Scalable Don't-Care-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

We describe an optimization method for combinational and sequential logic networks, with emphasis on scalability and the scope of optimization. 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 and interpolation to find a new representation for a node. This representation can be in terms of inputs from other nodes in the window thus effecting Boolean re-substitution. Experimental results on 6-input LUT networks after high effort synthesis show substantial reductions in area and delay. When applied to 20 large academic benchmarks, the LUT count and logic level is 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 count and logic level is reduced by 11.8% and 16.5%, respectively. Experimental results on 6-input LUT networks after high-effort synthesis show substantial reductions in area and delay. 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*
J.6 [**Computer-Aided Engineering**]: *Computer-Aided Design (CAD)*

General Terms

Algorithms, Performance, Experimentation

Keywords

FPGA, Technology Mapping, Logic Optimization, Windowing, Boolean Satisfiability, Interpolation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'09, February 22–24, 2009, Monterey, California, USA.
Copyright 2009 ACM 978-1-60558-410-2/09/02...$5.00..

1 Introduction

The size of the industrial FPGAs have grown significantly in the last several years. Virtex-5 FPGA Family [32] contains up to 207K 6-LUTs. Stratix IV Family [33] contains up to 212K Adaptive Logic Modules (ALMs). It is expected that the capacity of FPGAs will continue growing in the coming years. As a result, scalability is becoming a challenging issue 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. Boolean re-substitution is done by representing additional signals (not involved in the node being simplified) using SDCs. If there is a set of external don't cares (EXDCs), it is usually represented by BDDs or by a separate Boolean network in terms of the input variables. These methods are not scalable for many reasons and thus all should be avoided.

We give a method that avoids these non-scalable techniques. Scalability is achieved by:

- optimizing a node in a local window as in [19],
- extracting and representing EXDCs in terms of clauses on *k*-cuts in an underlying AIG of the network [4], and
- avoiding SOPs, BDDs, and Espresso and instead using SAT, truth tables, and interpolation [13].

These ideas are implemented in the system, ABC [1], 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, which restrict the states to an over approximation of the reachable states. These methods are shown experimentally to be scalable and effective.

The rest of the paper 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, simulation, SAT solving, interpolation and use of care clauses. Section 5 reviews relevant previous work. Section 6 reports experimental results. Section 7 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.

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. The *primary outputs* (POs) are a subset of the nodes of the network. The transitive fanin (TFI) of a node includes the node and the nodes in its transitive fanin, including the PIs. The transitive fanout (TFO)

of a node includes the node and the nodes in its transitive fanout, including the POs. 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 described in this paper are applicable to both AIGs and general-case logic networks.

A *cut C* of node *n*, called *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. Non-controllability occurs because some combinations of values are never produced at the fanins. Non-observability occurs because the node's effect on the POs is blocked under some combination of the PI values. Examples can be found in [16].

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* is complemented [16].

Traditionally, subsets of don't-cares are derived and used to optimize a node [28][16]. 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 re-expressed using the new fanins without changing the functionality of the network.

A necessary and sufficient *condition of the existence of resubstitution* is given in [19] (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 $C(x_1) \wedge C(x_2) = 1$.*

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, e.g. 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 compatibility 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 PIs. Typically EXDCs are hard to represent and generally hard to project to a window. Therefore, the use of EXDCs is 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 in sequential circuits 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 [13] (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, e.g. 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 over-approximates the reachable state space of a sequential circuit. This is done by induction and is based on ideas described in more detail in a separate paper [4]. Once extracted, the set of clauses is stored and used by the ABC command *mfs*. This algorithm is described in Section 4.2.

We compute a set of clauses on a set *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 [2], 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* - it holds in the initial state, and *inductive case* - 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 [9].

In our method, the invariant is a set of clauses, in which a group of variables participating in a clause is derived using efficient k-cut computation, which is adopted from LUT-based technology mapping [8][25]. It avoids exhaustive k-cut enumeration and computes only a small subset of useful cuts using priority heuristics similar to those in [22].

An initial set of candidate clauses is detected using simulation. Two types of random simulation are used, combinational and sequential. Sequential simulation is such that it starts at the initial state and only visits reachable states from there. Minterms at a k-cut that appear under combinational but not under sequential simulation are recorded. Then a candidate clause is derived as the complement of such a minterm. This initial candidate set of 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 over-approximation to the set of reachable states if the initial state satisfies the invariant.

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 N of times a minterm appears in combinational simulation but never in sequential simulation. Such minterms are more likely being 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 usually 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 [3]. 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 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 4.0 shows pseudo-code 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.

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

```
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 );
    }
}
```

Figure 4.0. Don't-care-based optimization of a 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.

4.1.1 Overview

Figure 4.1.1 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 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 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.

```
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 4.1.1. Improved windowing algorithm.

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 can 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.

4.2 Computing the care set for an inner window

The configuration shown in Figure 4.2.0 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 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 4.2.0 are additional constraints that limit the scope of satisfiable assignments and therefore contribute to the don't-cares 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 4.2.1, 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, m_s, 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*.

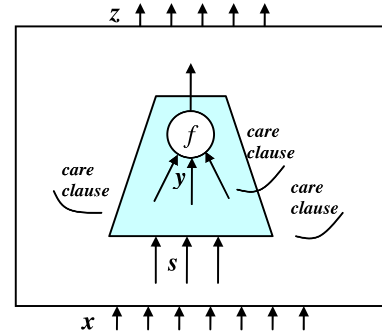

Figure 4.2.0. Inner and outer windows for node *f*.

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

When there are care clauses, we look for those whose clause variables are all contained in the outer window (illustrated in Figure 4.2.0). If any are found, these are added to the set of clauses generated for the outer windows (in both copies as shown in Figure 4.2.1) to form the SAT instance. Optionally, if there are care clauses that have only a few variables not in the outer window, we can expand the outer window to include these as PIs.

The SAT-based care computation is summarized in Figure 4.2.2. 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 4.2.1). The resulting compacted AIG *G* is constructed in one DFS traversal of the nodes in the miter, without actual duplication.

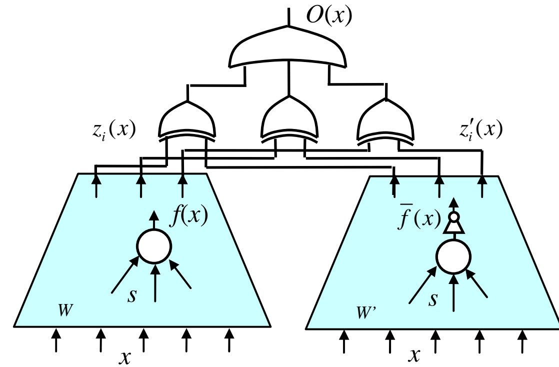

Figure 4.2.1. Miter for care set computation.

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.

```
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;
}
```

Figure 4.2.2. Pseudo-code of SAT-based care computation.

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 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 re-express $f(y)$ 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 free 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 taking only about 5% of the resynthesis runtime.

The following example from [19] illustrates the use of simulation for filtering resubstitution candidates. Consider function $f = (a \oplus b)(b \vee c)$ and two sets of candidate functions: ($g_1 = \overline{a}b$, $g_2 = a\overline{bc}$) and ($g_3 = a \vee b$, $g_4 = bc$). Table 4.3 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 the condition of Theorem 2.2.1 because all the minterm pairs distinguished by *f* are also distinguished by either g_1 or g_2.

Table 4.3. Checking resubstitution using simulation.

a b c	*f*	Set 1		Set 2	
		$g_1 = \overline{a}b$	$g_2 = a\overline{bc}$	$g_3 = a+b$	$g_4 = bc$
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

4.4 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.

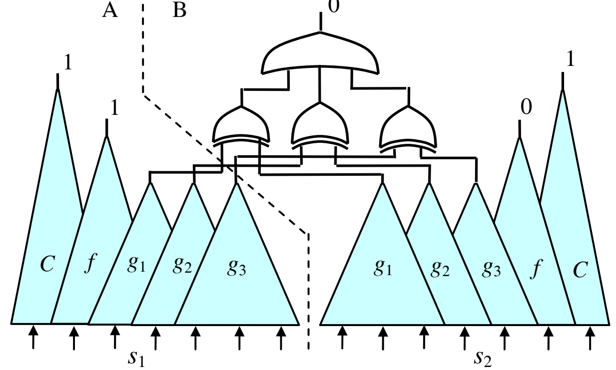

Figure 4.4. Miter for checking resubstitution using SAT.

Figure 4.4 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 don't cares in this computation are used not to extract the care set $C(s)$, as described in Section 4.2, but to simply put in the miter in Figure 4.2.1 in place of *C*, in both copies in Figure 4.4, with variable s_1 and s_2 kept in common.

If the SAT instance is satisfiable, then resubstitution with the given functions $\{g_i\}$ in the resubstitution set does not exist and it 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 is proved to exist and a dependency function $h(g)$ can be derived, as shown below.

4.5 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 fit 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 [28], BDDs [11], SPFDs [7] or by enumerating satisfying assignments of a SAT problem [16]. We follow the approach of [12] based on interpolation (see Section 2.5). The advantage is that $h(g)$ is computed as a byproduct of the resubstitution feasibility check, discussed in Section 4.4.

To compute a function $h(g)$, the clauses of the SAT instance are divided into subsets $A(x, y)$ and $B(x, y)$ derived from the two parts of the circuit separated by the dashed line, as shown in Figure 4.4. In this case, the common variables, *y*, of the

interpolant 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 [10]. For this, the SAT solver [9] 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 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.

4.6 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 non-existing variable, and for each missing variable finding a cut in terms of existing variables. Then compute the pre-image 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 pre-image, 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 pre-image of the complement of this clause and derive a set of equivalent clauses used in computing the care set of the window.

4.7 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 according to the pseudo-code in Figure 4.0.

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.

5 Previous work

Technology-independent optimization and post-mapping resynthesis of Boolean networks using internal flexibilities have long history [24][28][16][11][5], to mention a few publications. Traditional don't-care-based optimization [28] is part of the high-effort logic optimization flow in SIS [29]. 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 [16][27].

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 [14], which optimized nodes "in-place", without explicitly computing don't-cares. However, unlike [28][11], that work does not allow for resubstitution. As a result, the optimization space is limited to the current node boundaries. Another recent method [12] performs efficient SAT-based resubstitution but does not consider don't-cares, which may limit its optimization potential.

Some recent papers [31][26] 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 [6] 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.

6 Experimental results

The SAT-based resynthesis is implemented in ABC [1] 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 [9]. The algorithm is applicable to a mapped network and attempts resubstitution for each gate or LUT in the netlist. Experiments targeting 6-input LUTs implementations 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 (command *cec*) in ABC [20].

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 [18]
- *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 efficient FPGA mapper, which uses priority cuts [22], delay-optimal mapping, fine-tuned area recovery, and the capacity to map over a subject graph with structural choices[1].
- *mfs* is the new resynthesis command of this paper.

The benchmarks used were 20 large public benchmarks from the MCNC and ISCAS'89 suites used in previous work on FPGA mapping [8][21][2]. The results for these benchmarks are shown in Table 5.1[3]. An additional experiment was performed using 14 industrial benchmarks, and the results are shown in Table 5.2.

Tables 5.1 and 5.2 list results for four different runs:

- Section "Baseline" corresponds to a typical run of tech-independent synthesis [18] followed by default technology mapping (*dc2 –l; dc2 –l; if –C 12*)
- Section "Choices" corresponds to four iterations of mapping with structural choices (*st; dch; if –C 12*) [21] and picking the best result after any iteration.
- Section "mfs" corresponds to four iterations of technology mapping with structural choices, interleaved with the proposed resynthesis (*st; dch; if –C 12; mfs –W 4 –M 5000*) and picking the best result after any iteration.
- Section "exdc" in Table 5.2 shows the results of applying one round of optimization with artificial don't cares, as explained below..

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") 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.

The results listed in Tables 5.1 and 5.2 show that, compared to the baseline synthesis and mapping, the iterative mapping with structural choices reduces area and delay by 4.8% and 2.4% (for academic benchmarks) and by 5.1% and 9.3% (for industrial benchmarks). The improvement is likely 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 structures tend to be selected that offer an improvement, compared to the previous mapping. Several iterations of this evolutionary process yield restructuring favorable for the selected LUT size and delay constraints.

When the proposed don't-care-based resynthesis is included in the iteration, the area and delay are additionally reduced by 42.2% and 10.0% (for academic benchmarks) and by 7.0% and 7.9% (for industrial benchmarks). Without the five outlier circuits discussed below, the additional reduction for academic benchmarks is 13% in area and 2% in delay. 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.

It was observed that 5 circuits in Table 5.1 (*cpla, 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 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 produced much smaller improvements than the proposed resubstitution. This is because the tech-independent synthesis and mapping with choices rely on 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 leads 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. 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 academic benchmarks in Table 5.1 have don't-cares in their original PLA descriptions. It is not known whether the don't-cares were used during multi-level synthesis that produces the circuits. In any case, they were not available during mapping and resynthesis, which worked on multi-level circuits without don't-cares generated by another tool.

It was found experimentally that EXDCs computed as described in Section 3 did not produce noticeable area and delay savings for the industrial circuits in Table 5.2. Therefore, to demonstrate the potential of minimization with external don't-cares, artificial don't-care conditions were generated for these circuits, assuming that all their inputs are one-hot-encoded. Applying one round of resubstitution with these don't-cares reduced area and delay of the circuit by 31.3% and 2.6%, respectively. These results are shown in the last section of Table 5.2.

We emphasize that the experiment with artificial external don't-cares is performed to demonstrate the scalability of the proposed resubstitution with don't-cares. The resulting circuits are equivalent to the original ones only on the generated care-set.

7 Conclusions and Future Work

The paper proposes an integrated SAT-based logic optimization methodology 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:

- Fine-tuning resynthesis to focus on delay, power, placement, and other cost function.
- Using bi-decomposition [15] to perform resynthesis with don't-cares to minimize the total number of AIG nodes.

---

[1] 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.

[2] Circuit s298 was replaced by i10 because it contains only 24 6-LUTs.

[3] A similar version of these experimental results appeared in [23] to show the orthogonal nature of the optimization method reported in that paper.

Acknowledgements

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, Intel, Intrinsity, Magma, Synopsys, Synplicity, Tabula, and Xilinx. We also thank the anonymous reviewers for their helpful comments.

References

[1] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 80802. http://www.eecs.berkeley.edu/~alanmi/abc/

[2] P. Bjesse and K. Claessen. "SAT-based verification without state space traversal". *Proc. FMCAD'00*. LNCS, Vol. 1954, pp. 372-389.

[3] M. L. Case, A. Mishchenko, and R. K. Brayton, "Inductively finding a reachable state space over-approximation", *Proc. IWLS '06*, pp. 172-179. http://www.eecs.berkeley.edu/~alanmi/publications/2006/iwls06_inv.pdf

[4] M. L. Case, A. Mishchenko, and R. K. Brayton, "Cut-based inductive invariant computation", *Proc. IWLS'08*, pp. 253-258. http://www.eecs.berkeley.edu/~alanmi/publications/2008/iwls08_ind.pdf

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

[6] K.-C. Chen and J. Cong, "Maximal reduction of lookup-table-based FPGAs", *Proc. DATE '92*, pp. 224-229.

[7] J. Cong, Y. Lin, and W. Long, "SPFD-based global rewiring," *Proc. FPGA '02*, pp. 77-84.

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

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

[10] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas", *Proc. DATE '03*, pp. 886-891.

[11] V. N. Kravets and P. Kudva, "Implicit enumeration of structural changes in circuit optimization", *Proc. DAC '04*, pp. 438-441.

[12] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. "Scalable exploration of functional dependency by interpolation and incremental SAT solving", *Proc. ICCAD '07*, pp. 227-233. http://www.eecs.berkeley.edu/~alanmi/publications/2007/iccad07_fd.pdf

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

[14] K. McMillan, "Don't-care computation using *k*-clause approximation", *Proc. IWLS '05*, pp. 153-160.

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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] A. Mishchenko, R. K. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks", *Proc. ICCAD'08*, pp. 38-44. http://www.eecs.berkeley.edu/~alanmi/publications/2008/iccad08_lp.pdf

[24] 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.

[25] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.

[26] 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.

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

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

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

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

[31] 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.

[32] *Xilinx Virtex-5 Product Table*, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/v5product_table.pdf

[33] *Xilinx Stratix IV FPGA Family Overview*, http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/overview/stxiv-overview.html

Table 5.1. The results of resynthesis after technology mapping (K = 6) for academic benchmarks.

Example	Profile			Baseline			Choices			mfs		
	PI	PO	FF	LUT	Level	Time	LUT	Level	Time	LUT	Level	Time
alu4	14	8	0	845	5	0.46	786	5	2.23	499	5	15.53
apex2	39	3	0	987	6	0.53	922	6	5.80	674	6	33.71
apex4	9	19	0	821	5	0.41	798	5	2.10	786	5	16.41
bigkey	263	197	224	567	3	0.60	567	3	0.86	455	3	1.68
clma	383	82	33	3309	10	1.80	2910	9	16.23	701	7	122.24
des	256	245	0	880	5	0.62	872	4	2.90	638	4	7.88
diffeq	64	39	377	712	7	0.37	690	7	0.80	645	7	2.77
dsip	229	197	224	682	3	0.50	681	3	0.58	677	2	1.65
elliptic	131	114	1122	1877	10	0.85	1914	10	2.20	1813	10	4.80
ex1010	10	10	0	2934	6	1.48	2712	6	17.14	1342	6	101.13
ex5p	8	63	0	593	5	0.37	521	5	1.58	119	3	4.57
frisc	20	116	886	1777	12	1.06	1749	12	7.43	1757	11	16.64
i10	257	224	0	595	9	0.39	554	9	1.37	545	8	9.35
misex3	14	14	0	772	5	0.43	701	5	2.19	368	5	12.11
pdc	16	40	0	2113	7	1.35	1959	6	15.36	128	5	25.91
s38417	28	106	1636	2257	6	1.33	2271	6	7.09	2206	6	26.11
s38584	12	278	1452	2319	7	1.47	2373	7	8.41	2250	6	14.01
seq	41	35	0	872	5	0.50	834	5	4.64	684	5	17.73
spla	16	46	0	1622	6	1.08	1417	6	11.58	161	4	19.12
tseng	52	122	385	717	7	0.30	690	7	0.63	639	7	2.35
Geomean				**1150**	**6.08**	**0.68**	**1095**	**5.93**	**3.28**	**633**	**5.34**	**11.62**
Ratio				**1.000**	**1.000**	**1.000**	**0.952**	**0.976**	**4.831**	**0.550**	**0.878**	**17.101**
Ratio							**1.000**	**1.000**	**1.000**	**0.578**	**0.900**	**3.540**

Table 5.2. The results of resynthesis after technology mapping (K = 6) for industrial benchmarks.

Example	Profile			Baseline			Choices			mfs			exdc		
	PI	PO	FF	LUT	Lev	Time	LUT	Lev	Time	LUT	Lev	Time	LUT	Lev	Time
Design01	1332	5064	5625	15453	8	10.08	14830	8	62.17	13793	7	104.91	10506	7	112.30
Design02	1559	5701	10373	28091	10	21.50	26972	9	134.89	24997	9	312.14	16735	9	333.17
Design03	993	5533	6430	15033	10	7.43	14428	10	40.69	14010	10	118.00	12124	10	122.80
Design04	974	1301	940	2841	31	2.09	2723	30	7.82	2697	30	121.33	1891	30	128.49
Design05	101	198	1177	2649	6	1.60	2554	5	10.86	2222	5	20.80	1286	5	22.15
Design06	68	85	1355	3624	19	2.53	3385	16	27.58	3192	15	102.77	1871	15	106.65
Design07	6598	11151	22382	71637	17	61.73	69747	15	475.84	63116	13	1154.14	41937	13	1277.00
Design08	2126	6451	7075	20504	15	12.27	19860	14	70.61	18943	12	150.09	14352	11	158.00
Design09	2450	4798	3725	9951	4	3.13	9718	4	9.50	9374	3	21.67	8211	3	23.62
Design10	1032	1767	1124	4447	10	2.24	4299	10	15.13	4105	9	44.32	3055	9	49.06
Design11	4040	9406	35654	83113	16	71.99	81601	16	472.68	73478	14	1748.12	48429	14	1861.00
Design12	115	264	2293	5413	7	3.53	5209	6	24.07	4576	6	49.35	2759	6	56.72
Design13	56	87	465	1756	12	1.19	1311	8	8.19	1162	8	27.44	691	6	28.98
Design14	14	60	426	1448	9	0.91	1455	8	8.79	1382	7	34.77	879	7	37.07
Geomean				**8655**	**10.93**	**5.44**	**8215**	**9.92**	**34.33**	**7637**	**9.13**	**102.29**	**5244**	**8.89**	**109.79**
Ratio				**1.000**	**1.000**	**1.000**	**0.949**	**0.907**	**6.310**	**0.882**	**0.835**	**18.801**	**0.606**	**0.814**	**20.182**
Ratio							**1.000**	**1.000**	**1.000**	**0.930**	**0.921**	**2.979**	**0.638**	**0.896**	**3.198**
Ratio										**1.000**	**1.000**	**1.000**	**0.687**	**0.974**	**1.073**