnect optimizations are needed. We observe similar trends using our delay-based metric. The inclusion of gate delay on these paths results in greater non-monotonicity when compared to the wirelength-metric. Although not shown, each individual circuit exhibits similar trends.

B. Physically-aware Restructuring

We show the effectiveness of our delay-based optimization by reporting the delay improvements achieved over several circuits. In Table I, we provide the number of cells and nets for each benchmark. In the `Performance` columns, we give the percentage delay improvement, the runtime in seconds, and the percentage of equivalence-checking calls where candidate subcircuits preserved the functionality of the whole circuit.

Circuit	Cell count	Net count	Performance			Overhead	
			%delay impr	time (s)	%equi	%wire	%cells
sasc	563	568	14.1	41	100	2.35	3.13
spi	3227	3277	10.9	949	82	4.53	0.73
des_area	4881	5122	12.3	503	93	1.09	0.31
tv80	7161	7179	9.1	1075	71	2.50	0.17
s35932	7273	7599	27.5	476	100	2.14	0.19
systemcaes	7959	8220	13.9	748	95	0.89	-0.07
s38417	8278	8309	11.7	481	84	0.68	-0.21
mem_ctrl	11440	11560	9.2	678	37	0.05	-0.02
ac97	11855	11948	6.3	245	100	0.44	0.02
usb	12808	12968	12.2	605	80	0.30	0.06
DMA	19118	19809	14.5	845	65	0.16	0.08
aes	20795	21055	6.4	603	100	0.13	0.01
ethernet	46771	46891	3.7	142	100	0.08	0.06
average			11.7%		85.1%	1.20%	0.34%

TABLE I

SIGNIFICANT DELAY IMPROVEMENT IS TYPICALLY ACCOMPANIED BY A SMALL WIRELENGTH INCREASE.

We also report the overhead of our techniques in terms of increased wirelength and area (cell count).

Considering 8 independently generated initial placements for each circuit, our techniques improve delay by 11.7% on average. For some circuits, such as $s35932$, several don't-care enhanced optimizations enabled even greater delay improvements. We observe the following:

1) By optimizing only one output of a given subcircuit, we greatly reduce the arrival time of that output, while only slightly degrading performance of less critical outputs.
2) Through our efficient use of don't-cares, several χ-input subcircuits could be restructured to require fewer than χ inputs.
3) As a special case of the previous point, sometimes an input to the subcircuit was functionally equivalent to the output of the subcircuit when don't-cares were considered, enabling delay reduction along with removal of unnecessary logic. Signatures are efficient in exploiting these opportunities.
4) The decomposition of large gates into smaller gate primitives through our restructuring algorithm often produces better topologies because we more precisely construct a topology to meet the physical constraints.
5) We also expect gains due to the duplication and relocation of some cells.

We believe that further gains would be enabled by combining buffering, relocation, and gate sizing strategies between our restructuring optimizations. The runtime of our algorithm scales well for large circuits due to the use of logic simulation as the main optimization engine. Furthermore, the high percentage of equivalence checking calls that verified the equivalence of our transformations (as shown by column `%equi` in Table I) indicates that signatures are effective at finding functionally equivalent candidates. The wirelength and cell-count overhead are minimal because only a few restructurings are needed and the optimizations can simplify portions of logic. In some cases the number of cells is reduced.

To check if our techniques provide comparable improvement when the initial placement is optimized for timing, we performed the following experiment. We first produced 64 independent initial placements optimized for total wirelength. Compared to these 64 wirelength-optimized placements, the best placements achieve 17.0% smaller delay on average and serve as proxies for timing-optimized placements in our experiments. Starting with the best placements, our logic restructuring further decreased delay by 6.5%.

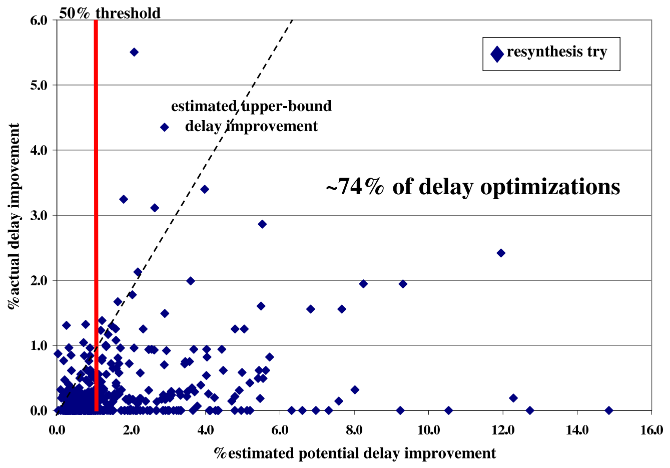


Fig. 13. The graph above illustrates that the largest *actual* delay improvements occur at portions of the critical path with the largest *estimated* gain using our metric. The data points are accumulated gains achieved by 400 different resynthesis attempts when optimizing the circuits in Table I.

Circuit	%delay	
	Ours	RAR
sasc	13.8	12.1
spi	15.0	12.6
des_area	15.4	11.1
tv80	12.7	3.1
s35932	23.1	21.8
systemcaes	10.1	4.0
s38417	26.3	2.9
mem_ctrl	12.9	8.2
ac97	5.3	3.1
usb	10.8	0.0
DMA	10.7	0.0
aes	5.3	4.7
average	13.5%	7.0%

TABLE II

EFFECTIVENESS OF OUR APPROACH COMPARED TO RAR.

In Figure 13, we demonstrate that our delay-based NMF metric is effective at guiding optimization. Each data point represents a different resynthesis try considering all of the circuits in Table I. The x-axis shows the predicted percentage delay gain possible (determined by the optimal-buffered delay). The y-axis indicates the actual gain. Data points that lie on the x-axis indicate resynthesis tries that did not improve delay (a better topology could not be found). The 50% threshold line divides the graph so that the number of resynthesis attempts are equal on both sides. The diagonal line indicates an upper-bound prediction for delay gain. Because some of the optimizations reduce the support of the original subcircuit, we can improve the delay beyond the estimate which considers all of the subcircuit's inputs. Therefore, some of the data points are above the upper-bound line. On the other hand, a resynthesis try produces a smaller than estimated improvement when the ideal topology is not logically feasible or when removing cell overlap degrades the quality of the initial placement. Although the NMF and gain calculations do not directly incorporate circuit functionality, 74% of all delay gains are found on the right half of the graph. The correlation to our metric could be further improved by incorporating the percentage of gain possible with respect to near-critical paths.

C. Comparison with Redundancy Addition and Removal

We compare our technique with timing optimizations using RAR. We implement redundancy removal using signatures [28] to identify equivalent nodes up to don't-cares. In the context of path-based resynthesis, the inputs to the subcircuit, along with signals that have earlier arrival time and are within a bounding box determined by the HPWL of the output, are considered as candidates for rewiring. If one of these signals is equivalent to the output up to don't-cares in the circuit, rewiring is performed and the timing improved.

In Table II, we show the delay improvement of our resynthesis strategy which uses global signature matching to redundancy addition and removal. For this experiment, we report results on a random slice of initial placements from our suite. First, note that our technique is almost twice as effective at improving delay. Furthermore, our results are more consistent than RAR over all the circuits and are never worse.

VIII. Conclusions

Interconnect delay is becoming a major obstacle for achieving timing closure, typically requiring numerous expensive design iterations. Current logic synthesis strategies often sacrifice other performance metrics to improve delay, requiring computationally expensive algorithms and companion placements. Despite these efforts, extensive post-placement optimizations are still needed, especially since buffers will represent a large fraction of standard cells in future technologies [29].

We propose a solution that improves the quality of delay optimization without sacrificing other performance metrics. To this end, we introduce a novel simulation-guided synthesis strategy that is more comprehensive than current restructuring techniques. We develop a path-monotonicity metric to focus our efforts on the most important parts of a design. Our optimizations lead to 11.7% delay improvement on average over several different initial placements, while our delay-based monotonicity metric indicated that 65% of the paths analyzed were non-monotonic. We further observe delay improvements on placements initially optimized for delay, which are consistent with our reported average improvement. We believe that our approach offers an effective bridge between current topological-based synthesis and lower-level physical synthesis approaches. It enables less conservative estimates early in the design flow to improve other performance metrics and reduce the amount of buffering required by shortening critical paths. Future work will explore the benefits of using our technique with other physical synthesis strategies such as buffering.

References

[1] A. Ajami and M. Pedram, "Post-layout timing-driven cell placement using an accurate net length model with movable steiner points", *DAC '01*, pp. 595-600.

[2] C. Alpert, A. Devgan, and C. Kashyap, "RC delay metric for performance optimization", *TCAD '01*, pp. 571-582.

[3] C. Alpert, A. Kahng, C. Sze, and Q. Wang, "Timing-driven steiner trees are (practically) free", *DAC '06*, pp. 389-392.

[4] G. Beraudo and J. Lillis, "Timing optimization of FPGA placements by logic replication", *DAC '03*, pp. 541-548.

[5] D. Brand, "Verification of large synthesized designs", *ICCAD '93*, pp. 534-537.

[6] A. Caldwell, A. Kahng, and I. Markov, "Can recursive bisection alone produce routable placements?", *DAC'00*, pp. 693-698.

[7] K.-H. Chang, I. Markov, and V. Bertacco, "Fixing design errors with counterexamples and resynthesis", *ASP-DAC '07*. pp. 944-949.

[8] K.-H. Chang, I. Markov, and V. Bertacco, "Safe delay optimization for physical synthesis", *ASP-DAC '07*. pp. 628-633.

[9] K.-H Chang, D. Papa, I. Markov, and V. Bertacco, "InVerS: an incremental verification system with circuit similarity metrics and error visualization", *ISQED '07*. pp. 487-494.

[10] S. Chatterjee and R. Brayton, "A new incremental placement algorithm and its application to congestion-aware divisor extraction", *ICCAD '04*, pp. 541-548.

[11] C.-W Chang, C.-K Cheng, P. Suaris, and M. Marek-Sadowska, "Fast post-placement rewiring using easily detectable functional symmetries", *DAC '00*, pp. 286-289.

[12] C. Chu and Y.-C. Wong, "Fast and accurate rectilinear steiner minimal tree algorithm for VLSI design", *ISPD'05*, pp. 28-35. (http://class.ee.iastate.edu/cnchu/flute.html)

[13] N. Een and N. Sorensson, "An extensible SAT-solver", *SAT '03*, (http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/).

[14] W. Gosti, A. Narayan, R. Brayton, and A. Sangiovanni-Vincentelli, "Wireplanning in logic synthesis", *ICCAD '98*, pp. 26-33.

[15] W. Gosti, S. Khatri, and A. Sangiovanni-Vincentelli, "Addressing the timing closure problem by integrating logic optimization and placement", *ICCAD '01*, pp. 224-231.

[16] M. Hrkic, J. Lillis, and G. Beraudo, "An approach to placement-coupled logic replication", *DAC '04*.

[17] Y.-Min. Jiang, A Krstic, K.-Ting Cheng, M. Marek-Sadowska, "Post-layout Logic Restructuring For Performance Optimization", *DAC '97*, pp. 662-665.

[18] L. Kannan, P. Suaris, and H. Fang, "A methodology and algorithms for post-placement delay optimization", *DAC'94*, pp. 327-332.

[19] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification", *TCAD '02*, pp. 1377-1394.

[20] V. Kravets and K. Sakallah, "Resynthesis of multi-level circuits under tight constraints using symbolic optimization", *ICCAD '02*, pp. 687–693.

[21] C. Li, C-K. Koh, and P. Madden, "Floorplan management: incremental placement for gate sizing and buffer insertion", *ASP-DAC'05*, pp. 349-354.

[22] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis", *DAC '06*, pp. 532-536.

[23] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification", *ERL Technical Report '05*, Berkeley. (http://www.eecs.berkeley.edu/∼alanmi/publications/).

[24] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver", *DAC '01*, pp. 530-535.

[25] R. Otten and R. Brayton. "Planning for performance", *DAC '98*, pp. 122-127.

[26] M. Pedram and N. Bhat. "Layout driven logic restructuring/decomposition", *ICCAD '91*, pp. 134-137.

[27] S. Plaza, I. Markov, and V. Bertacco, "Optimizing non-monotonic interconnect using functional simulation and logic restructuring", *ISPD '08*, pp. 95-102.

[28] S. Plaza, K.-H Chang, I. Markov, and V. Bertacco, "Node mergers in the presence of don't cares", *ASP-DAC '06*, pp. 414-419.

[29] P. Saxena, N. Menezes, P. Cocchini, and D. Kirkpatrick, "Repeater scaling and its impact on CAD", *TCAD '04*, pp. 451-463.

[30] J. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability", *IEEE Trans. Comp '99*, pp. 506-521.

[31] S. Sinha, A. Mishchenko, and R. Brayton, "Topologically constrained logic synthesis", *ICCAD '02*, pp. 679-686.

[32] G. Stenz, B. Riess, B. Rohfleisch, and F. Johannes, "Timing driven placement in interaction with netlist transformations", *ISPD '97*, pp. 36-41.

[33] L.P.P.P van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay", *ISCAS '90*, pp. 865-868.

[34] J. Werber, D. Rautenbach, and C. Szegedy, "Timing optimization by restructuring long combinatorial paths", *ICCAD '07*, pp. 536-543.

[35] S. Yamashita, H. Sawada, and A. Nagoya, "SPFD: A New Method to Express Functional Flexibility", *TCAD '00*, pp. 840-849.

[36] Y.-S. Yang, S. Sinha, A. Veneris, and R. Brayton, "Automating logic rectification by approximate SPFDs", *ASP-DAC '07*, pp. 402–407.

[37] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't cares", *DAC '06*, pp. 229-234.

[38] The International Technology Roadmap for Semiconductors, 2005 Edition, ITRS.

[39] `http://iwls.org/iwls2005/benchmarks.html`.

[40] Synopsys DesignCompiler. `http://www.synopsys.com`.

[41] UMICH Physical Design Tools, `http://vlsicad.eecs.umich.edu/BK/PDtools/`