Scalable Don't-Care-Based Logic Optimization and Resynthesis

ALAN MISHCHENKO and ROBERT BRAYTON, University of California, Berkeley
JIE-HONG ROLAND JIANG, National Taiwan University
STEPHEN JANG, Xilinx Inc.

We describe an optimization method for combinational and sequential logic networks, with emphasis on scalability. The proposed resynthesis (a) is capable of substantial logic restructuring, (b) is customizable to solve a variety of optimization tasks, and (c) has reasonable runtime on industrial designs. The approach uses don't-cares computed for a window surrounding a node and can take into account external don't-cares (e.g., unreachable states). It uses a SAT solver for all aspects of Boolean manipulation: computing don't-cares for a node in the window, and deriving a new Boolean function of the node after resubstitution. Experimental results on 6-input LUT networks after a high effort synthesis show substantial reductions in area and delay. When applied to 20 large academic benchmarks, the LUT counts and logic levels are reduced by 45.0% and 12.2%, respectively. The longest runtime for synthesis and mapping is about two minutes. When applied to a set of 14 industrial benchmarks ranging up to 83K 6-LUTs, the LUT counts and logic levels are reduced by 11.8% and 16.5%, respectively. The longest runtime is about 30 minutes.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Optimization*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*

General Terms: Algorithms, Performance, Design, Experimentation

Additional Key Words and Phrases: FPGA, don't-cares, resynthesis, Boolean satisfiability

ACM Reference Format:
Mishchenko, A., Brayton, R., Jiang, J.-H. R., and Jang, S. 2011. Scalable don't-care-based logic optimization and resynthesis. ACM Trans. Reconfig. Technol. Syst. 4, 4, Article 34 (December 2011), 23 pages.
DOI = 10.1145/2068716.2068720 http://doi.acm.org/10.1145/2068716.2068720

1. INTRODUCTION

The sizes of industrial FPGAs have grown significantly in the last several years. The Virtex-5 FPGA family [Xilinx 2011] contains up to 207K 6-LUTs. The Stratix IV family [Altera 2011] contains up to 212K Adaptive Logic Modules (ALMs). It is expected that the capacity of FPGAs will continue to grow in the coming years. As a result, scalability is becoming a challenge in the design flow, and in particular, in logic synthesis.

Traditional optimizations of Boolean networks using don't-cares (SDC, ODC, EXDC) are based on logic minimization packages, such as Espresso, and on logic representations, such as SOPs and BDDs. Logic restructuring with Boolean resubstitution is

S. Jang is currently affiliated with Agate Logic.
This work was supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668 entitled "Sequentially Transparent Synthesis", and the California Micro Program with industrial sponsors Actel, Altera, Atrenta, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys, Synplicity (Synopsys), Tabula, Verific, and Xilinx.
Authors' addresses: A. Mishchenko (corresponding author) and R. Brayton, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720; email: alanmi@eecs.berkeley.edu; J.-H. R. Jiang, Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan 10617; S. Jang, Agate Logic Inc., 1237 East Argues Ave., Sunnyvale, CA 94085.

DOI 10.1145/2068716.2068720 http://doi.acm.org/10.1145/2068716.2068720

traditionally done by modifying the set of fanins of a node under Satisfiability Don't-Cares (SDCs). If External Don't-Cares (EXDCs) are present, they can enhance the effect of SDCs. EXDCs are typically represented by BDDs or by a separate Boolean network in terms of the input variables. These methods are not scalable for large designs and therefore rarely used.

We give a method that avoids these nonscalable techniques. Scalability is achieved by:

—optimizing a node in a local window as in Mishchenko et al. [2006b],
—extracting and representing EXDCs in terms of clauses on k-cuts in an underlying AIG of the network [Case et al. 2008], and
—avoiding SOPs and BDDs, and instead using SAT and interpolation[McMillan et al. 2003].

Several preliminary aspects of this work appeared earlier in the following publications: don't-care computation [Mishchenko et al. 2005], computation of sequential don't-cares [Case et al. 2008], SAT-based resubstitution [Mishchenko et al. 2006], and resubstitution with don't-cares [Mishchenko et al. 2009].

The current work differs as follows. The algorithms for window computation have been further improved and made more scalable, compared to Mishchenko et al. [2005]. The use of Boolean satisfiability for resynthesis [Mishchenko et al. 2009] has been perfected by utilizing the interpolation procedure, which allows the SAT solver to use don't-cares without explicitly computing them. Another practical innovation is uniform handling of combinational and sequential don't-cares, compared to Mishchenko et al. [2005] and Case et al. [2008]. This is achieved by flexibly adjusting the window boundary to fit the care clauses precomputed for the network.

These ideas are implemented in the system, ABC [Berkeley 2011], as command *mfs*. For sequential networks, care sets characterizing the reachable state space (complement of EXDCs) are extracted directly in terms of clauses on k-cuts of the AIG. These clauses restrict the states to an overapproximation of the reachable states. The methods are shown experimentally to be scalable and effective.

The rest of the article is organized as follows. Section 2 describes some background. Section 3 describes the method for extracting external care sets in terms of k-clauses. Section 4 discusses optimization based on windowing, SAT solving, interpolation, and use of care clauses. Section 5 reviews relevant previous work. Section 6 reports experimental results. Section 7 concludes the article and outlines future work.

2. BACKGROUND

2.1. Networks and Nodes

A *Boolean network* is a Directed Acyclic Graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network and circuit are used interchangeably.

A node has zero or more *fanins,* that is, nodes that are driving this node, and zero or more *fanouts*, that is, nodes driven by this node. The *Primary Inputs* (PIs) are nodes without fanins. The *Primary Outputs* (POs) are a subset of the nodes of the network. The Transitive FanIn (TFI) of a node includes the node, its fanins, the fanins of the fanins, etc., until the PIs are reached. The Transitive FanOut (TFO) of a node includes the node, its fanouts, the fanouts of the fanouts, etc., until the POs are reached. If the network is sequential, it contains registers whose inputs/outputs are treated as additional POs/PIs.

A combinational network can be expressed as an And-Inverter Graph (AIG), composed of two-input ANDs and inverters represented as complemented attributes on the edges. Optimizations of this article are applicable to both AIGs and general-case logic networks.

A *cut C* of node n, called the *root*, is a set of nodes, called *leaves*, such that each path from a PI to n passes through at least one leaf. A cut is *K-feasible* if its cardinality does not exceed K. A cut is *dominated* if there is another cut of the same node, contained, set-theoretically, in the given cut. A *fanin* (*fanout*) *cone* of node n is the subset of the nodes of the network reachable through the fanin (fanout) edges from n.

2.2. Don't-Cares and Resubstitution

Internal flexibilities of a node arise because of limited controllability and observability of the node. Noncontrollability occurs because some combinations of values are never produced at the fanins. Nonobservability occurs because the node's effect on the POs is blocked under some combination of the PI values. Examples can be found in Mishchenko et al. [2005].

These internal flexibilities result in *don't-cares* at the node n. They can be represented by a Boolean function whose inputs are the fanins of the node and whose output is 1 when the value produced by the node does not affect the functionality of the network. The complement of this function gives the *care set*.

Given a network with PIs x and PO functions $\{z_i(x)\}$, the care set $C_n(x)$ of a node n is a Boolean function of the PIs

$$C_n(x) \sum_i [z_i(x) \oplus z_i'(x)],$$

where $z_i'(x)$ are the POs in a copy of the network but with node n complemented [Mishchenko et al. 2005].

Traditionally, subsets of don't-cares are derived and used to optimize a node [Savoj et al. 1992; Mishchenko et al. 2005]. This optimization may involve minimizing the node's function in isolation from other nodes, or expressing the node in terms of a different set of fanins. The former transformation is known as *don't-care-based optimization*; the latter is *resubstitution*. The potential new fanins of the node are its *resubstitution candidates*. A set of resubstitution candidates is feasible if the node can be reexpressed using the new fanins without changing the functionality of the network.

A necessary and sufficient *condition for the existence of resubstitution* is given in Mishchenko et al. [2006b, Theorem 5.1].

THEOREM 2.2.1. *There exists a function $h(g)$ of functions, $\{g_i(x)\}$, such that $C(x) \Rightarrow [f(x) = h(g(x))]$ if and only if there is no minterm pair (x_1, x_2), such that $f(x_1) \neq f(x_2)$ while $g_j(x_1) = g_j(x_2)$, for all j, where $C(x_1) \wedge C(x_2) = 1$.*

In the preceding formulation, minterm stands for a vector of assignments for all variables. Informally, resubstitution exists if and only if, on the care set, the capability of the set of functions $\{g_i(x)\}$ to distinguish minterms is no less than that of function $f(x)$.

2.3. Optimization with Don't-Cares

Computation of don't-cares involves exploring fanin and fanout cones, but if the network is large, it is infeasible to do this while considering the whole network as the context of each node. Therefore computation for a node is limited to a local neighborhood of the node, called a *window*. The node to be optimized is called the *pivot* and the scope of a window containing the pivot is controlled by user-specified parameters, for example, the number of fanin and fanout levels spanned, the number of inputs, outputs, and internal nodes of the window.

When a don't-care is computed for a node, it should be used immediately and the network updated before optimizing another node. This avoids don't-care incompatibility issues arising when don't-cares are computed for several nodes before they are used.

These window-based methods use ODCs and SDCs, extracted from the window. The use of EXDCs with window methods is problematic because they are usually expressed in terms of the PIs of the network, and for each window, they must be projected to the window's PIs. Typically EXDCs are hard to represent and generally hard to project to a window. Therefore, the use of EXDCs has been seen as not scalable, except for small circuits. In Section 4.3, we describe a new scalable method for computing and using information about EXDCs, appropriate for window-based optimization.

2.4. Interpolation

Given Boolean functions, $(A(x, y), B(y, z))$, such that $A(x, y) \wedge B(y, z) = 0$, and (x, y, z) is a partition of the variables, an *interpolant* is a Boolean function, $I(y)$, such that $A(x, y) \subseteq I(y) \subseteq \overline{B(y, z)}$. If $A(x, y)$ and $B(y, z)$ are sets of clauses, then their conjunction is an unsatisfiable SAT instance, and an interpolant can be computed from a proof of unsatisfiability using the algorithm in McMillan et al. [2003, Definition 2].

$A(x, y)$ can be interpreted as the onset of a function, $B(y, z)$ as the offset, and $\overline{A(x, y)} \wedge \overline{B(y, z)}$ as the don't-care set. Thus $I(y)$ can be seen as an optimized version of $A(x, y)$ where the don't-cares are used somehow, for example, I is only a function of the variables y.

3. EXTRACTING EXTERNAL CARE CLAUSES

In this section, we give an overview of how we obtain information about EXDCs, represented in terms of a set of care k-clauses, whose conjunction overapproximates the reachable state space of a sequential circuit. This is done by induction and is based on ideas detailed in Case et al. [2008]. Once extracted, the set of clauses can be stored and later used by the ABC command *mfs*. This algorithm is described in Section 4.2.

We compute a set of clauses on a set of k-cuts of an AIG representing the sequential circuit, as an inductive invariant. Previous methods obtain external don't-cares by computing the set (or subset) of unreachable states characterized by a function of the register outputs. To use these as well as SDCs and ODCs in a scalable way, the circuit is temporarily restricted to a window around a node to be simplified and the EXDCs must be projected onto a set of nodes or inputs of the window being used. Thus, the useful parts of the unreachable set are those which have nontrivial such projections. In contrast, we do not compute the set of unreachable states, but directly compute a set of its projections onto various cuts of the AIG.

These projections are computed by induction [Bjesse et al. 2000], which is one of the most practical methods for characterizing an approximate set of reachable states. The computation of such sets is applicable to large designs whose size and logic complexity would cause other methods (such as BDD-based reachability) to fail.

An invariant is *inductive* if it satisfies two conditions: *base case*, that is, it holds in the initial state, and *inductive case*, that is, if it holds in a state, then it holds in all states reachable from that state in one transition. Induction is scalable because both the base and inductive cases can be formulated as incremental instances of Boolean satisfiability (SAT), which can be solved efficiently using modern SAT solvers [Een et al. 2003].

In our method, the invariant is a set of clauses, in which a group of variables participating in a clause is derived using an efficient k-cut computation method adopted from technology mapping [Chen et al. 2004, Pan et al. 1998]. It avoids exhaustive k-cut enumeration and computes only a small subset of useful cuts using priority heuristics similar to those in Mishchenko et al. [2007b].

An initial set of candidate clauses is detected using simulation. Two types of random simulation are used: combinational and sequential. Sequential simulation starts at the

initial state and only visits reachable states from there. Minterms at the inputs of a k-cut that appear under combinational simulation and do not appear under sequential simulation are recorded. Then, a *candidate* clause is derived as the complement of such a minterm, provided that is satisfies the initial state of the circuit. The starting set of candidate clauses is iteratively refined using SAT-based induction, throwing out those clauses that do not hold inductively. The greatest fixed-point of this computation yields an inductive invariant (the conjunction of the remaining clauses) and represents an overapproximation to the reachable state set.

To make this computation efficient, a flexible framework heuristically trades the number and expressiveness of the clauses for computation time. Invariant sets can be proved in batches, each of which successively tightens the already computed invariant. The process is stopped when a resource limit is reached.

Scalability is achieved by using a heuristic for candidate clause generation and filtering. This counts the number of times N a minterm appears in combinational simulation but never in sequential simulation. Such minterms are more likely excluded from sequential simulation because they characterize unreachable states. The minterms are prioritized according to N and the top M (user specified) are selected and complemented to obtain the initial set of candidate clauses. Inductive proofs composed of such sets can be processed efficiently by partitioning the clauses and solving their logic cones in parallel without sacrificing the completeness of the result. A similar approach was used in Case et al. [2006]. In typical runs, several thousand clauses are selected initially with only a few hundred ending up in the invariant.

4. OPTIMIZATION AND RESYNTHESIS ALGORITHM

During optimization, the nodes of a *mapped* circuit are visited and optimized one at a time. Since the optimization order appears (experimentally) to be unimportant for large circuits, we use a topological order because it is simpler to compute.

Figure 1 shows pseudocode of our node optimization procedure based on structural analysis (windowing), satisfiability, SAT solving, and interpolation. It uses two windows: an outer window to provide the environment, from which cares are extracted (or represented) for the inner window (explained in detail in Section 4.3).

The parameters used by this procedure include the following:

—the number of fanin/fanout levels of the inner and outer windows to be used,
—the limit on PIs and internal nodes of the inner window,
—the largest number of Boolean divisors to collect,
—the runtime limit for the don't-care computation,
—the number of random patterns to simulate,
—the simulation success rate determining when random simulation is replaced by constrained guided simulation performed by the SAT solver,
—the SAT solver runtime and conflict limits,
—the resubstitution objective function based on the goals of resynthesis (e.g., area, delay, power).

The following subsections provide details on the theory and implementation of each part of the preceding resynthesis procedure, applied to the pivot node of the window (*node*).

4.1. Windowing

This subsection describes a windowing algorithm and its use. The procedure is the same for the construction of both the inner and outer windows, but the parameters are different.

```
nodeOptimization( node, iparameters, oparameters ) {

    // compute inner window for the node with the given parameters
    innerwindow = nodeWindow( node, iparameters );

    // compute outer window for care set computation
    outerwindow = nodeWindow( node, oparameters );

    // compute care set for inner window
    CS = computeCareSet( node, innerwindow, outerwindow );

    // collect candidate divisors of the node
    divisors = nodeDivisors( node, innerwindow, parameters );

    // find sets of resubstitution candidates using simulation as a filter
    cands = nodeResubCandsFilter( node, divisors, innerwindow, CS );

    // iterate through the sets of resubstitution candidates and evaluate
    best_cand = NULL;
    for each candidate set c in cands {

    // skip candidates that are worse that the given one
    if ( best_cand != NULL && resubCost(best_cand) < resubCost(c) )
        continue;

    // skip infeasible resubsitution candidates disproved by SAT
    if ( !resubFeasible( node, innerwindow, CS,  c ) )
        continue;

    // save the candidate that is feasible and better than the best
    best_cand = c;
  }

  // update the network if a feasible candidate is found
  if ( best_cand != NULL ) {

      // compute new dependency function using interpolation
      best_func = nodeInterpolate( node, best_cand, CS );

      // update the network by replacing the current node
      nodeUpdate( node, best_cand, best_func );
  }
}
```

Fig. 1. Don't-care-based optimization of a node.

Figure 2 summarizes a windowing procedure taking the pivot node (*node*) and two parameters (*tfi_level_max, tfo_level_max*), which determine the maximum number of TFI and TFO levels spanned by the window.

First, the TFI cone of the pivot is computed using a reverse topological traversal, reaching for several levels towards the PIs. The PIs of the window are detected as the nodes that are not in this cone but have fanouts in it. Next, the TFO cone of the pivot is computed by a topological traversal reaching for several levels towards the POs. If the TFO cone is empty (for example, if the pivot is a PO), the procedure

```
nodeWindow( node, tfi_level_max, tfo_level_max )  {

   // compute the TFI cone of the node with at most tfi_level_max levels
   tfi_cone = nodeTfiCone( node, tfi_level_max );

   // compute the PIs of the TFI cone
   window_pis = conePIs( tfi_cone );

   // compute the TFO cone of the node with at most tfo_level_max levels
   tfo_cone = nodeTfoCone( node, tfo_level_max );

   // return if the TFO cone is trivial
   if ( tfo_cone == ∅ )
        return coneCollectNodes( {node}, window_pis );

   // compute the POs of the TFO cone
   window_pos = conePOs( tfo_cone );

   // traverse the TFI of window_pos and mark the paths to window_pis
   // while skipping the paths going through the pivot node
   coneMarkPaths( window_pos, window_pis, node );

   // remove the nodes in the TFO without marked paths to window_pis
   coneFilterTfo( tfo_cone );

   // compute the POs of the reduced TFO cone
   window_pos = conePOs( tfo_cone );

   // return the nodes on the paths from window_pos to window_pis
   return coneCollectNodes( window_pos, window_pis );
}
```

Fig. 2. Improved windowing algorithm.

returns the window composed of nodes found on the paths between the pivot and the PIs.

If the TFO cone is not empty, the POs of the cone are detected as the nodes that are in the cone but have fanouts outside of it. Next, a reverse topological traversal is performed from the window POs towards the window PIs while skipping the paths through the pivot. This traversal is useful to detect reconvergent paths between the window POs and window PIs that do not include the pivot node. The scope of this traversal is made local by finding the lowest level of the window PIs and not traversing below that level.

Since some nodes in the TFO cone may have no path to any of the window PIs, these nodes are removed from the TFO cone because they cannot produce observability don't-cares. Since the TFO cone may have changed, the window POs are recomputed. Finally, all nodes on the paths from the updated window POs to the window PIs are collected and returned as the window. This traversal augments the set of the window PIs with those fanins of the collected nodes that are not on the paths to the window PIs.

4.2. Computing the Care Set for an Inner Window

The configuration shown in Figure 3 has two windows: an outer window containing an inner window. The inner window contains the pivot node $f(y)$. In this subsection, we consider the outer window with PIs x and the POs z and the inputs s to the inner

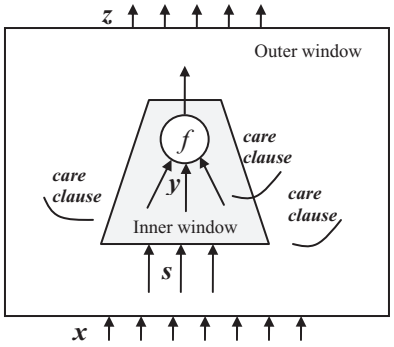

Fig. 3. The inner and outer windows for node f.

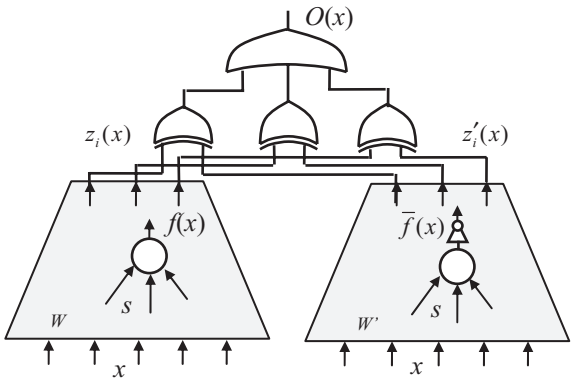

Fig. 4. Miter for care set computation.

window. We use these to extract a care set for the inner window in terms of its inputs s. The care clauses shown in Figure 3 are additional constraints on the satisfiable assignments of the SAT problem, which correspond to the care minterms. The more powerful are these constraints, the more don't-cares are computed.

SAT and random simulation are used in the care set computation. The care set for f in the s-space is extracted from this window. We first describe the case where there are no external care clauses. A miter is formed as shown in Figure 4, where the inner window is shown as a circle and the function f is viewed as a global function of the outer window PIs, x. When a 1 is asserted at the output of the miter $O(x)$, a solution of the SAT problem corresponding to this miter gives a satisfying assignment for all network signals. The values of variables s (the PIs of the inner window) in this assignment form a *care set minterm*, m_s, in the s-space. This is because, for the corresponding m_x of the assignment, at least one pair of POs, $(z_i(m_x), z_i'(m_x))$, has opposite values and thus we care about the output of f.

All such care set minterms, m_s, are collected by enumerating the satisfying assignments of the SAT problem where the variables s are the projection variables.

When there are care clauses, we look for those whose support variables are all contained in the outer window. Such clauses are shown as curvy lines in Figure 3. If any are found, they are added to the set of clauses generated for the outer windows in both copies, as shown in Figure 4, to form the SAT instance. Optionally, if there are

```
function CompleteCare( innerwindow N , outerwindow W )
{
    aig G = ConstructMiter( W, N );
    function F1(y)= RandomSimulation( G );
    cnf P = CircuitToCNF( G ) ∧ FunctionToCNF( !F1 );
    cnf P = cnf P ∧ CareClauses (W) ∧ CareClauses (W');
    function F2 = SatSolutions( P );
    return F1 + F2;
}
```

Fig. 5. Pseudocode of SAT-based care computation.

care clauses with some variables outside of the outer window, the outer window can be expanded to include them as PIs.

The SAT-based care computation is summarized in Figure 5. The top-level procedure *CompleteCare* takes inner window N and its context W (outer window). Procedure *ConstructMiter* applies structural hashing to the miter of the two copies of W (W and W', as shown in Figure 4). The resulting compacted AIG G is constructed in one DFS traversal of the nodes in the miter, without actual duplication.

For efficiency, random simulation is used to derive part of the care set, F_1. The CNF P is the conjunction of clauses derived from G and the complement of F_1. The CNF is conjoined with the care clauses falling within W and W' obtained by the procedure *CareClauses*(). Finally, a 1 is asserted at the PO of the miter.

The all-SAT solver *SatSolutions* enumerates through the remaining satisfying solutions, F_2, of the care set. In practice, it often happens that the SAT problem has no solution ($F_2 = 0$), but SAT is still needed to prove the completeness of the care set derived by random simulation.

Since this approach enumerates through the satisfying assignments that represent *s-minterms* of the inner window PIs, $|s|$ should be limited by roughly 10 inputs or less. The size of $|x|$ is less important. To make the approach appropriate for networks with individual nodes with large fanins, such nodes should be decomposed first. The implementation of the SAT solver should be further modified to return incomplete satisfying assignments corresponding to *cubes* rather than minterms of s.

Once the care minterms of the inner window PIs have been found, the outer window can be ignored.

4.3. Using Don't-Care Set for Node Minimization

The complete don't-care set computed for the outer window can be used to optimize the inner window, by applying them as constraints in SAT solving. Alternatively, if the inner window is a node, the don't-cares explicitly computed, as shown in Section 4.2, can be used to optimize the node directly. In this subsection, we describe the use of bidecomposition [Mishchenko et al. 2001] for the direct optimization of the node.

In this work, the nodes of the network correspond to LUTs after technology mapping. The complexity measure of the node's function is the number of AIG nodes in its local function. The proposed optimization of the node consists in converting its local function into a truth table, and subjecting it to a decomposition, which may result in a different AIG. If the resulting AIG is smaller than the original one, the representation of the local function is updated. If not, the node is left unchanged, and the computation moves on to the next node. The optimization potential of this transformation is enhanced through the use of don't-cares computed as shown before, and bidecomposition.

Bidecomposition [Mishchenko et al. 2001] is a Boolean decomposition algorithm applied to an incompletely specified function represented as a truth table, resulting

Table I. Checking Resubstitution Using Simulation

$a\bar{b}c$	f	Set 1		Set 2	
		$g_1 = \bar{a}b$	$g_2 = a\bar{b}c$	$g_3 = a + b$	$g_4 = bc$
000	0	0	0	0	0
001	0	0	0	0	0
010	1	1	0	1	0
011	1	1	0	1	1
100	0	0	0	1	0
101	1	0	1	1	0
110	0	0	0	1	0
111	0	0	0	1	1

in a multilevel representation of the function using two-input gates (AND, OR, and XOR) and MUXes, all of which can be represented as an AIG. The algorithm is called bidecomposition because it proceeds by recursively dividing an incompletely specified function into two logic blocks feeding into a two-input gate realizing the function. The advantages of bidecomposition over other algorithms are the following.

—It efficiently utilizes both the external and the internally generated don't-cares. The don't-cares are propagated recursively as the computation proceeds, and the resulting completely specified function is guaranteed to agree with the initial incompletely specified function.
—It detects logic sharing across the components and allows for minimizing area of the resulting decompositions.
—It allows for creating well-balanced multilevel structures, which leads to delay minimization.

The last two observations are especially important when bidecomposition is used in the inner-loop of an interactive logic optimization, as is the case in the proposed work.

This way of using explicitly computed don't-cares to optimize the node is implemented in command *mfs –r* in ABC. Refer to the experimental results section to see the contribution of this technique to the flow based on the implicit use of don't-cares.

4.4. Candidate Divisors and Resubstitutions

At this stage, we have the pivot node $f(y)$, an inner window containing node f and with inputs s, and a characterization of a care set $C(s)$. The idea is to reexpress $f(y)$ as $h(g)$, where $g = \{g_j(s)\}$. The $g_j(s)$ are a subset of a set of nodes called candidate Boolean divisors of f. These are nodes already existing in the inner window (called "window" now) that can be used as inputs to a new function that will replace f. Note that the current inputs y of $f(y)$ are included in the candidate set. To collect the candidates, first the window PIs, s, are divided into: (a) those in the TFI cone of the pivot node, and (b) the remainder. All nodes on paths between the pivot and the PIs of type (a) are added to the set of candidates, excluding the node itself and any node in the fanout cone of the pivot. Second, other nodes of the window are added if their structural support has no window PIs of type (b). A resource limit is used to control the number of collected candidate divisors. In most cases, collecting up to 100 candidate divisors works well in practice, while adding only about 5% to the resynthesis runtime.

The following example from Mishchenko et al. [2006b] illustrates the use of simulation for filtering resubstitution candidates. Consider function $f = (a \oplus b)(b \vee c)$ and two sets of candidate functions: ($g_1 = \bar{a}b$, $g_2 = a\bar{b}c$) and ($g_3 = a \vee b$, $g_4 = bc$). Table I shows the truth tables of all functions. The set (g_3, g_4) is not a valid resubstitution candidate set for f because minterm pair (101, 110), which can be found by simulation, is distinguished by f but not distinguished by (g_3, g_4). However, the set (g_1, g_2) satisfies

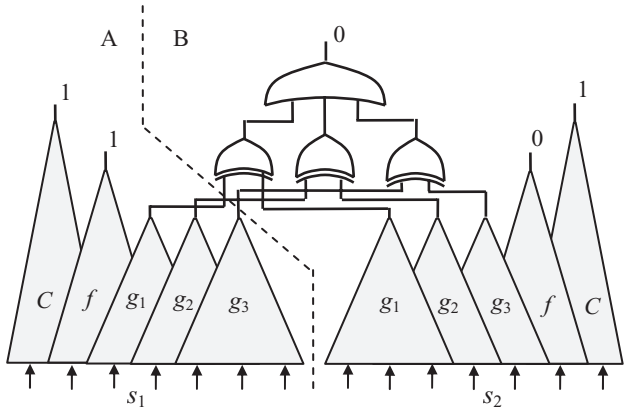

Fig. 6. Miter for checking resubstitution using SAT.

the condition of Theorem 2.2.1 because all the minterm pairs distinguished by f are also distinguished by g_1 or g_2.

4.5. Checking Resubstitution Using SAT

Simulation only filters out some resubstitution sets while the remaining ones have to be checked. Checking a candidate resubstitution set is done by generating a SAT instance which reflects the conditions of Theorem 2.2.1.

Figure 6 shows the circuit representation of the SAT instance. The left and right parts of the figure contain structurally identical logic cones for: (1) the care set C, (2) the node's function f, and (3) candidate divisors $\{g_i\}$ expressed using variable sets s_1 and s_2, respectively. Assignments of variables s_1 and s_2 represent two minterms. The circuitry in the middle expresses that all the functions $\{g_i\}$ are equal for these two minterms. The output of f is asserted to 1 on the left and 0 on the right, meaning that f takes different values for this pair of minterms. Finally, both the left and right care sets C are set to 1 to restrict both minterms to be in the care set.

It should be noted that this procedure does not extract the care set $C(s)$, as described in Section 4.2. Instead, the care set is represented as miter C, shown in Figure 4. This miter is added to both sides of Figure 6, depending on both s_1 and s_2.

If the SAT instance is satisfiable, then resubstitution with the given functions $\{g_i\}$ in the resubstitution set does not exist and is discarded. The counter-example derived by the SAT solver is used to filter the remaining candidates. If the SAT instance is unsatisfiable, the resubstitution exists and a dependency function $h(g)$ can be derived, as shown next.

4.6. Deriving a Dependency Function by Interpolation

The intuition behind the use of interpolation is that it implicitly uses the flexibility and derives a dependency function that can be used to replace the original one. The optimality of this function is not important as long as it fits into one LUT, which is achieved constructively by limiting the number of considered divisors.

We seek a "dependency" function $h(g)$, such that $C(s) \Rightarrow [h(g(s)) \equiv f(s)]$, to express f using candidates $\{g_i\}$. The computation of $h(g)$ can be done using SOPs [Savoj et al. 1992], BDDs [Kravets et al. 2004], SPFDs [Cong et al. 2002], or by enumerating satisfying assignments of a SAT problem [Mishchenko et al. 2005]. We follow the approach of Lee et al. [2007] based on interpolation (see Section 2.4). The advantage is that $h(g)$ is computed as a byproduct of the resubstitution feasibility check (see Section 4.4).

To compute a function $h(g)$, the clauses of the SAT instance are divided into subsets $A(x, y)$ and $B(y, z)$ derived from the two parts of the circuit separated by the dashed line, as shown in Figure 6. In this case, the common variables, y, of the interpolant are the outputs of functions g_i. They constitute the support of the resulting dependency function.

The proof of unsatisfiability needed for interpolation is generated, as shown in Goldberg et al. [2003]. For this, the SAT solver [Een et al. 2003] is minimally modified (by adding exactly five lines of code) to save both the original problem clauses and the learned clauses derived by the solver during the proof of unsatisfiability. The last clause derived is the empty clause, which is also added to the set of saved clauses.

The interpolation computation works on the set of all clauses, partitioned into three subsets: clauses of $A(x,y)$, clauses of $B(y,z)$, and the learned clauses. It considers the learned clauses in their order of generation during the proof of unsatisfiability. For each learned clause, a fragment of a resolution proof is computed and converted into an interpolant on-the-fly. The interpolant of the last (empty) clause is returned.

Since for most applications (for example, netlist rewiring) the support of the dependency function is small, the interpolant can be computed using truth tables. This is in contrast to the general case where the interpolant is constructed as a multilevel circuit. The previous approach is efficient for typical SAT instances encountered in checking resubstitution. In our experiments, the runtime of interpolation did not exceed 5% of the total runtime.

4.7. Maintaining the Care Clauses

As the circuit is restructured, some nodes are removed and new nodes may appear. When the care clauses were derived, it was with reference to a particular circuit, so the clauses may refer to nodes that have been removed. It might be useful to update the clause set to refer to existing clauses. This can be done by taking each clause that refers to a nonexisting variable, and for each missing variable finding a cut in terms of existing variables. Then compute the preimage of the clause minterm (complement of the old clause) in terms of existing variables. This gives possibly a set of minterms. However, each one is a minterm that can't appear when the state space is restricted to the reachable state set. This is because if that minterm did appear, then the original minterm would also appear. For each of the minterms in this preimage, complement this and add the resulting clause to the care clauses, while removing the old clause. In this way no information is lost about the set of reachable states.

This same computation can be used to project more don't-care information into a window. For example, suppose the set of nodes of a care clause is not in the outer window but a common cut of these nodes is in the outer window. In this case, we can compute the preimage of the complement of this clause and derive a set of equivalent clauses used in computing the care set of the window.

4.8. Resynthesis Heuristics

These heuristics express the goal of resynthesis in terms of the type of resubstitutions attempted. Before resynthesis begins, the network is scanned to find: (a) the set of nodes that will be targeted by resubstitution, and (b) the priority of those nodes. The targeted nodes are considered in the order of their priority. For each target, a window is computed and a set of candidate divisors is collected (within resource limits). The candidate divisors are the nodes whose support is a subset of the inner window PIs and (if increased delay is of concern) whose arrival times do not exceed the required time of the targeted node minus the estimated delay of the new function at the node after resubstitution. Next, the resubstitution candidates of the window are processed as shown in Figure 1.

The following paragraphs discuss several types of resynthesis.

Area minimization. For this, the network is scanned to find the node having the largest MFFC and a fanout of 1. Such a node has a good potential for area saving if the function of its fanout can be expressed without this node.

Edge count minimization. When minimizing the total number of edges, any fanin of a node can be targeted. If the node's function can be expressed without this fanin, one edge is saved.

Delay minimization. This is done by detecting timing-critical edges. For level-driven optimization, an edge is critical if: (1) both the source and sink nodes are critical, and (2) the difference of the logic levels of source and sink nodes is one. A node is critical if at least one of its fanin edges is critical. The priority of an edge depends on the number of critical paths it is on. Each critical edge is targeted for resubstitution by rewiring.

4.9. Specializing SAT for Don't-Care Computation

Runtime profiling has shown that the initial steps of resynthesis, namely, determining the node to be resynthesized and computing its window, are relatively fast. Most of the runtime is spent in setting up and running the SAT problem. This runtime includes:

—Duplicating the window for the purpose of SAT solving. This is not an exact copy of the window but rather a miter, as shown in Figure 4 and Figure 6.
—Optionally synthesizing the logic of the window using fast logic synthesis, such as AIG rewriting [Mishchenko et al. 2006a].
—Converting the synthesized logic of the window into CNF using one of the CNF generation algorithms.
—Running an off-the-shelf SAT solver, such as MiniSat [Een et al. 2003], on the CNF to prove it UNSAT, or to derive a sequence of counter-examples for don't-care computation.

The preceding steps are performed for every resynthesized node, which takes a substantial runtime for large designs. Meanwhile it was found that most of the intermediate SAT problems are relatively easy and can be solved after less than 100 conflicts.

To reduce the runtime of our implementation spent in SAT solving, we intend to develop a specialized circuit-based SAT solver which works directly on the circuit structure representing the underlying problem, without the need to create a miter, derive CNF, and call MiniSat. A similar solver has been used in another project, resulting in a 10x speedup. The improvement is possible because of several simplifying assumptions that hold for the case of multiple, incremental, relatively easy SAT problems.

—The logic cone given to the solver is typically small (up to 500 AIG nodes).
—The expected number of conflicts is low (up to 100 conflicts).
—Most of the problems are satisfiable (true about 95% of problems in this work).

A prototype of the circuit-based SAT solver shows that the runtime can be substantially reduced, compared to MinSat that was used to collect experimental results included in this article. This new solver, when properly integrated, can potentially make the proposed resynthesis practical for much larger designs.

Next we describe several salient features of the circuit-based SAT solver.

—The solver works directly on the AIG data structure. For this, each node is enhanced with two bit-flags: one is used to label the node that already has a value while the other is used to remember this value (0 or 1).

—The solver uses minimalistic additional data structures. Besides the AIG, it maintains two arrays during SAT solving: the propagation queue and the justification queue. The propagation queue remembers a sequence of assignments made to the nodes so far. The justification queue at any time contains the set of AND nodes whose output is assigned 0 while both inputs are unassigned. The values of these nodes have to be justified for an assignment to be satisfiable.
—The SAT solver is called for the output of the miter and proceeds recursively towards the inputs. If constraint propagation leads to a conflict, UNSAT is returned. If the justification queue is empty, SAT is returned and the propagation queue is used to find a satisfying assignment of the primary inputs.
—The recursive procedure splits around a decision variable. If both branches return UNSAT, the result is UNSAT. If at least one of the branches is SAT, the result is SAT. If the conflict limit is reached, the procedure returns UNDECIDED.
—Decision heuristics are very simple, such as pick the last node in the topological order among the nodes that are unassigned in the TFI of the assigned nodes.
—Learned clauses are recorded, used for constraint propagation, and recycled when the solver moves on to a new part of the circuit. Nonchronological backtracking is implemented as a byproduct of conflict analysis and clause recording.

5. PREVIOUS WORK

Technology-independent optimization and postmapping resynthesis of Boolean networks using internal flexibilities have long histories [Muroga et al. 1989; Savoj et al. 1992; Mishchenko et al. 2005; Kravets et al. 2004; Chang et al. 2007] to mention a few publications. Traditional don't-care-based optimization [Savoj et al. 1992] is part of the high-effort logic optimization flow in SIS [Sentovich et al. 1992]. This optimization plays an important role in reducing area by minimizing the number of Factored Form (FF) literals before technology mapping. Its main drawback is poor scalability and excessive runtime. To cope with these problems, several window-based approaches for don't-care computation have been proposed [Mishchenko et al. 2005; Saluja et al. 2004].

Both traditional and the newer algorithms for don't-care-based optimization compute don't-cares before using them. This may explain long runtimes of these algorithms when applied to large industrial designs, even if windowing is used. A notable exception is the SAT-based approach [McMillan et al. 2005], which optimized nodes "in-place" without explicitly computing don't-cares. However, unlike Savoj et al. [1992] and Kravets et al. [2004], that work does not allow for resubstitution. As a result, the optimization space is limited to the current node boundaries. Another recent method [Lee et al. 2007] performs efficient SAT-based resubstitution but does not consider don't-cares, which may limit its optimization potential.

Some recent papers [Zhu et al. 2006; Plaza et al. 2007] propose optimization for And-Inverter Graphs (AIGs) using the notion of equivalence under don't-cares. These approaches are not applicable to postmapping resynthesis. They are also limited because they can optimize an AIG node only if there is another node with a similar logic function that can replace the given node.

It should be noted that some approaches to resynthesis [Chen et al. 1992] achieve sizeable reduction of the network without exploiting don't-cares, by precomputing all resynthesis possibilities and solving a maximum-independent set problem to perform as many resynthesis moves as possible. Compared to incremental greedy approaches based on don't-cares, this approach may have scalability issues due to the need to represent information about resynthesis possibilities for the whole network.

6. EXPERIMENTAL RESULTS

SAT-based resynthesis is implemented in ABC [Berkeley 2011] as command *mfs,* which currently performs area and edge count minimization. The SAT solver used is a modified version of MiniSat-C_v1.14.1 [Een et al. 2003]. The algorithm is applicable to a mapped network and attempts resubstitution for each gate or LUT in the netlist. Experiments targeting 6-LUT implementations were run on an Intel Xeon 2-CPU 4-core computer with 8Gb of RAM. The external don't-cares were not used. The resulting networks were checked by a combinational equivalence checker (command *dcec*) [Mishchenko et al. 2006c] and a sequential equivalence checker (command *dsec*) [Mishchenko et al. 2009; Mishchenko et al. 2008b] in ABC.

The following ABC commands are included in the scripts used to collect experimental results targeting area minimization.

—*resyn* is a logic synthesis script that performs 5 iterations of AIG rewriting [Mishchenko et al. 2006a];
—*dc2* is a logic synthesis script that performs 10 iterations of AIG rewriting;
—*dch* is a logic synthesis script that accumulates structural choices; it runs *resyn* followed by *dc2* and collects three snapshots of the network: the original, the intermediate one saved after *resyn*, and the final;
—*if* is an FPGA mapper which uses priority cuts [Mishchenko et al. 2007b], delay-optimal mapping, fine-tuned area recovery, and the capacity to map a subject graph with structural choices. The following mapper settings were used: at most 12 6-input priority cuts are computed for each node; five iterations of area recovery are performed, three with area flow and two with exact local area;
—*mfs* is the new resynthesis command of this article;
—*mfs –r* is the new command that uses don't-cares to optimize the node [Mishchenko et al. 2005].

The benchmarks used were 20 large public and 15 industrial benchmarks. The public ones are from the MCNC and ISCAS'89 suites used in previous work on FPGA mapping [Chen et al. 2004; Mishchenko et al. 2007a].[1] The results for these benchmarks are shown in Table II.[2] The results for the 15 industrial benchmarks are shown in Table III.

Tables II and III list results at the end of five different runs.

—Section "Baseline" corresponds to a typical run of tech-independent synthesis [Mishchenko et al. 2006a] followed by technology mapping. The other flows are compared to these results. To make the comparison more fair, the commands (*dc2 –l; dc2 –l; if –C 12*) are run four times to ensure that a strong optimization was obtained. All subsequent flows start the baseline results.
—Section "Choices" corresponds to four iterations of mapping with structural choices (*st; dch; if –C 12*) [Mishchenko et al. 2007a] and picking the best result after any iteration.
—Section "Mfs" corresponds to four iterations of technology mapping with structural choices, interleaved with the proposed resub-based resynthesis (*st; dch; if –C 12; mfs –W 4 –M 5000*) and picking the best result after any iteration.
—Section "MfsR" corresponds to four iterations of technology mapping with structural choices, interleaved with the proposed don't-care-based node optimization (*st; dch; if –C 12; mfs –r; st; if –C 12*) and picking the best result after any iteration. The command *mfs –r* uses the don't-cares created from the window around the pivot node

[1]Circuit s298 was replaced by i10 because it contains only 24 6-LUTs.
[2]A similar version of these experimental results appeared in Mishchenko et al. [2008] to show the orthogonal nature of the optimization method reported in that paper.

Table II. The Results of Resynthesis after Technology Mapping (K = 6) for Academic Benchmarks

Name	Statistics			Baseline			Choices			Mfs			MfsR			Mfs+MfsR		
	PI	PO	FF	LUT	LV	T	LUT	LV	T	LUT	LV	T	LUT	LV	T	LUT	LV	T
alu4	14	8	0	810	5	1.28	773	5	1.86	506	5	10.34	514	5	10.29	433	5	18.79
apex2	39	3	0	864	6	1.42	867	6	4.26	640	6	22.39	741	6	21.29	650	6	43.63
apex4	9	19	0	767	5	1.16	816	5	1.83	776	5	11.36	784	5	12.7	791	5	23.61
bigkey	263	197	224	734	3	2.07	677	3	2.26	499	3	3.38	495	3	4.15	485	3	4.51
clma	383	82	33	2830	10	5.51	2749	9	11.54	856	7	97.44	1079	7	48.61	609	7	118.07
des	256	245	0	916	4	1.99	865	4	1.91	559	4	6.47	645	4	7.46	558	4	11.54
diffeq	64	39	377	684	7	0.91	649	7	0.17	645	7	0.64	649	7	0.17	648	7	6.55
dsip	228	197	224	680	3	1.33	681	3	0.19	681	3	0.18	681	3	0.18	681	3	0.18
elliptic	131	114	1122	1870	10	2.56	1767	10	0.48	1765	10	2.82	1754	10	8.33	1767	10	0.47
ex1010	10	10	0	2605	6	4.63	2748	6	13.71	1649	6	82.57	1375	6	40.66	1250	6	97.57
ex5p	8	63	0	533	5	0.99	551	4	1.30	131	3	4.86	126	3	1.97	105	3	4.30
frisc	20	116	886	1750	12	3.45	1759	11	5.94	1726	11	22.34	1725	10	16.2	1717	10	30.23
i10	257	224	0	580	9	1.14	582	8	1.33	530	8	5.08	558	8	1.73	558	8	11.64
misex3	14	14	0	701	5	1.21	661	5	1.90	372	5	9.24	507	4	7.62	367	5	14.49
pdc	16	40	0	2182	6	4.72	2134	6	13.89	140	5	16.62	185	5	10.24	166	4	22.06
s38417	28	106	1636	2257	6	3.77	2273	6	0.67	2202	6	12.71	2230	6	9.60	2203	6	35.75
s38584	39	304	1260	2156	6	4.13	2135	6	4.06	2074	5	11.23	2055	5	12.78	2044	5	18.1
seq	41	35	0	755	5	1.38	765	5	3.22	565	5	11.62	657	5	10.54	548	5	19.0
spla	16	46	0	1449	6	3.22	1384	6	8.5	170	4	17.11	162	4	5.58	135	4	15.53
tseng	53	123	385	708	7	0.79	650	7	0.74	651	6	2.10	650	6	3.27	649	6	5.26
GMean				1103	5.92	1.98	1084	5.76	2.08	640	5.36	7.83	675	5.27	6.06	607	5.27	11.88
Ratio1				1	1	1	0.982	0.974	1.052	0.581	0.905	3.962	0.612	0.891	3.063	0.551	0.891	6.007
Ratio2							1	1	1	0.591	0.929	3.767	0.623	0.915	2.912	0.561	0.915	5.711
Ratio3										1	1	1	1.055	0.984	0.773	0.948	0.984	1.516
Ratio4													1	1	1	0.899	1.000	1.961
The following results exclude the five outlier examples																		
GMean				964.44	5.76	1.67	942.05	5.69	1.35	793.39	5.56	5.28	838.0	5.44	4.86	784.93	5.52	8.93
Ratio1				1	1	1	0.977	0.986	0.811	0.823	0.965	3.161	0.869	0.944	2.911	0.814	0.958	5.352
Ratio2							1	1	1	0.842	0.978	3.896	0.890	0.957	3.588	0.833	0.972	6.597
Ratio3										1	1	1	1.056	0.979	0.921	0.989	0.994	1.693
Ratio4													1	1	1	0.937	1.015	1.839

Table III. The Results of Resynthesis after Technology Mapping (K = 6) for Industrial Benchmarks

Name	Statistics			Baseline			Choices			Mfs			MfsR			Mfs+MfsR		
	PI	PO	FF	LUT	LV	T	LUT	LV	T	LUT	LV	T	LUT	LV	T	LUT	LV	T
D1	30	32	985	1661	6	3.1	1566	5	4.4	1476	5	10.4	1456	5	10.3	1455	5	15.7
D2	99	204	1186	2518	5	4.6	2356	5	5.1	2190	5	14.2	2152	5	14.4	2122	5	21.3
D3	67	90	1350	3394	16	7.5	3231	16	12.3	2973	14	74.2	3051	13	38.8	2940	13	84.4
D4	129	357	1455	3453	10	6.6	3393	8	8.8	3084	7	27.7	3123	7	26.7	3071	7	47.8
D5	112	480	3439	11135	9	22.9	11301	8	48.6	10728	8	198.8	10871	8	150.0	10621	8	307.6
D6	1071	1069	5867	11797	8	26.8	11439	8	34.8	10877	7	99.7	10802	7	104.8	10632	7	167.0
D7	1134	3965	8371	15327	12	35.7	14572	12	51.8	13987	11	90.8	13803	12	86.1	13708	12	120.8
D8	210	299	6662	16719	12	43.9	15975	11	66.0	14822	10	208.3	14870	10	157.8	14504	10	301.7
D9	1069	747	8689	22049	6	52.0	20871	5	80.9	19135	5	192.7	18922	5	158.9	18768	5	339.6
D10	1425	4991	26705	44393	11	97.0	41910	10	150.0	39161	9	307.2	40138	7	189.2	39640	7	424.0
D11	781	5563	16205	51569	10	141.9	49506	10	162.8	46258	9	881.1	46235	9	742.9	45288	9	1752.7
D12	2561	9586	23612	52462	9	126.0	49266	9	160.0	46695	9	293.3	46252	9	297.3	45945	9	430.3
D13	4725	16657	43309	74723	12	172.8	70827	11	276.9	66515	10	563.5	66175	10	470.1	65212	10	743.0
D14	2418	6000	34834	78359	20	236.5	72487	19	337.5	68073	17	945.8	67923	17	805.4	67235	17	1397.7
D15	8879	52334	41521	126757	11	223.2	122863	11	325.5	117056	11	622.8	116821	8	591.7	116174	8	890.0
GMean				17142	9.84	38.40	16379	9.19	55.42	15344	8.59	152.9	15341	8.28	129.6	15120	8.28	229.3
Ratio1				1	1	1	0.955	0.934	1.443	0.895	0.874	3.983	0.895	0.84	3.377	0.882	0.842	5.971
Ratio2							1	1	1	0.937	0.935	2.760	0.937	0.901	2.340	0.923	0.901	4.138
Ratio3										1	1	1	1.000	0.964	0.848	0.985	0.964	1.499
Ratio4													1	1	1	0.986	1.000	1.768

to perform bidecomposition for each mapped LUT. After running *mfs –r* the internal AIG size has been reduced. To observe the improvement, we need to run technology mapping (*if* –C 12) one more time.

—Section "Mfs+MfsR" corresponds to four iterations of technology mapping with structural choices, interleaved with the proposed don't-care-based node optimization and don't-care-based resubstitution (*st; dch; if –C 12; mfs –r; st; if –C 12; mfs –W 4 –M 5000*) and picking the best result after any iteration.

The tables contain the number of primary inputs (column "PIs"), primary outputs (column "POs"), registers (column "Reg"), area calculated as the number of 6-LUTs (columns "LUT"), the depth of the 6-LUT network (columns "LV"), and the total runtime of the specific flow in seconds (columns "T"). The ratios in the tables are the geometric averages of the corresponding ratios reported in the columns.

The results listed in Tables II and III show that, compared to the baseline synthesis and mapping, the iterative mapping with structural choices reduces area and depth by 1.8% and 2.6% (for academic benchmarks) and by 4.5% and 6.6% (for industrial benchmarks). The improvement is likely due to repeated recomputation of structural choices by applying logic synthesis to the previously mapped network. When the resulting subject graph with choices is mapped again, those structures tend to be selected that offer an improvement, compared to the previous mapping. Several iterations of this evolutionary process find good structures for the selected LUT size and delay constraints.

Table II and III lead to the following observations.

—When the proposed don't-care-based resynthesis ("*mfs*") is included in the iteration, the area and depth are additionally reduced by 40.9% and 7.1% (for academic benchmarks) and by 6.7% and 6.5% (for industrial benchmarks).

—Without the five outlier circuits discussed shortly, the additional reduction for academic benchmarks is 15.8% in area and 2.2% in depth. The results excluding the outliers are shown at the bottom of Table II. The results demonstrate that iterating optimization with choices and "*mfs*" leads to a more substantial improvement than optimization with choices only (column "Choices"). This improvement is likely because "*mfs*" allows for a deeper restructuring of the subject graph that is particularly favorable for area minimization.

—When the proposed don't-care-based node optimization ("*mfs –r*") is included in the iteration, the area and depth are additionally reduced by 37.7% and 8.5% (for academic benchmarks) and by 6.3% and 9.1% (for industrial benchmarks). Without the five outlier circuits discussed shortly, the additional reduction for academic benchmarks is 11.0% in area and 4.3% in depth.

—When both methods, "*mfs*" and "*mfs –r*" are used in the iteration, the area and depth are additionally reduced by 43.8% and 8.5% (for academic benchmarks) and by 7.7% and 9.9% (for industrial benchmarks). Without the five outlier circuits discussed shortly, the additional reduction for academic benchmarks is 16.7% in area and 2.8% in depth. This means that the combining of the two methods can outperform either "*mfs*" or "*mfs –r*" alone.

It was observed that 5 "outlier" circuits in Table II (*clma, ex1010, ex5p, pdc, spla*) were reduced more substantially than other circuits in the set. These are the circuits originating from PLA descriptions. It is likely that the logic synthesis tool used to generate multilevel representations of these circuits did a poor job of extracting shared logic among the outputs of the PLA. This resulted in highly suboptimal logic structures, which introduced heavy structural bias into technology mapping. It is interesting to

note that mapping with structural choices produced much smaller improvements than the proposed resubstitution and/or node optimization of this article. This is because the tech-independent synthesis and mapping with choices rely on local transforms and so they are still subject to structural bias, albeit less so than mapping without structural choices.

Applying resynthesis to these outlier benchmarks as part of the iterative synthesis and mapping led to dramatic improvements in both area and delay (about 5x in area and 20% in delay). This surprising result was thoroughly verified and proved correct. It shows that the proposed SAT-based resubstitution can cope with a substantial structural bias and gradually derive a new logic structure that is more suitable for 6-LUT mapping.

It was noted that some of the academic benchmarks in Table II have don't-cares in their original PLA descriptions. It is not known whether the don't-cares were used during multilevel synthesis that produces the circuits. In any case, they were not available during mapping and resynthesis, which worked on multilevel circuits without don't-cares generated by another tool.

Typically, reducing logic depth could be achieved by collapsing a LUT into its fanout. As a result, the average fanout will be increased which is detrimental to placement and routing due to congestion. In this section, we further investigate this. Table IV gives additional details on the quality of the optimization for the same 20 academic benchmarks. Reported are the edge count (Columns "EC"), the average fanin (Columns "AFI"), and the average fanout (Columns "AFO").

Table IV leads to several observations: When the proposed don't-care-based resynthesis ("Mfs") is included in the iteration, the edge count and average fanout are reduced by 41.0% and 0.6%. The overall conclusion is that the LUT count reduction does not cause edge count and fanout penalties.

The place-and-route experiment reported in Table V compares the results shown in Tables II produced by VPR [Betz et al. 1999] for 20 academic benchmarks. The following steps were performed: clock signal was added for all sequential circuits, the 6-LUT architecture was created using the 4-LUT architecture file, the CLB is assumed to contain one 6-LUT, and the T-VPack tool was used to preprocess all the netlists. VPR was run to find the minimum routing channel width using 10 different random seeks. The values of critical path (in nanoseconds), total wire-length, and minimum channel width reported in Table V as columns "CP", "TWL", and "MCW", respectively, are geometric averages over the 10 different runs for each benchmark.

Table V leads to the following observations. When comparing "Mfs+MfsR" against "Mfs", the former is 2.0%, 6.2%, and 0.003% better in terms of critical delay, total wire-length, and minimum channel width, respectively.

7. CONCLUSIONS AND FUTURE WORK

The article proposes an integrated SAT-based logic optimization methodology useful as part of tech-independent synthesis and as a new postmapping resynthesis. The algorithms used in the integrated solution were selected based on their scalability and efficient implementation. They include improved algorithms for structural analysis (windowing), simulation, and new ways of exploiting don't-cares.

A SAT solver was used to perform all aspects of Boolean functions manipulation during resynthesis. In particular, it was shown how an optimized implementation of a node can be computed directly using interpolation, without first explicitly computing a don't-care set and then minimizing the logic function with this don't-care.

Future work will include fine-tuning resynthesis to focus on delay, power, placement, and other cost functions.

Table IV. Detailed Statistics of the Results of Resynthesis after Technology Mapping (K = 6) for Academic Benchmarks

Name	Baseline			Choices			Mfs			MfsR			Mfs+MfsR		
	EC	AFI	AFO	EC	AFI	AFO	EC	AFI	AFO	EC	AFI	AFO	EC	AFI	AFO
alu4	3728	2.22	4.60	3620	1.88	4.66	2324	1.95	4.62	2442	2.06	4.55	2070	1.97	4.52
apex2	3934	2.72	4.55	3927	2.74	4.57	2951	2.84	4.65	3566	2.71	4.55	3163	2.62	4.68
apex4	3642	3.63	4.75	3795	3.66	4.67	3640	3.54	4.72	3837	3.55	4.68	3821	3.55	4.74
bigkey	3927	1.95	4.45	3363	1.94	4.14	2595	1.73	4.19	2463	1.70	4.00	2534	1.70	3.86
clma	12364	3.28	4.37	12265	3.38	4.46	3549	2.95	4.43	4810	3.05	4.43	2607	2.84	4.28
des	4535	2.89	4.86	4170	2.77	4.75	2605	2.68	4.68	3108	2.56	4.69	2648	2.47	4.54
diffeq	3126	2.13	4.55	3008	2.07	4.57	2985	2.18	4.63	3008	2.18	4.63	2897	2.26	4.40
dsip	3182	2.52	3.87	3184	2.52	3.87	3184	2.02	3.87	3184	2.02	3.87	3184	2.02	3.87
elliptic	8295	2.16	4.44	7951	2.16	4.44	7936	2.28	4.50	8020	2.22	4.32	7951	2.36	4.44
ex1010	12425	4.27	4.77	13282	4.10	4.79	7550	3.74	4.89	6908	3.82	4.90	6323	3.75	4.93
ex5p	2508	3.38	4.67	2526	3.03	4.63	540	2.15	4.25	579	2.19	4.61	459	2.10	4.34
frisc	7729	2.97	4.41	7758	2.96	4.46	7634	2.96	4.45	8229	2.96	4.48	8306	2.99	4.51
i10	2491	2.78	4.23	2457	2.81	4.20	2277	2.78	4.25	2407	2.78	4.26	2419	2.82	4.24
misex3	3298	2.40	4.70	3079	2.56	4.64	1715	2.37	4.67	2460	1.92	4.78	1780	2.45	4.72
pdc	10033	3.66	4.60	9922	3.64	4.70	618	2.50	4.52	877	2.64	4.56	795	2.82	4.71
s38417	10315	2.03	3.84	10228	2.13	3.91	10057	2.07	3.91	10190	2.10	3.85	10038	2.09	3.89
s38584	9628	1.77	4.10	9500	1.80	4.15	9218	1.71	4.09	9245	1.71	4.14	9212	1.71	4.13
seq	3457	2.63	4.58	3531	2.74	4.58	2618	2.68	4.68	3156	2.68	4.60	2627	2.72	4.72
spla	6724	3.76	4.64	6524	3.75	4.71	716	2.75	4.40	759	2.57	4.44	596	2.78	4.37
tseng	2633	2.14	3.67	2397	2.27	3.71	2411	2.32	3.70	2563	2.09	3.75	2444	2.09	3.70
GMean	5052	2.68	4.42	4937	2.67	4.42	2868	2.46	4.39	3152	2.42	4.39	2809	2.45	4.37
Ratio1	1	1	1	0.977	0.994	1.000	0.568	0.915	0.994	0.624	0.901	0.994	0.556	0.913	0.988
Ratio2				1	1	1	0.581	0.921	0.994	0.638	0.906	0.994	0.569	0.919	0.988
Ratio3							1	1	1	1.099	0.984	1.000	0.979	0.998	0.994
Ratio4										1	1	1	0.891	1.014	0.994

Table V. The Results of Place-and-Route for Academic Circuits Using VPR

Name	Baseline			Choices			Mfs			MfsR			Mfs+MfsR		
	CP	WL	MCW	CP	WL	MCW	CP	WL	MCW	CP	WL	MCW	CP	WL	MCW
alu4	118.09	14295	11.79	121.30	13543	11.69	97.02	8060	10.29	101.98	8220	10.10	86.43	6042	9.10
apex2	117.17	18037	13.49	115.10	17187	12.89	104.44	12292	12.10	110.09	15255	13.00	103.69	13394	13.00
apex4	147.87	17633	15.20	155.48	18535	14.90	146.13	17557	14.90	150.05	17526	14.59	150.86	18595	15.00
bigkey	227.62	11379	5.89	199.57	9380	5.89	172.17	6808	5.00	179.52	7419	5.00	191.96	6943	5.00
clma	218.96	53573	13.29	213.91	52958	13.00	108.80	13694	11.19	121.42	18312	11.90	69.81	5595	9.29
des	222.68	20097	8.08	201.22	18900	7.48	202.88	13925	6.99	246.71	14290	7.29	224.59	13699	6.99
diffeq	88.53	7144	7.58	79.39	6785	7.38	85.83	6763	7.38	79.39	6785	7.38	93.40	6856	7.58
dsip	168.87	8986	6.00	213.64	9099	5.19	213.64	9099	5.19	213.64	9099	5.19	213.64	9099	5.19
elliptic	212.60	24038	9.59	204.02	23154	9.29	219.11	22944	9.19	200.86	23897	9.59	204.61	22544	9.59
ex1010	308.17	82372	20.20	323.36	83408	19.39	237.34	43896	17.00	210.72	36185	16.29	204.14	33903	16.99
ex5p	116.27	10245	12.19	108.63	10177	12.00	42.44	1391	6.00	42.34	1446	6.09	35.98	1066	5.38
frisc	223.82	29498	11.00	220.47	28978	11.00	210.82	29094	11.00	210.60	30088	11.29	210.06	30269	11.49
i10	206.86	15459	7.07	188.10	15041	7.48	190.57	15049	7.29	197.11	14519	7.27	216.01	15213	7.57
misex3	110.52	12452	11.39	109.31	11220	11.00	79.15	5718	9.79	88.97	7437	9.19	81.68	5975	9.90
pdc	262.15	59184	18.20	267.90	55299	17.10	66.42	1617	7.00	78.66	2356	7.58	68.34	1821	7.00
s38417	79.90	27291	7.09	79.56	27163	7.09	77.88	27005	7.00	83.17	27144	7.48	78.64	26678	7.19
s38584	120.16	24764	8.19	139.53	24407	8.00	117.20	23716	7.89	114.45	23631	7.89	113.09	23440	8.00
seq	119.51	15817	13.59	120.95	15657	13.10	106.53	10751	12.10	112.94	13195	12.90	103.78	11019	12.59
spla	204.36	31559	14.10	192.14	29684	14.20	49.46	2052	7.00	45.67	1867	7.00	48.86	1860	7.00
tseng	93.96	6142	6.38	85.47	5542	6.00	84.55	5664	6.00	85.93	5973	6.58	91.67	5661	6.38
GMean	156.37	19283	10.32	154.58	18527	10.02	116.03	9772	8.51	119.24	10443	8.69	113.58	9089	8.50
Ratio1	1	1	1	0.989	0.961	0.971	0.742	0.507	0.825	0.763	0.542	0.842	0.726	0.471	0.823
Ratio2				1	1	1	0.751	0.527	0.850	0.771	0.564	0.867	0.735	0.491	0.848
Ratio3							1	1	1	1.028	1.069	1.021	0.979	0.930	0.998
Ratio4										1	1	1	0.953	0.870	0.978
The following results exclude the five outlier examples															
GMean	141.46	15315	9.05	140.08	14651	8.78	130.52	12315	8.39	134.65	13085	8.56	133.31	12274	8.53
Ratio1	1	1	1	0.990	0.957	0.970	0.923	0.804	0.927	0.952	0.854	0.946	0.942	0.801	0.943
Ratio2				1	1	1	0.932	0.841	0.955	0.961	0.893	0.975	0.952	0.838	0.972
Ratio3							1	1	1	1.032	1.063	1.020	1.021	0.997	1.017
Ratio4										1	1	1	0.990	0.938	0.997

ACKNOWLEDGMENTS

We acknowledge the help of Zile Wei in running the place-and-route experiments in Table V of this article. We thank the anonymous reviewers for their helpful comments.

REFERENCES

ALTERA CORP. 2011. Altera Stratix IV FPGA family overview. http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/overview/stxiv-overview.html.

BERKELEY VERIFICATION AND SYNTHESIS RESEARCH GROUP (BVSRG). 2011. ABC: A system for sequential synthesis and verification, release 80802. http://www.eecs.berkeley.edu/~alanmi/abc/.

BETZ, V., ROSE, J., AND MARQUARDT, A. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers.

BJESSE, P. AND CLAESSEN, K. 2000. SAT-Based verification without state space traversal. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*. Lecture Notes in Computer Science, vol. 1954, Springer, 372–389.

CASE, M. L., MISHCHENKO, A., AND BRAYTON, R. K. 2006. Inductively finding a reachable state space over-approximation. In *Proceedings of the International Workshop on Logic and Synthesis (IWLS'06)*. 172–179. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_inv.pdf.

CASE, M. L., MISHCHENKO, A., AND BRAYTON, R. K. 2008. Cut-Based inductive invariant computation. In *Proceedings of the International Workshop on Logic and Synthesis (IWLS'08)*. 253–258. http://www.eecs.berkeley.edu/~alanmi/publications/2008/iwls08_ind.pdf.

CHANG, K.-H., MARKOV, I. L., AND BERTACCO, V. 2007. Fixing design errors with counterexamples and resynthesis. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'07)*. 944–949.

CHEN, K.-C. AND CONG, J. 1992. Maximal reduction of lookup-table-based FPGAs. In *Proceedings of the Design Automation and Test in Europe Conference (DATE'92)*. 224–229.

CHEN, D. AND CONG, J. 2004. DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'04)*. 752–757.

CONG, J., LIN, Y., AND LONG, W. 2002. SPFD-Based global rewiring. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'02)*. 77–84.

EEN, E. AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *Proceedings of the SAT'03 Conference*. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/.

GOLDBERG, E. AND NOVIKOV, Y. 2003. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Design Automation and Test in Europe Conference (DATE'03)*. 886–891.

KRAVETS, V. N. AND KUDVA, P. 2004. Implicit enumeration of structural changes in circuit Optimization. In *Proceedings of the Design Automation Conference (DAC'04)*. 438–441.

LEE, C.-C., JIANG, J.-H. R., HUANG, C.-Y., AND MISHCHENKO, A. 2007. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'07)*. 227–233. http://www.eecs.berkeley.edu/~alanmi/publications/2007/iccad07_fd.pdf.

MCMILLAN, K. L. 2003. Interpolation and SAT-based model checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'03)*. Lecture Notes in Computer Science, vol. 2725, Springer, 1–13.

MCMILLAN, K. 2005. Don't-Care computation using *k*-clause approximation. In *Proceedings of the International Workshop on Logic and Synthesis (IWLS'05)*. 153–160.

MISHCHENKO, A., STEINBACH, B., AND PERKOWSKI, M. A. 2001. An algorithm for bi-decomposition of logic functions. In *Proceedings of the Design Automation Conference (DAC'01)*. 103–108.

MISHCHENKO, A. AND BRAYTON, R. 2005. SAT-Based complete don't-care computation for network optimization. In *Proceedings of the Design Automation and Test in Europe Conference (DATE'05)*. 418–423. http://www.eecs.berkeley.edu/~alanmi/publications/2005/ date05_satdc.pdf.

MISHCHENKO, A. AND BRAYTON, R. 2006. Scalable logic synthesis using a simple circuit structure. In *Proceedings of the International Workshop on Logic and Synthesis (IWLS'06)*. 15–22. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_sls.pdf.

MISHCHENKO, A., CHATTERJEE, S., AND BRAYTON, R. 2006a. DAG-Aware AIG rewriting: A fresh look at combinational logic synthesis. In *Proceedings of the Design Automation Conference (DAC'06)*. 532–536. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf.

MISHCHENKO, A., ZHANG, J. S., SINHA, S., BURCH, J. R., BRAYTON, R., AND CHRZANOWSKA-JESKE, M. 2006b. Using simulation and satisfiability to compute flexibilities in Boolean networks. *IEEE Trans. Comput. Aid. Des. 25*, 5, 743–755. http://www.eecs.berkeley.edu/~alanmi/publications/2005/tcad05_s&s.pdf.

MISHCHENKO, A., CHATTERJEE, S., BRAYTON, R., AND EEN, E. 2006c. Improvements to combinational equivalence checking. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'06)*. 836–843. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_cec.pdf.

MISHCHENKO, A., CHATTERJEE, S., AND BRAYTON, R. 2007a. Improvements to technology mapping for LUT-based FPGAs. *IEEE Trans. Comput. Aid. Des. 26*, 2, 240–253. http://www.eecs.berkeley.edu/~alanmi/publications/2006/tcad06_map.pdf.

MISHCHENKO, A., CHO, S., CHATTERJEE, S., AND BRAYTON, R. 2007b. Combinational and sequential mapping with priority cuts. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'07)*. http://www.eecs.berkeley.edu/~alanmi/publications/2007/iccad07_map.pdf.

MISHCHENKO, A., BRAYTON, R., AND CHATTERJEE, S. 2008a. Boolean factoring and decomposition of logic networks. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'08)*. 38–44. http://www.eecs.berkeley.edu/~alanmi/publications/2008/iccad08_lp.pdf.

MISHCHENKO, A., BRAYTON, R., JIANG, J.-H. R., AND JANG, S. 2009. Scalable don't care based logic optimization and resynthesis. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'09)*. 151–160. http://www.eecs.berkeley.edu/~alanmi/ publications/2009/fpga09_mfs.pdf.

MISHCHENKO, A., CASE, M. L., BRAYTON, R., AND JANG, S. 2008b. Scalable and scalably-verifiable sequential synthesis. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'08)*. 234–241.

MUROGA, S., KAMBAYASHI, Y., LAI, H. C., AND CULLINEY, J. N. 1989. The transduction method- Design of logic networks based on permissible functions. *IEEE Trans. Comput. 38*, 10, 1404–1424.

PAN, P. AND LIN, C.-C. 1998. A new retiming-based technology mapping algorithm for LUT- based FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'98)*. 35–42.

PLAZA, S., CHANG, K.-H., MARKOV, I. L., AND BERTACCO, V. 2007. Node mergers in the presence of don't cares. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'07)*. 414–419.

SALUJA, N. AND KHATRI, S. P. 2004. A robust algorithm for approximate compatible observability don't care (CODC) computation. In *Proceedings of the Design Automation Conference (DAC'04)*. 422–427.

SAVOJ, H. 1992. Don't cares in multi-level network optimization. Ph.D. dissertation, University of California, Berkeley.

SENTOVICH, E., SINGH, K. J., LAVAGNO, L., MOON, C., MURGAI, R., ET AL. 1992. SIS: A system for sequential circuit synthesis. Tech. rep. UCB/ERI, M92/41, ERL, Department of EECS, University of California, Berkeley.

XILINX CORP. 2011. Xilinx Virtex-5 product table. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/v5product_table.pdf.

YANG, Y.-S., SINHA, S., VENERIS, A., AND BRAYTON, R. 2007. Automating logic rectification by approximate SPFDs. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'07)*.

ZHU, Q., KITCHEN, N., KUEHLMANN, A., AND SANGIOVANNI-VINCENTELLI, A. L. 2006. SAT sweeping with local observability don't-cares. In *Proceedings of the Design Automation Conference (DAC'06)*. 229–234.

Received May 2009; revised August 2009; accepted November 2010