SAT-Based Logic Optimization and Resynthesis

Alan Mishchenko **Robert Brayton**
Department of EECS
University of California, Berkeley
{alanmi, brayton}@eecs.berkeley.edu

Jie-Hong Roland Jiang
Department of EE
National Taiwan University
jhjiang@cc.ee.ntu.edu.tw

Stephen Jang
Xilinx Inc.
San Jose, CA
sjang@xilinx.com

Abstract

The paper develops a technology-independent optimization and post-mapping resynthesis for combinational logic networks, with emphasis on scalability and optimizing power. 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 large industrial designs. The approach is based on several heterogeneous algorithms, which include structural analysis, random and constrained simulation, and manipulation of Boolean functions using a SAT solver. Structural methods include improved windowing, which focuses on reconvergent logic structures rich in functional flexibilities. It is shown how a mainstream SAT solver can be minimally modified by combining it with an interpolation package, which computes Boolean functions of nodes after resynthesis as a by-product of completed feasibility proofs. Experimental results focusing on the minimization of the number of 6-LUTs after high-effort iterative FPGA mapping with structural choices, demonstrate that the proposed resynthesis, applied to 15 benchmarks reduced area by 6.0% and delay by 2.3% on average. For 5 benchmarks derived from PLA descriptions, the reduction of 5x in area and 20% in depth was obtained, which speaks for the powerful nature of Boolean optimization employed in the proposed resynthesis.

1 Introduction

Technology-independent optimization and post-mapping resynthesis of Boolean networks using internal flexibilities have long histories [22][25][10][15], to mention a few publications. Traditional don't-care-based optimization [25] is part of the high-effort logic optimization flow in SIS [26]. 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 [15][24].

Both traditional and the newer algorithms for don't-care-based optimization compute don't-cares before using them. This may be one explanation for long runtimes of these algorithms when applied to large industrial designs, even if windowing is used. A notable exception is the SAT-based approach [13], which optimized nodes "in-place", without explicitly computing don't-cares. However, unlike [25][10], that work does not allow for resubstitution. As a result, the optimization space is limited to the current node boundaries. Another recent method [11] performs efficient SAT-based resubstitution but does not consider don't-cares, which may limit its optimization potential.

Some recent papers [28][23] propose optimization for And-Inverter Graphs (AIGs) using the notion of equivalence under don't-cares. These approaches are not applicable to post-mapping resynthesis. They are also limited because they can optimize an AIG node only if there is another AIG node with a similar logic function that can replace the given node.

It should be noted that some approaches to resynthesis [4] achieve sizeable reduction of the network without exploiting don't-cares, by pre-computing 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.

The *main contribution of the present paper* is an improved SAT-based resubstitution, which takes into account internal don't cares, without explicitly computing them. This algorithm is well adapted to the resynthesis and rewiring for FPGAs. When applied to a large window, the challenge is to compute a new lookup-table function after rewiring. Traditionally, this is done using BDDs [25][10], SPFDs [5], or by enumerating all satisfiable assignments of a SAT problem [18]. The present paper follows [11] and uses interpolation [12] to derive a new function as a by-product of an unsatisfiable run of the SAT solver. We discuss implementation details of this approach.

Another contribution of paper is an improved windowing algorithm, which has two distinctive aspects: (a) better structural analysis that skips non-reconvergent paths, thereby generating windows with more internal flexibilities, and (b) a more reliable window computation, which works robustly for large networks containing nodes with multiple fanouts.

The proposed algorithm was implemented in ABC [2] and tested on industrial benchmarks. Although current experiments have been limited to resynthesis of logic networks after FPGA mapping, the algorithms are applicable to a wider range of challenging problems:

- technology-independent optimization for logic networks to minimize the number of FF literals,
- technology-independent optimization for AIGs to minimize the number of AIG nodes (and record structural choices),
- technology-dependent resynthesis for FPGAs and standard-cells to improve area, delay, power, the number of edges, etc,
- timing-driven resynthesis and rewiring after placement.

These applications are waiting to be explored.

The rest of the paper is organized as follows. Section 2 describes some background. Section 3 discusses optimization based on windowing, simulation, SAT solving, and interpolation. Section 4 reports experimental results. Section 5 concludes the paper 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 in this paper.

A node has zero or more *fanins,* i.e. nodes that are driving this node, and zero or more *fanouts*, i.e. nodes driven by this node. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, it contains registers whose inputs and output are treated as additional PIs/POs. It is assumed that each node has a unique integer called its *node ID.*

The *level* of a node is the length of the longest path from any PI to the node. The node itself is counted towards the path lengths but the PIs are not. The network *depth* is the largest level of an internal node. The *area* and *edge count* of a network are the node count and the sum total of fanin counts of all nodes.

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 described in this paper are applicable to both AIGs and general-case logic networks.

2.2 Cuts and cones

A *cut C* of node *n*, called *root*, is a set of nodes of the network, called *leaves*, such that each path from a PI to *n* passes through at least one leaf. A *trivial cut* of node *n* is the cut {*n*} composed of the node itself. A non-trivial cut *covers* all the nodes found on the paths from the root to the leaves, including the root and excluding the leaves. A trivial cut does not cover any nodes. A cut is *K-feasible* if the number of nodes in it does not exceed *K*. A cut is said to be *dominated* if there is another cut of the same node, which is contained, set-theoretically, in the given cut.

A *fanin* (*fanout*) *cone* of node *n* is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node. A *maximum fanout free cone* (MFFC) of node *n* is a subset of the fanin cone, such that every path from a node in the subset to the POs passes through *n*. Informally, the MFFC of a node contains all the logic used exclusively by the node. When a node is removed or replaced, the logic in its MFFC can be removed.

2.3 Don't-cares and resubstitution

Internal flexibilities of a node in the Boolean network arise because of limited controllability and observability of the node. Controllability of the node may be restricted because some combinations of values are never produced by the fanins. Observability may be limited because the fanout logic blocks the node's effect on the POs under some combinations of the PI values. Examples can be found in [15].

These internal flexibilities result in *don't-cares* at the node. These can be represented by a single 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 $C(x)$ of a node is a Boolean function of the PIs derived as follows:

$$C(x) = \prod_i [z_i(x) \oplus z_i'(x)],$$

where $z_i'(x)$ are the PO functions in a network isomorphic to the original one, except that the given node is complemented [15].

Traditionally, some types, or subsets, of don't-cares are derived and used to optimize the node [25][15]. This optimization may involve minimizing the node's function in isolation from other nodes, and/or an attempt to express the node in terms of the available nodes. 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 resubstitution with these candidates exists, that is, the node can be re-expressed using the new fanins without changing the functionality of the network.

A necessary and sufficient *condition of resubstitution* is given in [18] (Theorem 5.1): Functions $g_i(x)$ can resubstitute function $f(x)$ 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 i. Informally, resubstitution exists if and only if the capability of functions $g_i(x)$ to distinguish minterms is no less than that of function $f(x)$. In the presence of don't-cares, the property is restricted to hold only on the care set, $C(x_1) \wedge C(x_2)$.

2.4 Optimization with don't-cares

Computation of don't-cares involves exploring the structural neighbors of the node in the network. If the network is large, as in most present-day designs, exploration of the whole network for the benefit of each node is infeasible. Therefore computation is limited to a pocket of logic surrounding the node, called a *window*. The node is called the *pivot* of the window. The scope of the window is controlled by user-specified parameters, such as the number of levels spanned by the window as well as the number of its PIs, POs, and internal nodes.

When a don't-care is computed for a node, all other nodes are assumed to be fixed. The computed don't-care should be used immediately to optimize the node. The network is updated before optimization moves on to the other node. This avoids don't-care compatibility issues, which may arise when don't-cares are computed for several nodes before they are used.

There are two approaches to don't-care-based optimization:

- A traditional approach based on computing a full set, or a subset, of don't-cares and using it to optimize the function of the node (not discussed in this paper).
- A new approach when a don't-care is not explicitly derived, but internal flexibilities are exploited to find an optimized function of the node directly (discussed in Section 3).

2.5 Interpolation

Consider a pair of Boolean functions, $A(x, y)$ and $B(y, z)$, such that $A(x, y) \wedge B(y, z) = 0$, where x and z are variables appearing only in the clauses of A and of B, respectively, and y are variables common to A and B. An *interpolant* of function $A(x, y)$ w.r.t. function $B(y, z)$ is a Boolean function, $I(y)$, depending only on the common variables y, such that $A(x, y) \Rightarrow I(y)$ and $I(y) \Rightarrow \overline{B}(y, z)$.

Consider an unsatisfiable SAT instance composed of two sets of clauses A and B. In this case, $A(x,y) \wedge B(y,z) = 0$. An interpolant of A can be computed from the proof of unsatisfiability of the SAT instance using the algorithm found in [12] (Definition 2). $A(x,y)$ can be interpreted as the onset of a function, $B(y,z)$ as the offset, and the complement of $A(x,y) + B(y,z)$ 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 have been used in a particular way.

2.6 Structural choices

Don't-care-based optimization of AIGs is different from that applied to a logic network whose nodes typically have several fanins and non-trivial logic functions. To enable efficient AIG

optimization, several levels of AIG nodes should be collapsed into a larger node, for which don't-cares are computed. After optimization using observability don't-cares, the global function of an AIG node can change. As a result, the two nodes (before and after optimization) are not equivalent and hence the pair cannot be used as a structural choice of the original node.

On the other hand, recording structural choices, after all intermediate transformations, can be useful for scalable verification and for improving the quality of technology mapping. For example, using several snapshots of the same AIG obtained by different optimizations, can improve both area and delay of technology mapping for FPGAs, especially if AIG minimization and mapping are interleaved and iterated (see Section 4).

The key observation enabling recording structural choices after AIG minimization with don't-cares is that the window POs preserve their global functionality. Thus, after optimizing an AIG node in a window, structural choices cannot be recorded for the node but can be recorded for the POs of the window. These choices can be used independently from each other, since each combines two AIG nodes with the same global function.

3 Optimization and resynthesis algorithm

During optimization, nodes are visited in some order and optimized one at a time. It was found that a reverse topological order leads to larger reductions for small benchmarks optimized as a whole [15], but for large network processed with windowing, the order appears to be unimportant. As a result, a topological order is often used because it is simpler to compute.

```
nodeOptimization( node, parameters ) {
    // compute window for the node with the given parameters
    window = nodeWindow( node, parameters );
    // collect candidate divisors of the node
    divisors = nodeDivisors( node, window, parameters );
    // find sets of resubstitution candidates using simulation as a filter
    cands = nodeResubCandsFilter( node, window, parameters );
    // 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, window, 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( sat_solver, node );
        // update the network by replacing the current node
        nodeUpdate( node, best_cand, best_func );
    }
}
```

Figure 3.0. Don't-care-based optimization of a node.

Figure 3.0 shows a self-explanatory pseudo-code of a node optimization procedure based on structural analysis (windowing), satisfiability, SAT solving, and interpolation.

The parameters used by this procedure include the following:

- the number of fanin/fanout levels of the window to compute,
- the maximum number of window PIs and internal nodes,
- 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 cost based on the goal of resynthesis.

The following subsections provide details on the theory and implementation of each part of the above resynthesis procedure.

3.1 Windowing

This subsection describes improvements to the windowing algorithm presented in [15] and [16].

3.1.1 Overview

Figure 3.1.1 summarizes the improved 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 returns the window composed of nodes found on the paths between the pivot and the PIs.

```
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 );
}
```

Figure 3.1.1. Improved windowing algorithm.

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 going through the pivot. This traversal is useful to detect the 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 do not 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.

3.1.2 Modifications compared to the previous algorithm

Since developing windowing for don't-care computation in [15], the windowing algorithm was used in several projects. A major drawback was found to be non-robustness when applying windowing for large designs containing nodes with more than 100 fanouts. The original algorithm involved topological traversals of the network from the window PIs in order to find the window POs. Nodes with many fanouts, each of which had to be visited, led to a substantial slow-down during this traversal. The problem was aggravated by the fact that multiple-fanout nodes were involved in many windows and, therefore, had to be traversed many times.

This led to the following modification. The original algorithm first detected the window PIs, then the window POs. The current algorithm does the opposite: it performs a shallow topological traversal to detect the window POs, followed by a deeper reverse-topological traversal from the POs to find the window PIs. The topological traversal is performed with a fanout limit set to 10. The limit stops the traversal at multiple-fanout nodes and assumes them to be the window POs because they are unlikely to yield any observability don't-cares (due to many outgoing paths). After this modification, windowing, which previously took about 75% of runtime on some benchmarks, now takes on average 5%.

Another important improvement, reflected in the pseudo-code in Figure 3.1.1, is that only those window POs are computed that have reconvergence involving the pivot node and the window PIs. The POs without reconvergence are not included in the window because they do not contribute don't-cares.

Once windowing for a node is completed, the window is considered as the network for the purpose of don't-care computation. For this reason, the don't-care computation can be considered in the context of a network.

3.2 Computing Boolean divisors

Boolean divisors are all the nodes of a window that can be used as resubstitution candidates of the pivot node. Presented below is an outline of an efficient algorithm for collecting the divisors.

First, window PIs are divided into (a) those in the TFI node of the pivot node, and (b) the remainder. All nodes on paths between the pivot and PIs of type (a) are added to the set of divisors, excluding the node itself and the node's MFFC (although it may be advantageous to include some of the MFFC nodes when re-synthesizing for delay). 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 divisors. In most cases, collecting up to 100 divisors works well in practice, while taking only about 5% of the resynthesis runtime.

3.3 Filtering resubstitutions using simulation

This subsection discusses selection of the resubstitution candidates using simulation. Given are (a) the node with its function $f(x)$ and care set $C(x)$, (b) a target fanin of the node, $p_k(x)$, and (c) a set of candidate divisors, $\{d_j(x)\}$. The goal is to find sets of resubstitution candidates, $\{g_i(x)\}$, which include all fanins except p_k, and possibly one or two divisors from the set $\{d_j(x)\}$, such that function $f(x)$ can be re-expressed in terms of $\{g_i(x)\}$.

First, random and/or guided simulation is performed to find two sets of the PI patterns, each containing N patterns. Patterns in both sets are in the care set: $C(x) = 1$. Patterns in the first (second) set belong to the on-set (off-set) of node's function: $f(x) = 1$ (0). The number N determines the amount of simulation performed. In our current experiments, N varies between 64 and 256.

For some nodes, random simulation does not produce enough patterns of some type. In this case, a SAT solver and distance-1 simulation from counter-examples is used to enumerate through the patterns of the type that is hard to obtain.

Next, the patterns from the two sets are paired, and bit-matrices composed of N^2 entries are constructed for the target node, its fanins, and the candidate divisors. An entry (i, j) of a matrix is set to 1 if the corresponding node distinguishes pattern i in the first set from pattern j in the second set, i.e. different values for these patterns are produced by simulation. By construction, the bit-matrix of the pivot node is filled with 1s.

The bit-matrix for a node is a handy representation of the minterm pairs distinguished by the node global function. Filtering of candidates is done using the necessary and sufficient condition of resubstitution, formulated in Section 2.3. Thus, the bitwise-OR of bit-matrices is created for all fanins of the node, except p_k. If the resulting bit-matrix is all 1s, the fanin p_k might be removable without the need to add other nodes as fanins. If not, the divisors are scanned for a node (or a combination of nodes) that fills in the remaining 0's of the result. Thus, a node of this type, taken together with the remaining fanins, distinguishes all minterm pairs distinguished by the original node. All such nodes are collected and used to construct as many resubstitution candidate sets by combining them with the remaining fanins.

This simulation method can be formulated in terms of SPFDs, as presented in [18]. Approaches based on a similar type of simulation include [23] and [27].

The following example from [18] illustrates the use of simulation for filtering resubstitution candidates. Consider function $g = (a \oplus b)(b \vee c)$ and two sets of candidate functions: ($y_1 = \overline{a}b$, $y_2 = a\overline{b}c$) and ($y_3 = a \vee b$, $y_4 = bc$). Table 3.3 shows the truth tables of all functions. The set (y_3, y_4) is not a valid resubstitution candidate for g because minterm pair (101, 110), which be found by simulation, is distinguished by g but not distinguished by y_3 and y_4. Meanwhile, the set (y_1, y_2) satisfies the condition of resubstitution (see Section 2.3) because all the minterm pairs distinguished by g are also distinguished by at least one function in the set.

Table 3.3. Checking resubstitution using simulation.

$a\ b\ c$	g	Set 1		Set 2	
		$y_1 = \overline{a}b$	$y_2 = a\overline{b}c$	$y_3 = a+b$	$y_4 = bc$
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

3.4 Checking resubstitution using SAT

Checking the legality of a candidate resubstitution set for function f is performed by generating a SAT instance, which reflects the condition of resubstitution formulated in Section 2.3.

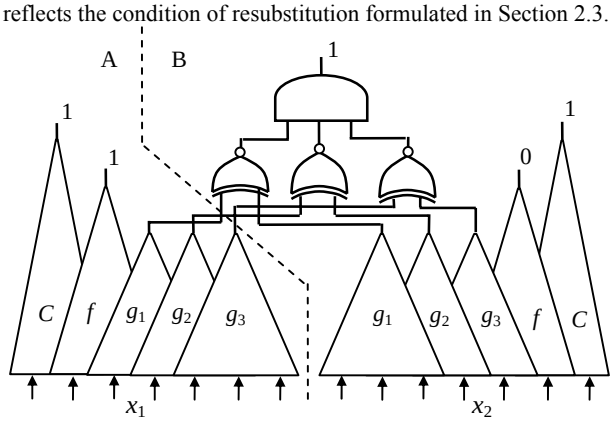

Figure 3.4. Checking resubstitution using SAT.

Figure 3.4 shows the circuit representation of the SAT instance. The left and right parts of the figure show structurally identical logic cones for the care set C, the node's function f, and candidate functions $\{g_i\}$ expressed using variables x_1 and x_2, respectively. An assignment of variables x_1 and x_2 represents two minterms. The circuitry in the middle expresses the condition that the functions $\{g_i\}$ are equal for these minterms. The output of f is set to 1 on the left and 0 on the right, meaning that f takes different values for these minterms. Finally, the left and right care sets C are set to 1 to restrict both minterms to be in the care set.

If the SAT instance is satisfiable, then resubstitution with the given functions $\{g_i\}$ in the candidate set does not exist, and the computation moves on to check other candidate sets. If the SAT instance is unsatisfiable, the resubstitution is proved to exist and the resubstitution function can be derived as shown below.

3.5 Deriving dependency function using SAT

The next goal is to find function $h(g)$, such that $h(g(x))$ can replace $f(x)$ on the care set $C(x)$, that is, $C(x) \Rightarrow [h(g(x)) \equiv f(x)]$. The dependency function $h(g)$ expresses the node in terms of $\{g_i\}$. It can be used to simplify or restructure the network.

Previously the computation of $h(g)$ is done using SOPs [25], BDDs [10], SPFDs [5] or by enumerating satisfying assignments of a SAT problem [15]. Instead, we follow the approach of [11], which relies on interpolation (see Section 2.5). The advantage of using interpolation is that $h(g)$ is computed as a by-product of checking feasibility of a resubstitution candidate.

To compute the interpolant implementing the dependency function $h(g)$, the clauses of the SAT instance are divided into subsets A and B, as shown in Figure 3.4 using the dashed line. In this case, the common variables are the outputs of the functions g_i. They constitute the support of the resulting dependency function.

The proof of unsatisfiability needed for interpolation is generated as shown in [9]. For this, the SAT solver [7] is minimally modified (by adding 5 lines of code) to save both the original problem clauses and the learned clauses derived by the solver. If the instance is unsatisfiable, the last clause derived is the empty clause, which is also added to the set of saved clauses.

The interpolation package 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 the order of their generation. For each learned clause, a fragment of the resolution proof is computed and converted into an interpolant on-the-fly. The interpolant of the last (empty) clause is returned.

Since in most applications (for example, netlist rewiring) the support of the dependency function is small (and equal to the largest node size plus some additional inputs), the interpolant can be computed using truth tables. This is in contrast to the general case when the interpolant is constructed as a multi-level circuit. The above 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.

3.6 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, (b) the priority of the targets. The targets are considered in the order of their priority. For each target, a window is computed and a set of candidate divisors is collected (using resource limits). The candidate divisors are the nodes whose support is a subset of the window PIs and whose arrival time does 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 3.0.

The following subsections discuss several types of resynthesis.

3.6.1 Area minimization

When area minimization is attempted, the network is scanned to find the nodes having (a) large MFFC and (b) a reference counter equal to 1 (has only one fanout). A node satisfying these criteria has a good potential for area saving if the function of its fanout can be expressed without this node.

3.6.2 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, or this fanin can be replaced with another node used in the network, one edge is saved.

3.6.3 Delay minimization

Delay optimization is performed by detecting timing critical edges. For level-driven optimization, an edge is critical if (1) both the driving and loading nodes are critical (2) the difference of the logic level of driving and loading node is one. A node is critical is at least one of the fanin edge is critical. The priority of an edge depends on the number of critical paths the edge is presented on. Each critical edge is targeted by resubstitution.

4 Experimental results

The new SAT-based resynthesis package based on windowing, resubstitution, SAT solving, and interpolation, described in the present paper, was implemented in ABC [2]. The SAT solver used is a modified version of MiniSat-C_v1.14.1 [7].

The experiments targeting 6-input LUTs were run on an Intel Xeon 2-CPU 4-core computer with 8Gb of RAM. The resulting networks were all verified using the combinational equivalence checker in ABC (command "cec") [19].

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

- *resyn* is a logic synthesis script that performs 5 iterations of AIG rewriting [17]
- *resyn2* is a logic synthesis script that performs 10 iterations of AIG rewriting, which are more diverse than those of *resyn*

- *choice* is a logic synthesis script that allows for accumulation of structural choices; *choice* runs *resyn* followed by *resyn2* and collects three snapshots of the current network: the original, the final, and the intermediate one saved after *resyn*
- *if* is an efficient FPGA mapper with priority cuts [21], fine-tuned area recovery, and the capacity to map over a subject graph with structural choices (the mapper was run with the following settings: at most 12 6-input priority cuts are stored at each node; five iterations of area recovery are performed, three with area flow and two with exact local area)
- *imfs* is the new logic optimization and resynthesis engine described in the present paper.

The benchmarks used in this experiment are 20 large public benchmarks from MCNC and ISCAS'89 suites used in the previous work on FPGA mapping [6][20]. (In this set, circuit s298 was replaced by i10 because it contains only 24 6-LUTs.) The experimental results are shown in Table 4.1. The performance of resynthesis on 5 out of 20 selected benchmarks was different, which motivated presenting them as a separate table (Table 4.2).

Tables 4.1 and 4.2 list results for three different runs:

- Section "Baseline" corresponds to a typical run of tech-independent synthesis followed by default technology mapping (*resyn; resyn2; if*)
- Section "Choices" corresponds to four iterations of mapping with structural choices (*choice; if*).
- Section "Imfs" corresponds to four iterations of technology mapping with structural choices, interleaved with the proposed resynthesis (*choice; if*; *imfs*).

Both tables show the number of primary inputs (column "PIs"), primary outputs (column "POs"), registers (column "Reg"), area calculated as the number of 6-LUTs (columns "LUT") and delay calculated as the depth of the 6-LUT network (columns "Level"). The ratios in the tables are the geometric averages of the corresponding ratios reported in the columns.

Table 4.1 shows that iterative mapping with structural choices reduces area and delay by 7.1% and 7.7%, respectively, compared to the baseline tech-independent synthesis and mapping. We conjecture that improvement is due to repeated re-computation of structural choices by applying logic synthesis to the previously mapped network. When the resulting subject graph with choices is mapped again, those logic structures tend to be selected that offer an improvement, compared to the previous mapping. Several iterations of this evolutionary process lead to logic restructuring favorable for the selected LUT size and delay constraints.

When the proposed resynthesis is included in the iteration, the area is improved by 6.0% and delay by 2.3%, compared to iterating mapping without resynthesis. This improvement is likely because the proposed resynthesis allows for an even deeper restructuring of the subject graph that is particularly favorable for area minimization. Applying logic synthesis to improve the results of resynthesis, running mapping, and iterating this process turns out to be an even better form of logic restructuring.

While running experiments on the selected benchmarks, it was observed that 5 circuits included in Table 4.2 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 multi-level 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 reduced delay by 9.2% on these 5 examples, but achieved only negligible area reduction on average. This is because the tech-independent synthesis and mapping are based on mostly local transforms and so they are still subject to the structural bias, albeit less so than mapping without structural choices.

Applying resynthesis to these benchmarks as part of the iterative synthesis and mapping flow leads to dramatic improvements in both area and delay (5x in area and 21% in delay). This shows that the proposed SAT-based resubstitution can cope with a substantial structural bias and gradually derives new logic structures that are more suitable for 6-LUT mapping.

It was noted that some of the 5 circuits shown in Table 4.2 have don't-cares in their original PLA descriptions. It is unknown whether the don't-cares were used during multi-level synthesis that produces the benchmark circuits. In any case, they were not available during mapping and resynthesis, which in their current implementation work on completely-specified multi-level AIGs.

Finally, we mention the runtime taken by the commands reported Tables 4.1 and 4.2. For most of the benchmarks the runtime for the computation reported in Section "Imfs" took less than 20 sec. The longest runtime was observed for the following three: *clma* (572 sec), *pdc* (96 sec), and *spla* (53 sec).

Fine-tuning of the current implementation is not yet finished, and we expect the proposed resynthesis (command "imfs") to be about 3x faster without quality degradation after simulation and Boolean satisfiability are integrated more tightly. Improved runtime measurements, with a break-down for different aspects of resynthesis, may be included in the final version of the paper.

5 Conclusions and future work

The paper proposes an integrated SAT-based logic optimization useful as part of tech-independent synthesis and as a new post-mapping 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:

- Better integration of SAT counter-examples and simulation.
- Fine-tuning resynthesis to focus on delay, placement, and other cost function.
- Using bi-decomposition [14] to perform resynthesis with don't-cares to minimize the total number of AIG nodes.

Acknowledgements

This work was supported in part by SRC contracts 1361.001 and 1444.001, and the California Micro Program with industrial sponsors Actel, Altera, Calypto, Magma, Synopsys, and Synplicity.

References

[1] Altera Corp., *Quartus II University Interface Program*, www.altera.com/education/univ/research/unv-quip.html

[2] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 70909. http://www.eecs.berkeley.edu/~alanmi/abc/

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

[4] K.-C. Chen and J. Cong, “Maximal reduction of lookup-table-based FPGAs”, *Proc. DATE ’92*, pp. 224-229.

[5] J. Cong, Y. Lin, and W. Long, “SPFD-Based Global Rewiring,” *Proc. FPGA ‘02*, pp. 77-84.

[6] D. Chen and J. Cong. “DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs,” *Proc. ICCAD ’04*, pp. 752-757.

[7] N. Een and N. Sörensson, “An extensible SAT-solver”. *SAT ‘03.* http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/

[8] N. Een, A. Mishchenko, and N. Sorensson, “Applying logic synthesis to speedup SAT”, *Proc. SAT ’07* (to appear).

[9] E. Goldberg and Y. Novikov, “Verification of proofs of unsatisfiability for CNF formulas”, *Proc. DATE ’03*, pp. 886-891.

[10] V. N. Kravets and P. Kudva, “Implicit enumeration of structural changes in circuit optimization”, *Proc. DAC ’04*, pp. 438-441.

[11] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. “Scalable exploration of functional dependency by interpolation and incremental SAT solving”, *ERL Technical Report*, EECS Dept., UC Berkeley, Dec. 2006.

[12] K.L. McMillan, “Interpolation and SAT-based model checking”. *Proc. CAV ‘03*, pp. 1-13, LNCS 2725, Springer, 2003.

[13] K. McMillan, “Don't-care computation using *k*-clause approximation”, *Proc. IWLS ’05*, pp. 153-160.

[14] A. Mishchenko, B. Steinbach, and M. A. Perkowski, "An algorithm for bi-decomposition of logic functions", *Proc. DAC '01*, pp. 103-108.

[15] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *Proc. DATE '05*, pp. 418-423. http://www.eecs.berkeley.edu/~alanmi/publications/2005/date05_satdc.pdf

[16] A. Mishchenko and R. K. Brayton, “Scalable logic synthesis using a simple circuit structure”, *Proc. IWLS ’06*, pp. 15-22. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_sls.pdf

[17] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", In Proc. DAC '06, pp. 532-536. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf

[18] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks", *IEEE T CAD*, Vol. 25(5), May 2006, pp. 743-755. http://www.eecs.berkeley.edu/~alanmi/publications/2005/ tcad05_s&s.pdf

[19] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", Proc. ICCAD '06, pp. 836-843. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iccad06_cec.pdf

[20] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs". *IEEE TCAD*, Vol. 26(2), Feb 2007, pp. 240-253. http://www.eecs.berkeley.edu/~alanmi/publications/2006/tcad06_map.pdf

[21] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, “Combinational and sequential mapping with priority cuts”, *Proc. ICCAD ’07*. http://www.eecs.berkeley.edu/~alanmi/publications/2007/iccad07_map.pdf

[22] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, “The transduction method-design of logic networks based on permissible functions”, *IEEE Trans. Comp,* Vol.38(10), pp. 1404-1424, Oct 1989.

[23] S. Plaza, K.-H. Chang, I. L. Markov, and V. Bertacco, “Node mergers in the presence of don't cares", *Proc. ASP-DAC’07*, pp. 414-419.

[24] N. Saluja and S. P. Khatri, ”A robust algorithm for approximate compatible observability don't care (CODC) computation”, *Proc. DAC ’04*, pp. 422-427.

[25] H. Savoj. *Don't cares in multi-level network optimization*. Ph.D. Dissertation, UC Berkeley, May 1992.

[26] E. Sentovich et al. “SIS: A system for sequential circuit synthesis.” *Technical Report, UCB/ERI, M92/41, ERL*, Dept. of EECS, UC Berkeley, 1992.

[27] Y.-S. Yang, S. Sinha, A. Veneris, and R. K. Brayton, “Automating logic rectification by approximate SPFDs," *Proc. ASP-DAC ’07*.

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

Table 4.1. The results of resynthesis after technology mapping (K = 6).

Designs	PI	PO	Reg	Baseline		Choices		Imfs	
				LUT	Level	LUT	Level	LUT	Level
alu4	14	8	0	821	6	785	5	558	5
apex2	39	3	0	992	6	866	6	806	6
apex4	9	19	0	838	5	853	5	800	5
bigkey	263	197	224	575	3	575	3	575	3
des	256	245	0	794	5	512	5	483	4
diffeq	64	39	377	659	7	632	7	636	7
dsip	229	197	224	687	3	685	2	685	2
elliptic	131	114	1122	1773	10	1824	9	1820	9
frisc	20	116	886	1748	13	1671	12	1692	12
i10	257	224	0	589	9	560	8	548	7
misex3	14	14	0	785	5	664	5	517	5
s38417	28	106	1636	2684	6	2674	6	2621	6
s38584	12	278	1452	2697	7	2647	6	2620	6
seq	41	35	0	931	5	756	5	682	5
tseng	52	122	385	647	7	649	6	645	6
Ratio1				**1.000**	**1.000**	**0.929**	**0.923**	**0.873**	**0.901**
Ratio2						**1.000**	**1.000**	**0.940**	**0.977**

Table 4.2. The results of resynthesis after technology mapping for circuits derived from PLAs (K = 6).

Designs	PI	PO	Reg	Baseline		Choices		Imfs	
				LUT	Level	LUT	Level	LUT	Level
clma	383	82	33	3323	10	2715	9	1277	8
ex1010	10	10	0	2847	6	2967	6	1282	5
ex5p	8	63	0	599	5	669	4	118	3
pdc	16	40	0	2327	7	2500	6	194	5
spla	16	46	0	1913	6	1828	6	289	4
Ratio1				**1.000**	**1.000**	**0.995**	**0.908**	**0.212**	**0.718**
Ratio2						**1.000**	**1.000**	**0.213**	**0.790**